Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Магістерська кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему: «Розроблення алгоритму генерації лабіринтів на основі моделі Амарі»

Виконав: студент 6 курсу групи КНз-61м

спеціальності 122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Флис Т. Б.
(прізвище та ініціали)

Керівник: Капран І. Д., Думанський О. І.
(прізвище та ініціали)

Рецензент: Левкович М. В.
(прізвище та ініціали)

Львів – 2024

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ *комп'ютерних наук та інформаційних технологій*

Кафедра *комп'ютерних наук*

Рівень вищої освіти *другий (магістерський)*

Спеціальність *122 "Комп'ютерні науки"*
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
_____ Борецька І. Б.
"05" січня 2024 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Флис Тетяні Богданівній
(прізвище, ім'я, по батькові)

1. Тема роботи *«Розроблення алгоритму генерації лабіринтів на основі моделі Амарі»*
керівник роботи *Капран Ігор Дмитрович, Думанський Остап Іванович к.ф-м. н., доцент*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від *"13" лютого 2023 року* № С-50
2. Термін подання студентом роботи *05 січня 2024 року*
3. Вихідні дані до роботи *Огляд та аналіз шляхів вирішення поставлених завдань, характеристика процесу генерації лабіринтів з використанням поширених алгоритмів, аналіз технологій для програмної реалізації, організаційна структкра для розробленого алгоритму генерації лабіринтів.*
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) *Вступ. Розділ 1. Стан проблемної області. Розділ 2. Інформаційне забезпечення. Розділ 3. Математичне забезпечення. Розділ 4. Програмне забезпечення. Розділ 5. Розроблення стартап-проекту. Висновки. Список використаної літератури. Додатки.*
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Слайди для доповіді загальним обсягом 10-12 одиниць
6. Дата видачі завдання *15 лютого 2023 року*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел згідно досліджуваної тематики. Збір необхідних матеріалів. Формування функціональних вимог та постановка технічного завдання. Оформлення першого розділу пояснювальної записки.	15.02.2023 р. 20.03.2023 р.	***Виконано***
2	Розроблення математичного та алгоритмічного забезпечення	21.03.2023 р. 24.04.2023 р.	***Виконано***
3	Програмна реалізація та аналіз результатів	25.04.2023 р. 28.09.2023 р.	***Виконано***
4	Розроблення стартап-проекту	02.10.2023 р. 15.11.2023 р.	***Виконано***
5	Оформлення пояснювальної записки та здача на рецензування	20.11.2023 р. 05.01.2024 р.	***Виконано***

Студент _____ (підпис) Флис Т. Б. (прізвище та ініціали)

Керівник роботи _____ (підпис) Капран І. Д. (прізвище та ініціали)

Керівник роботи _____ (підпис) Думанський О. І. (прізвище та ініціали)

РЕФЕРАТ

Дипломна робота містить 60 сторінок пояснювальної записки, 24 рисунків, 1 таблицю, 2 додатки і 15 джерел.

В даному дипломному проекті було проведено огляд та аналіз алгоритмів генерації лабіринтів, їхні особливості, переваги та недоліки. Визначені та проаналізовані основні інформаційні та програмні засоби необхідні для реалізації поставленого технічного завдання. Досліджені параметри моделі Амарі, які впливають на процес генерації лабіринтів. Результатом проведеної роботи є розроблений унікальний алгоритм, який заснований на моделі Амарі (нейронної активності кори головного мозку), що являється безперервним аналогом нейронних мереж. Алгоритм представлений у форматі інтерактивних форм на яких можна змінювати параметри моделі Амарі та характеристики лабіринту. Програмне забезпечення реалізовано за допомогою таких технологій: Python, C++.

Ключові слова: C++, Python, алгоритм, нейрон, мережа, модель.

SUMMARY

The thesis contains 60 pages of explanatory note, 24 figures, 1 table, 2 appendix and 15 sources.

In this diploma project, a review and analysis of labyrinth generation algorithms, their features, advantages and disadvantages was carried out. Identified and analyzed the main information and software tools necessary for the implementation of the technical task. The parameters of the Amari model, which affect the process of generating mazes, are studied. The result of the work is a developed unique algorithm, which is based on the Amari model (neuronal activity of the cerebral cortex), which is a continuous analogue of neural networks. The algorithm is presented in the format of interactive forms on which you can change the parameters of the Amari model and the characteristics of the maze. The software is implemented using the following technologies: Python, C++.

Keywords: C++, Python, algorithm, neural, network, model.

ТЕХНІЧНЕ ЗАВДАННЯ

Необхідно дослідити та розробити алгоритм генерації лабіринтів, який заснований на моделі Амарі (нейронної активності кори головного мозку), що являється безперервним аналогом нейронних мереж. За певних умов дана модель дозволяє створювати гарні лабіринти дуже складної форми.

Завдання включає такі кроки:

- провести огляд літературних джерел щодо алгоритмів генерації лабіринтів;
- охарактеризувати процес генерації лабіринтів з використанням поширених алгоритмів;
- провести аналіз технологій, які знадобляться для програмної реалізації;
- створити графічний інтерфейс для розробленого алгоритму;
- провести тестування розробленого алгоритму генерації лабіринтів на основі моделі Амарі з метою перевірки працездатності.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

A* – A-стар, інформований алгоритм пошуку;

ДРП – дискретне робоче поле;

МР – мобільний робот;

ОБ – алгоритм Олдоса-Бродера;

BFS – Breadth-first search, пошуку в ширину;

BSP – Binary Space Partitioning, алгоритм двійкового дерева;

CBT – Close Beta Testing, закрите бета-тестування;

DFS – Depth-first search, пошук в глибину;

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

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

OBT – Open Beta Testing, відкрите бета-тестування;

RW – Random Walk, випадковий пошук.

ВСТУП

Лабіринти на сьогоднішній день відіграють важливу роль у розважальній сфері. І це не лише дзеркальний лабіринт у парках атракціонів, а й відеоігри, адже якщо подивитися на карти та рівні у різних іграх, то це все один великий та складний лабіринт, побудований за деякими правилами.

Актуальність проблеми. Існує безліч способів створення ідеальних лабіринтів, і кожен із них має власні характеристики. Зокрема цікаві та унікальні лабіринти можна створити використовуючи наявні алгоритми для генерації лабіринтів. Звичайно, у поточних алгоритмів присутні деякі недоліки, які потрібно виправляти. І це потрібно робити шляхом продовження модифікації поточного варіанту, або розробленням нової концепції. Перший варіант вимагає багато змін програмного коду, який необхідно налагоджувати і перевіряти на працездатність. А другий, природно, вимагає вигадати більш вигідний алгоритм.

Об'єктом дослідження в даній роботі є алгоритми генерації та пошуку рішень для лабіринтів. Вони базуються на складних математичних формулах та на сьогодні мають практичне застосування у багатьох сферах.

Предметом дослідження є один унікальний підхід до генерації лабіринтів, який заснований на моделі Амарі (нейронної активності кори головного мозку, що є безперервним аналогом нейронних мереж). За певних умов вона дозволяє створювати красиві лабіринти дуже складної форми.

Метою даної роботи є дослідження та розроблення унікального алгоритму генерації лабіринтів з використанням моделі Амарі. В основі даного алгоритму лежать теоретико-практичні знання з обчислювальної нейробіології.

Новизна роботи. Новизна роботи полягає в тому, що це єдиний, незвичний алгоритм для генерації лабіринтів різної складності. Він заснований на моделі Амарі - нейронної активності кори головного мозку, що є безперервним аналогом нейронних мереж. За певних умов він дозволяє

створювати красиві лабіринти дуже складної форми. Велика кількість користувачів вже стикалася із завданням генерації лабіринтів в тій чи іншій формі і знає, що для її вирішення часто використовують алгоритми Пріма і Крускала, знаходження мінімального дерева у графі, вершини якого є комірками лабіринту, а ребра представляють проходи між сусідніми комірками. Ми ж зробили сміливий крок подалі від теорії графів у бік обчислювальної нейробіології.

Практичне значення одержаних результатів. Даний алгоритм може бути використаний розробниками ігор для процедурної генерації рівнів, перешкод та карт у відеоіграх.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Класифікація лабіринтів

Загалом лабіринти та алгоритми для їх розроблення можна розподілити за сімома різними класифікаціями: розмірності, гіперрозмірності, топології, тесселяції, маршрутизації, текстурі та пріоритету.

Лабіринт може використовувати по одному елементу окремого класу в довільному поєднанні.

Розмірність: клас розмірності насправді визначає, скільки вимірів у просторі заповнює лабіринт. Існують такі типи: двовимірні, тривимірні, багатовимірні, переплетення.

Гіперрозмірність: класифікація по гіперрозмірності стосується розміру об'єкта, який проходить даний лабіринт. Існують такі типи: не-гіперлабіринти, гіперлабіринти, гіпер-гіперлабіринт.

Топологія: клас топології визначає геометрію простору лабіринту, де той існує як одне ціле. Є такі типи: звичайний, пленарний.

Тесселяція: класифікація форми деяких комірок, з яких складається лабіринт. Існуючі типи: ортогональний, дельта, сігма, тета, іпсилон, дзета, омега, фрактальний.

Маршрутизація: класифікація маршрутизації - це, ймовірно, найцікавіший аспект у генерації лабіринтів. Від пов'язаний із типами проходів у межах геометрії, визначеної в описаних вище категоріях. Вони поділяються на: ідеальний, плетений, одномаршрутний, розріджений, частково плетений.

Текстура: класифікація за текстурою описує стиль виходів за різної маршрутизації та геометрії. Ось кілька прикладів змінних: зміщення, прольоти, елітність, симетрія, однорідність, плинність.

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: додавання стін, вирізання проходів, шаблон.

Описане вище в жодному разі не є вичерпним списком всіх можливих класів або елементів усередині кожного класу. Майже кожен тип лабіринту, у тому числі лабіринти з особливими правилами, можна виразити у вигляді орієнтованого графа, в якому буде кінцева кількість станів і кінцева кількість варіантів вибору в кожному стані, і це називається еквівалентністю лабіринтів.

1.2. Алгоритми створення ідеальних лабіринтів

Існує безліч способів створення ідеальних лабіринтів, і кожен із них має власні характеристики. Нижче наведено список конкретних алгоритмів. У всіх них описано створення лабіринту вирізанням проходів, проте якщо не вказано інше, кожен також можна реалізувати додаванням стін.

Recursive backtracker: даний метод дозволяє розв’язувати лабіринти, з використанням стека, пам’ять якого може сягати розмірів лабіринту. У ході вирізанні лабіринт веде себе дуже жадібно, і завжди вирізує прохід в неіснуючій частині лабіринту, якщо вона є поруч із взятою коміркою. Щоразу, коли ми переміщаємося до нового осередку, записуємо попередній осередок у стек. Якщо поруч із поточною позицією немає нестворених осередків, то витягаємо з стека попередню позицію. Лабіринт завершено, коли в стеку більше нічого не залишається. Це призводить до створення лабіринтів з максимальним показником плинності, глухих кутів менше, але вони довші, а рішення зазвичай виявляється дуже довгим і звивистим. При правильній реалізації він виконується швидко, і швидше працюють лише дуже спеціалізовані алгоритми. Recursive backtracking не може працювати з додаванням стін, тому що зазвичай призводить до шляху рішення, що йде по зовнішньому краю, коли вся внутрішня частина лабіринту з'єднана з кордоном одним проходом (рис. 1. 1).

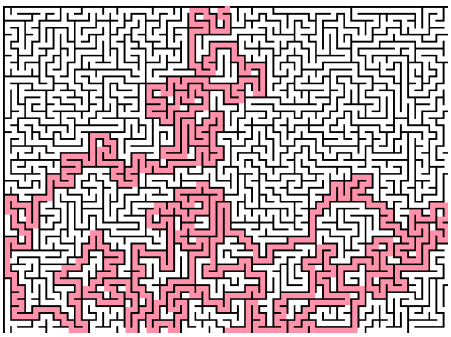


Рисунок 1. 1. Алгоритм Recursive backtracker

Алгоритм Краскала: даний алгоритм створює мінімальне сполучне дерево. Він не «вирощує» лабіринт подібно до дерева, швидше вирізує частини проходів в лабіринті випадковим чином, проте у результаті

утворюється ідеальний лабіринт. Даному алгоритму необхідний обсяг пам'яті, пропорційний розміру лабіринту та можливість перелічення кожного ребра чи стіни між комірками лабіринту у хаотичному порядку. Позначаємо кожен осередок унікальним ідентифікатором, а потім обходимо всі ребра у випадковому порядку. Якщо комірки з обох сторін від кожного ребра мають різні ідентифікатори, то видаляємо стіну і задаємо всім коміркам з одного боку той самий ідентифікатор, що і коміркам з іншого. Якщо комірки на обох сторонах стіни вже мають однаковий ідентифікатор, то між ними вже існує якийсь шлях, тому стіну можна залишити, щоб не створювати петель. Цей алгоритм створює лабіринти з низьким показником плинності, але не таким низьким, як алгоритм Прима. Об'єднання двох безліч по обидва боки стіни буде повільною операцією, якщо кожна комірка має тільки номер і вони об'єднуються в циклі. Об'єднання, а також пошук можна виконувати майже за постійний час завдяки використанню алгоритму об'єднання-пошуку (union-find algorithm): поміщаємо кожну комірку в деревоподібну структуру, кореневим елементом є ідентифікатор. Об'єднання виконується швидко завдяки зрощуванню двох дерев. При правильній реалізації цей алгоритм працює досить швидко, але повільніше більшості через створення списку ребер та управління множинами (рис. 1. 2).

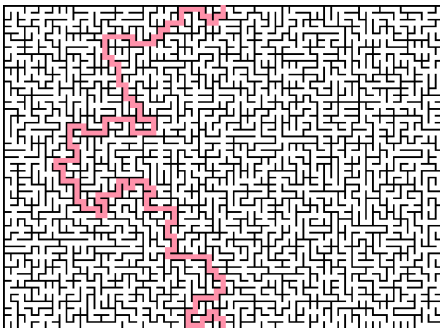


Рисунок 1. 2. Алгоритм Краскала

Алгоритм Пріма: цей алгоритм створює мінімальне сполучне дерево, обробляючи унікально випадкові ваги ребер. Об'єм пам'яті пропорційний розміру лабіринту. Починаємо з будь-якої вершини (готовий лабіринт буде однаковим, з якої б вершини ми не почали). Виконуємо вибір ребра проходу з

найменшою вагою, що з'єднує лабіринт до точки, яка ще не міститься, а потім приєднуємо її до лабіринту. Створення лабіринту завершується, коли більше не залишилося ребер, що розглядаються. Для ефективного вибору наступного ребра необхідна черга з пріоритетом (яка зазвичай реалізується за допомогою купи), що зберігає всі ребра кордону. Тим не менш, цей алгоритм є досить повільним, тому що для вибору елементів з обробки купи вимагає часу log(n). Тому краще віддати перевагу алгоритму Краскала, який теж створює мінімальне сполучне дерево, адже він швидше і створює лабіринти з ідентичною структурою (рис. 1. 3).

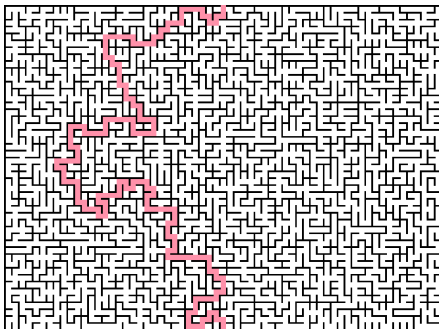


Рисунок 1. 3. Алгоритм Пріма

Алгоритм Ейлера: цей алгоритм являється найшвидшим й не має зміщеності чи недоліків. Він створює лабіринт рядково, після завершення генерації рядка алгоритм більше її не враховує. Кожен осередок у рядку міститься у безлічі; дві осередки належать одній множині, якщо між ними є шлях уже створеним лабіринтом. Ця інформація дозволяє вирізати проходи у поточному рядку без створення петель чи ізольованих областей. Насправді це досить схоже на алгоритм Фарбала, тільки тут формується по одному рядку за раз, у той час як Краска переглядає весь лабіринт. Створення рядка і двох частин: випадковим чином з'єднуємо сусідні у межах рядка осередки, тобто. вирізаємо горизонтальні проходи, потім випадковим чином з'єднуємо комірки між поточним і наступним рядками, тобто. вирізаємо вертикальні проходи. При вирізанні горизонтальних проходів ми не з'єднуємо комірки, що вже знаходяться в одній множині (бо інакше створиться петля), а при вирізанні вертикальних проходів ми повинні з'єднати комірку, якщо вона має

одиничний розмір (бо якщо її залишити, вона створить ізольовану область). При вирізанні горизонтальних проходів ми з'єднуємо комірки, якщо вони знаходяться в однаковій множині (бо тепер між ними є шлях), а при вирізанні вертикальних проходів коли не з'єднуємося з коміркою, поміщаємо її в окрему множину (бо тепер вона відокремлена від решти лабіринту). Створення починається з того, що перед з'єднанням осередків у першому рядку кожен осередок має власну безліч. Створення завершується після з'єднання осередків в останньому рядку. Існує особливе правило завершення: до моменту завершення кожен осередок повинен знаходитися в однаковій множині, щоб уникнути ізольованих областей. (Останній рядок створюється об'єднанням кожної з пар сусідніх осередків, що ще не знаходяться в одній множині). Найкраще реалізовувати безліч за допомогою циклічного двозв'язного списку осередків (який може бути просто масивом, що прив'язує осередки до пар осередків з обох боків тієї ж множини), що дозволяє виконувати за постійний час вставку, видалення та перевірку перебування сусідніх осередків в одній множині. Проблема цього алгоритму полягає у незбалансованості обробки різних країв лабіринту; щоб уникнути плям у текстурах потрібно виконувати з'єднання та пропуск з'єднання осередків у правильних пропорціях (рис. 1. 4).

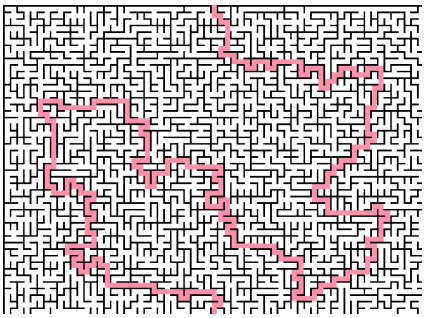


Рисунок 1. 4. Алгоритм Ейлер

Висновки до розділу

У більшості алгоритмів (таких, наприклад, які розглянуті в даному розділі) створюються щільні лабіринти, тобто такі, які мають тільки один вірний шлях і не мають петель. Вони схожі на лабіринти, які публікуються у газетних розділах.

Однак в більшість ігор приємніше грати, коли лабіринти неідеальні і в них є петлі. Вони повинні бути великими і складними для відкритих просторів, а не з вузькими звивистими коридорами. Це особливо справедливо для жанру rogue-like, у якому процедурні рівні є не так лабіринтами, а скоріше підземеллями.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Рейтинг мов програмування

Згідно звіту RedMonk за січень 2023 року, Python став другою за популярністю мовою програмування після Java Script. Раніше цю позицію упродовж тривалого часу впевнено утримував Java, проте на початку року ця мова програмування змістився на третій рядок рейтингу, який формується на базі інформації репозиторіїв GitHub. Якщо бути точними, пара Java Script і Java утримували топ-2 популярності мов програмування з моменту початку формування зазначеного рейтингу, тобто з 2012 року. (рис. 2. 1).

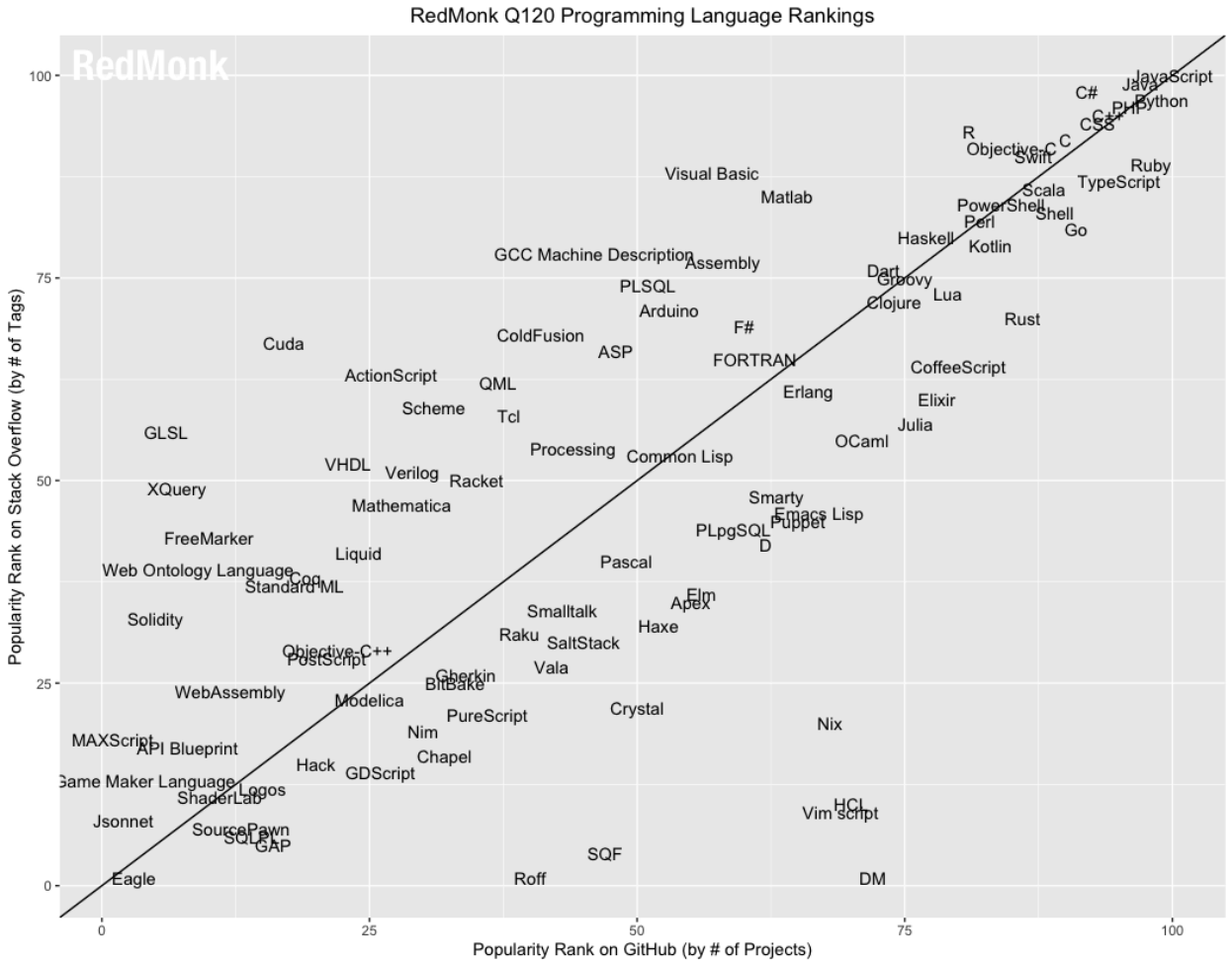


Рисунок 2. 1. Рейтинг мов програмування

Популярність JS та Java та їх топові позиції в рейтингу очікувані. Java Script – основна мова веб-розробки вже довгі роки, тоді як Java – стандарт у розробці під Android та у сфері корпоративних додатків.

У рейтингу не зовсім коректно враховано такі специфічні наукові мови, як Mathematica, тому що основна сфера їх застосування та спільноти знаходяться поза ресурсами, на базі яких складався рейтинг. Повний топ-20 найпопулярніших мов програмування на січень 2023 року за версією RedMonk має такий вигляд:

1. JavaScript
2. Python
3. Java
4. PHP
5. C#
6. C++
7. Ruby
8. CSS
9. TypeScript
10. C
11. Swift
12. Objective-C
13. Scala
14. R
15. Go
16. Shell
17. PowerShell
18. Perl
19. Kotlin
20. Haskell

Основною причиною зростання Python у рейтингу дослідники називають універсальність мови. Як і Java, Python застосовується у величезній кількості областей і здатний долати абсолютно різні завдання. Ця пластичність у плані застосування і робить Python настільки привабливим для багатьох розробників по всьому світу. Доповнює зазначений фактор ще й низький поріг входження в мову, тому армія «пітоністів», як у випадку з PHP або JS, постійно поповнюється новими та новими людьми (рис. 2. 2).

Рисунок 2. 2. Універсальна мова програмування Python

Зростання популярності Python виглядає ще більшим, якщо згадати про припинення підтримки Python 2 і перспективу вимушеної міграції безлічі проектів на Python 3 або переписування їх під інші мови програмування. Звичайно, підтримку Python 2 планувалося завершити ще в 2015 році, проте спільноті знадобилося набагато більше часу, щоб адаптуватися до змін.

Загалом весь рейтинг дає деякий матеріал для роздумів. На четвертому рядку рейтингу очікувано розмістився PHP, що комфортно почувається, а ось слідом йдуть дві такі серйозні мови як C# і C++. На 11 позиції Swift ледве

випереджає Objective-C, який він, нібито, мав практично повністю замінити. Молодіжні мови останніх років - Scala, Go і Kotlin - взагалі в другій десятці.

Особливо відчувається падіння Kotlin — другої мови програмування, що швидко зростає, в історії після Swift. На старті Kotlin відразу ж увірвався до топ-5 рейтингу RedMonk, проте це був просто одноразовий сплеск: мови від JetBrains не вдалося втриматися навіть у першій десятці, і з місяця на місяць його популярність лише падає. Цього місяця Kotlin додав один рядок і перемістився з 20 на 19 місце, проте лише час покаже, чи постійне це зростання, чи це просто «сезонні» коливання. Цілком ймовірно, він зафіксується на якійсь позначці у другій десятці, як це було зі Swift.

Весь рейтинг складався на базі даних GitHub Archive з перехресною перевіркою баз Stackoverflow. Ці два ресурси використовувалися для збору статистики, оскільки є найбільшими спільнотами розробників у всьому світі.

2.2. Галузі застосування Python

JetBrain спільно з Python Software Foundation опублікували результати великого дослідження, що допомагає зрозуміти, як розробники використовували Python у 2022 році. Дослідження будується з урахуванням опитування розробників. У 2022 році в ньому взяло участь понад 23 тис. осіб із майже 200 регіонів.

Результати дослідження:

- розробники на Python у 86% випадків використовують його разом з іншими мовами та технологіями. У п'ятірку популярних входять JavaScript, HTML/CSS, SQL, Bash/Shell та C/C++;

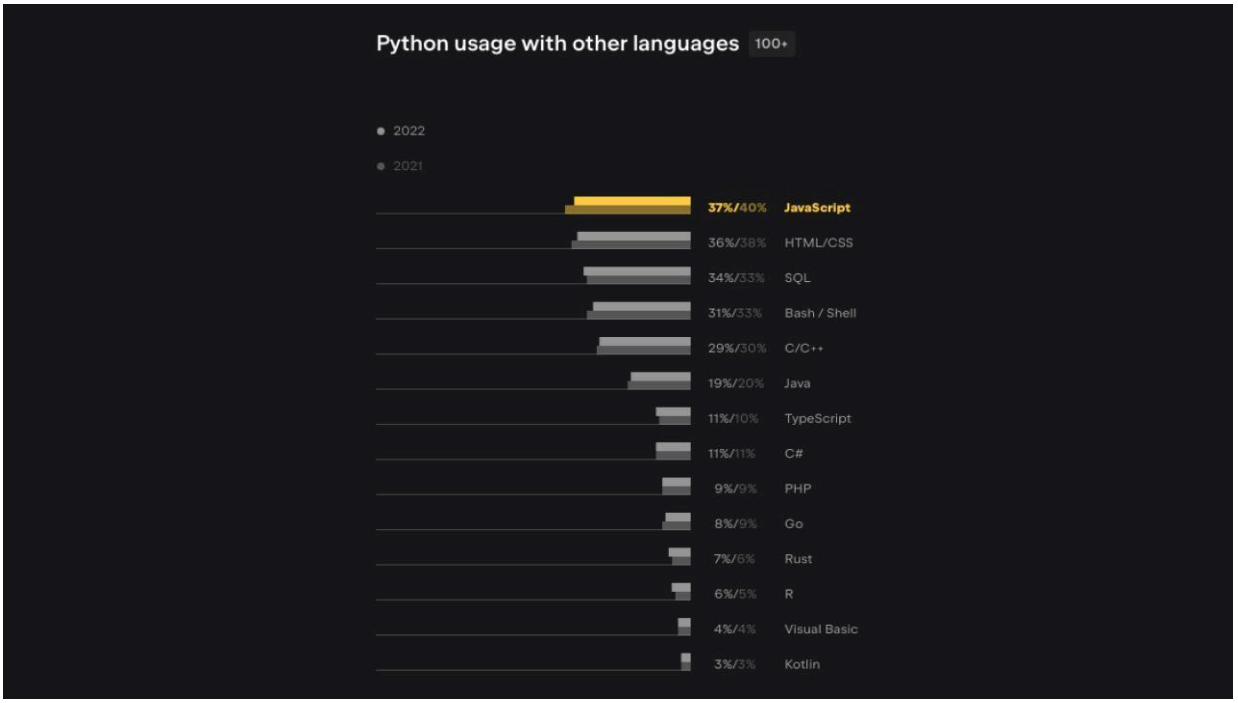


Рисунок 2. 3. Використання Python з іншими мовами програмування

- 51% опитаних використовує мову для роботи та власних проектів, ще 28% використовує лише для освіти та реалізації власних ідей та 21% — лише на роботі;
- в основному Python використовується для аналізу даних, веб-розробки, машинного навчання, DevOps та розробки веб-парсерів;

- більше 90% розробників перейшли на Python 3, а старою версією користується лише 7%; (рис. 2. 4)

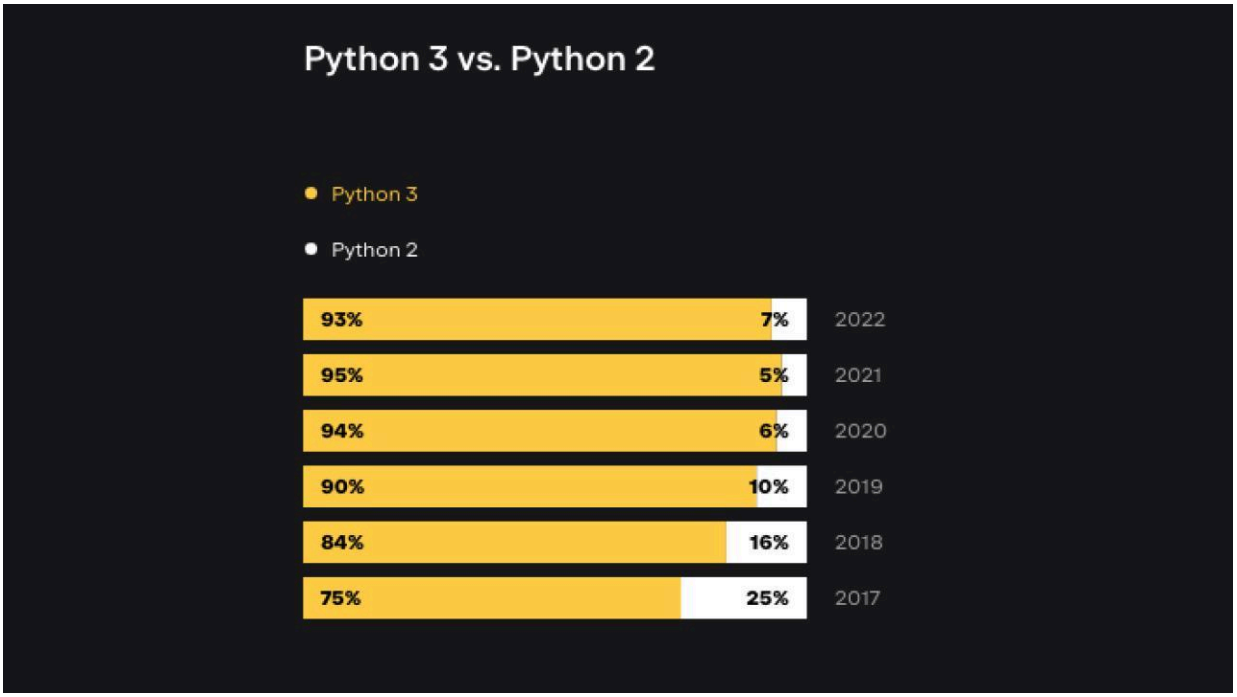


Рисунок 2. 4. Кількість користувачів старою і новою версіями Python

- найчастіше Python встановлюється та оновлюється з офіційного сайту, пакетних менеджерів в ОС (apt-get, yum, homebrew та інших), Anaconda, Docker та pynew;
- для ізоляції середовища Python використовують Virtualenv, Docker, Conda, Pipenv та Poetry. Важливо відзначити, що у 2022 році популярність Poetry значно зросла серед розробників;
- трійку популярних веб-фреймворків на Python представляють Flask, Django та FastAPI; (рис. 2. 5)

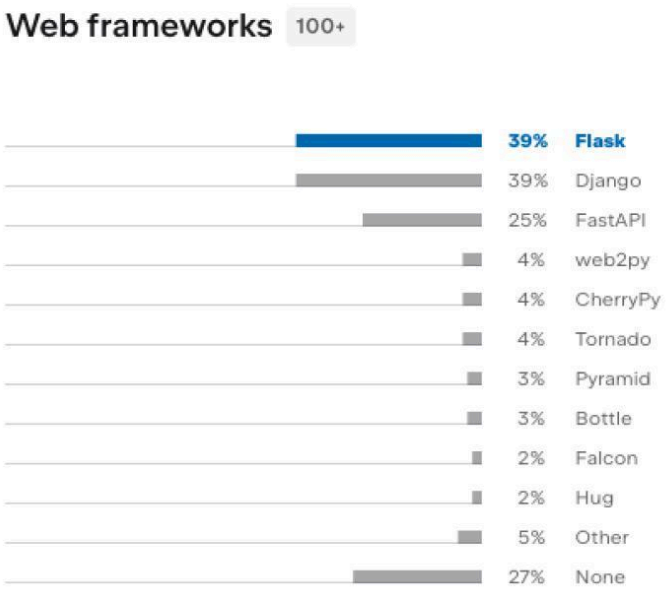


Рисунок 2. 5. Популярні веб-фреймворки на Python

- п'ятірку популярних бібліотек на Python представляють Requests, Pillow, Asyncio, Tkinter та PyQT; (рис. 2. 6)

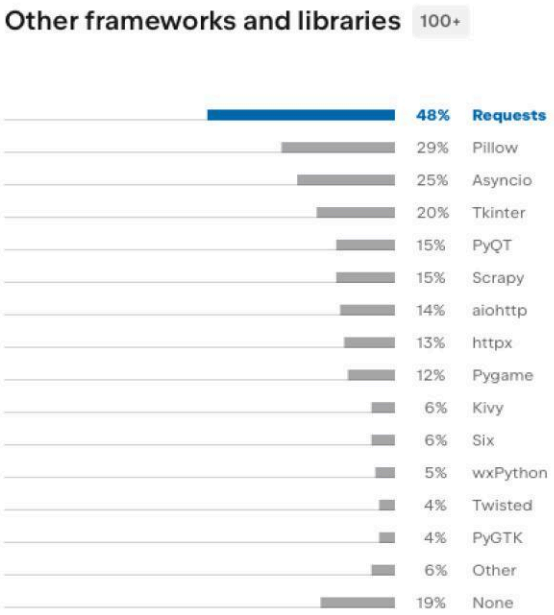


Рисунок 2. 7. Популярні бібліотеки на Python

- для тестування найчастіше використовують pytest, unittest та mock;

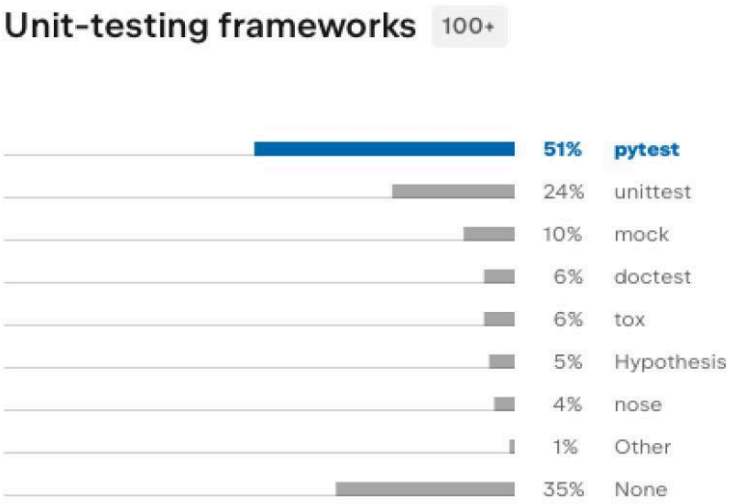


Рисунок 2. 8. Фреймворки для тестування

- як CI-систем розробники вибирають GitHub Actions, Gitlab CI та Jenkins;
- найпопулярніший редактор коду – VS Code, а PyCharm відстає на 8%;

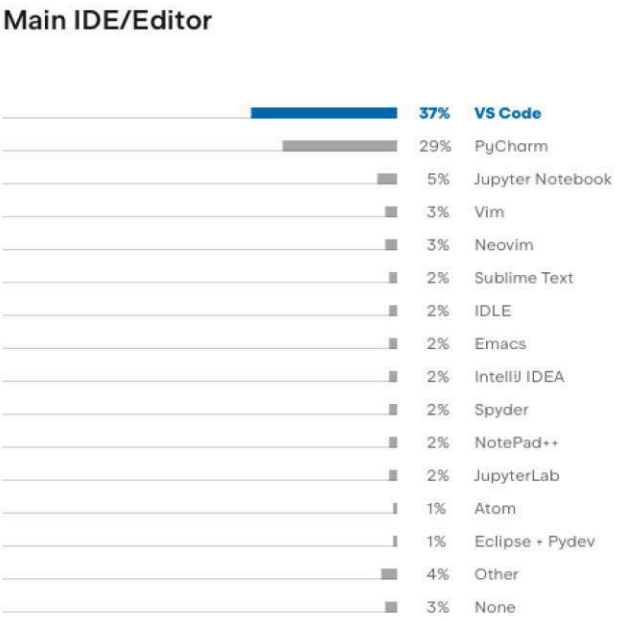


Рисунок 2. 9. Найпопулярніші редактори коду

- 59% опитаних постійно працюють із Python на роботі, 13% використовують мову для навчання, 7% — фрілансери.

Детальніше з результатими дослідження можна ознайомитись на офіційному сайті проекту.

За даними широко відомої у вузьких колах Tiobe Index мова Python швидше за все стане мовою 2024 року, вчетверте у своїй кар'єрі. Крім того, швидше за все він обжене Java і займе другий рядок у загальному рейтингу мов програмування за мовою C. Python як компілювана статично типізована мова програмування.

Основним драйвером зростання популярності мови Python стало її широке використання у завданнях машинного навчання. Python мова, що динамічно типізується і інтерпретується. Це все дуже повільно. Як його використовувати у наукових обчисленнях, які потребують максимальної продуктивності.

Зазвичай вважається, що Python це лише обгортка над обчислювальним ядром, написаним C++. А що ще можна очікувати від такої повільної мови? Але ядро то ядром, але місцями хочеться обробити дані тут і зараз на такому зручному Python.

За всю історію Python було вигадано велику кількість рішень, що дозволяють прискорити Python код: a) Fortran, C, C++ модулі; b) NumPy масиви; c) Cython розширення, та багато іншого. Це все працювало, звичайно, але все це було лише зовнішнім кодом по відношенню до Python. Місцями досить незграбним.

Для швидкого коду потрібні типи. А потім все це якось треба було комплювати в рамках інтерпретованої мови. Звучить якось не дуже реально. Але! Це все працює прямо зараз.

Дві дороги: python інструкції типів і LLVM jit компіляція зійшлися в релізі Numba 0.52.0. Дивимося на код, який каже сам за себе.

```
from typing import List
from numba.experimental import jitclass
from numba.typed import List as NumbaList

@jitclass
class Counter:
    value: int

    def __init__(self):
        self.value = 0

    def get(self) -> int:
        ret = self.value
        self.value += 1
        return ret

@jitclass
class ListLoopIterator:
    counter: Counter
    items: List[float]

    def __init__(self, items: List[float]):
        self.items = items
        self.counter = Counter()

    def get(self) -> float:
        idx = self.counter.get() % len(self.items)
        return self.items[idx]

items = NumbaList([3.14, 2.718, 0.123, -4.])
loop_itr = ListLoopIterator(items)
```

Кожен клас компілюється за допомогою LLVM у нативний код платформи з використанням анотацій типів.

Висновки до розділу

Python - це інтерпретована високорівнева мова програмування, яка була створена наприкінці 1980-х років Гвідо ван Россумом. Python швидко став однією з найпопулярніших мов програмування у світі завдяки своїй простоті та широкому спектру застосувань. Зараз Python використовується практично скрізь, від науки та фінансів, до веб-розробки і штучного інтелекту.

Тому не дивно, що багато хто хоче вивчити цю мову.

РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Математична модель Амарі

Протягом XX століття вчені будували математичні моделі одиночних нейронів (клітин нервової системи) та їх взаємодії між собою. У 1975 році С. Амарі представив світу свою безперервну модель кори головного мозку. У ній нервова система розглядалася як суцільне середовище, у кожній точці якого знаходиться «нейрон», що характеризується значенням потенціалу своєї мембрани, яка змінює свій потенціал, обмінюючись зарядами із сусідніми нейронами та зовнішніми подразниками. Модель Амарі відома тим, що пояснює багато феноменів людського зору і, зокрема, зорові галюцинації, що викликаються психотропними речовинами.

Модель Амарі, в її найпростішому вигляді, є задачею Коші для одного інтегрально-диференціального рівняння:

$$\begin{cases} \dfrac{\partial u}{\partial t}(x,t) = -u(x,t) + h + \displaystyle\int_{\Omega} w(x-y)H(u(y,t))\,dy + s(x,t), \;\; x \in \Omega, \;\; t > 0, \\ u(x,0) = \phi(x), \;\; x \in \Omega. \end{cases}$$

(1)

де *U (x, t)* - значення потенціалу мембрани нейрона в точці *x* в момент часу *t*,

h - потенціал спокою (деяка константа),

H - ступінчаста функція Хевісайда:

$$H(u) = \begin{cases} 1, & u > 0, \\ 0, & u \leq 0. \end{cases}$$

(2)

w - вагова функція,

s - зовнішній подразник,

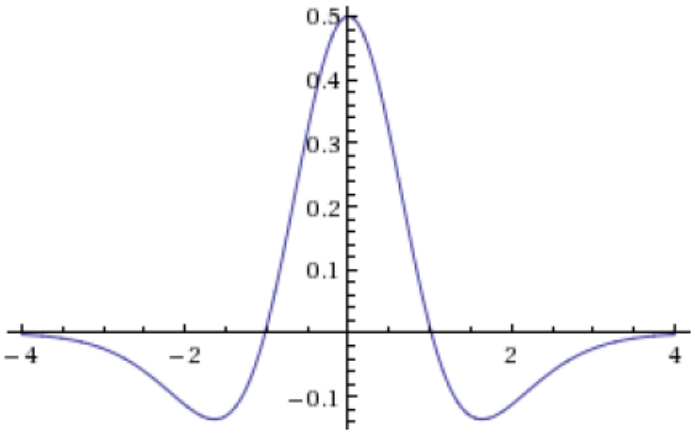
ϕ - розподіл потенціалу в початковий момент часу,

x - довільна точка області Ω, на якій визначено потенціал. Оскільки ми плануємо генерувати двовимірне зображення лабіринту, в якості Ω розглядатимемо всю площину.

Частина похідна u за часом у лівій частині означає миттєву зміну потенціалу u. Права частина визначає правило цієї зміни.

Перші дві складові правої частини означають, що за відсутності подразників значення потенціалу прагне до значення потенціалу спокою. Наступний доданок враховує вплив сусідніх нейронів. Функція Хевісайда грає роль активаційної функції нейрона: нейрон починає впливати на сусідів лише за умови, що його потенціал більший за нуль. Далі називатимемо такі нейрони активними, а безліч точок з позитивним потенціалом — областю активності. Ясно, що нейрони, які знаходяться у стані спокою, не повинні бути активними, тобто потенціал спокою не повинен бути позитивним. Активних сусідів можна умовно розділити на дві групи: збуджуючі та гальмівні. Збудливі нейрони збільшують потенціал сусідів, а гальмівні зменшують. При цьому збуджуючі створюють потужний сплеск активності в малій околиці, а гальмуючі поступово гасять активність в околиці великого радіусу. Саме цей факт відображено у виборі вагової функції у формі «мексиканського капелюха»:

$$w(r) = Ke^{-k\|r\|^2} - Me^{-m\|r\|^2},\ K > M,\ k > m.$$



(3)

Останній доданок правої частини рівняння враховує дію зовнішнього подразника. Наприклад, для зорової кори головного мозку природним подразником є сигнал, отриманий із сітківки ока. Будемо вважати, що подразник заданий невід'ємною стаціонарною (незалежною від часу) функцією.

Поставимо питання: чи можна підібрати параметри моделі ϕ, h, K, k, M, m, s так, щоб її стаціонарне рішення (при $t \to \infty$) було зображенням деякого лабіринту?

Для аналізу рішень моделі Амарі нам достатньо обмежитися розглядом одновимірного випадку. Для простоти будемо вважати, що s постійна на всій прямій. Насамперед нас цікавлять так звані бамп-рішення. Вони чудові тим, що позитивні лише на деякому кінцевому інтервалі $(-A(t), A(t))$ з рухомими межами. Рівняння Амарі для них записується так:

$$\frac{\partial u}{\partial t}(x,t) = -u(x,t) + h + \int_{-A(t)}^{A(t)} w(x-y)\,dy + s, \quad -\infty < x < \infty, \ t > 0.$$

(4)

Щоб зрозуміти, як поводиться його рішення, введемо функцію

$$W(a) = \int_{-a}^{a} w(x)\,dx.$$

(5)

Тепер те саме рівняння можна переписати так:

$$\frac{\partial u}{\partial t}(x,t) = -u(x,t) + h + \frac{1}{2}\left[W(x + A(t)) - W(x - A(t))\right] + s.$$

(6)

Нам відомо, що бамп-рішення перетворюється в нуль на межах інтервалу активності (тому вони і називаються межами). Запишемо цю умову на правій межі:

$$u(A(t), t) = 0, \ t > 0.$$

(7)

А тепер продиференціюємо останню тотожність щодо змінної *t*:

$$\frac{du}{dt}(A(t), t) = \frac{\partial u}{\partial x}(A(t), t)A'(t) + \frac{\partial u}{\partial t}(A(t), t) = 0. \quad (8)$$

Звідси:

$$\frac{\partial u}{\partial t}(A(t), t) = -\frac{\partial u}{\partial x}(A(t), t)A'(t).$$

(9)

Підставляючи останній вираз у рівняння для бамп-рішення при *x=A(t),* отримаємо:

$$-\frac{\partial u}{\partial x}(A(t), t)A'(t) = h + \frac{1}{2}W(2A(t)) + s, \ t > 0.$$

(10)

Тепер зауважимо, що частина похідна по лівій частині завжди негативна, оскільки ліворуч від правого кордону рішення більше нуля, а праворуч від неї менше. Тому

$$\operatorname{sgn} A'(t) = \operatorname{sgn}\left\{h + \frac{1}{2}W(2A(t)) + s\right\}.$$

(11)

Таким чином, напрямок зсуву кордону залежить лише від значення виразу у правій частині. Якщо воно більше за нуль, то область активності розширюється, якщо менше — звужується. За рівності нулю досягається рівновага. Поглянемо на можливі графіки функції *W (a)*.

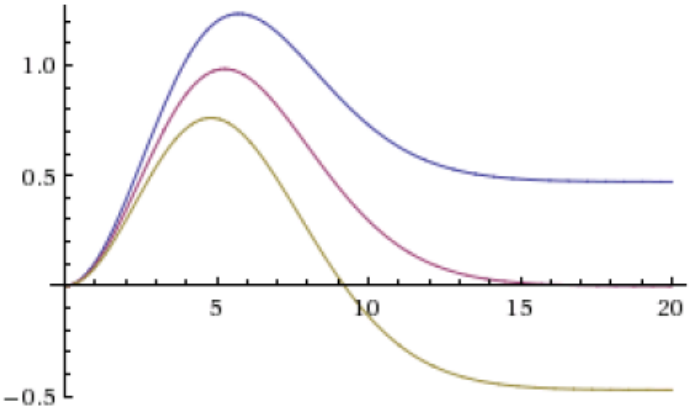


Рисунок 3. 1. Графіки функції *W (a)*

Очевидно, можливі два випадки:

1. Граничне значення $h + \frac{1}{2}W(\infty) + s$ невід'ємне. Тоді область активності бамп-рішення необмежено розширюватиметься.

2. Граничне значення $h + \frac{1}{2}W(\infty) + s$ є негативним. Тоді область активності буде обмежена. Більш того, у цьому випадку можна показати, що зв'язкові компоненти області активності розв'язування рівняння Амарі ніколи не зливаються.

На жаль, у двовимірному випадку отримати явний вираз для функції важко, тому ми просто оцінимо її:

$$W(a) \approx \int\limits_{\|x\|<a} w(x)\,dx = \pi\left(\frac{K}{k}\left(1 - e^{-ka^2}\right) - \frac{M}{m}\left(1 - e^{-ma^2}\right)\right).$$

(12)

Звідси:

$$W(\infty) \approx \pi\left(\frac{K}{k} - \frac{M}{m}\right).$$

(13)

Висновки до розділу

Протягом XX століття вчені будували математичні моделі одиночних нейронів (клітин нервової системи) та їх взаємодії між собою. У 1975 році С. Амарі представив світу свою безперервну модель кори головного мозку. У ній нервова система розглядалася як суцільне середовище, у кожній точці якого знаходиться «нейрон», що характеризується значенням потенціалу своєї мембрани, яка змінює свій потенціал, обмінюючись зарядами із сусідніми нейронами та зовнішніми подразниками. Модель Амарі відома тим, що пояснює багато феноменів людського зору і, зокрема, зорові галюцинації, що викликаються психотропними речовинами.

РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

4.1. Генерація лабіринту

Зібравши багаж необхідних знань, ми можемо розпочати, власне, алгоритм генерації лабіринту.

Насамперед, визначимося із самим поняттям «лабіринт». Під лабіринтом будемо мати на увазі бінарну функцію $L:\Omega\rightarrow\{0,1\}$ таку, що область $\{x\in\Omega: L(x)=0\}$ зв'язана. Значення 0 відповідає вільному осередку, а значення 1 - непрохідній стіні. Умова зв'язності говорить про те, що з будь-якого вільного осередку можна дістатися до будь-якого іншого, не руйнуючи при цьому стіни. Функцію L шукатимемо у вигляді:

$$L(x) = \begin{cases} 0 & u(x,\infty) \leq 0 \\ 1 & u(x,\infty) > 0 \end{cases}$$

(14)

де u- рішення моделі Амарі.

Залишилося лише визначитися із параметрами моделі. Почнемо з того, що зафіксуємо довільне негативне значення *h*. Природно покласти $\phi \equiv h\, \phi \equiv h$. Тепер задамо функцію *s*. Нехай її значення у кожній точці визначається випадковою величиною, рівномірно розподіленою на відрізку *[0, -h]*. У такому разі збурюючий фактор не буде проявляти активності. Зафіксуємо довільне, позитивне значення *K*. Цей параметр впливає лише на абсолютну величину потенціалу, тому не становить інтересу. Зафіксуємо довільні позитивні *k, m:k>m*. Вони визначають характерну товщину стінок лабіринту. Параметр *M* спробуємо визначити експериментально, а потім порівняти з теоретичною оцінкою, отриманою у попередньому розділі.

Стаціонарне рішення шукатимемо методом послідовних наближень:

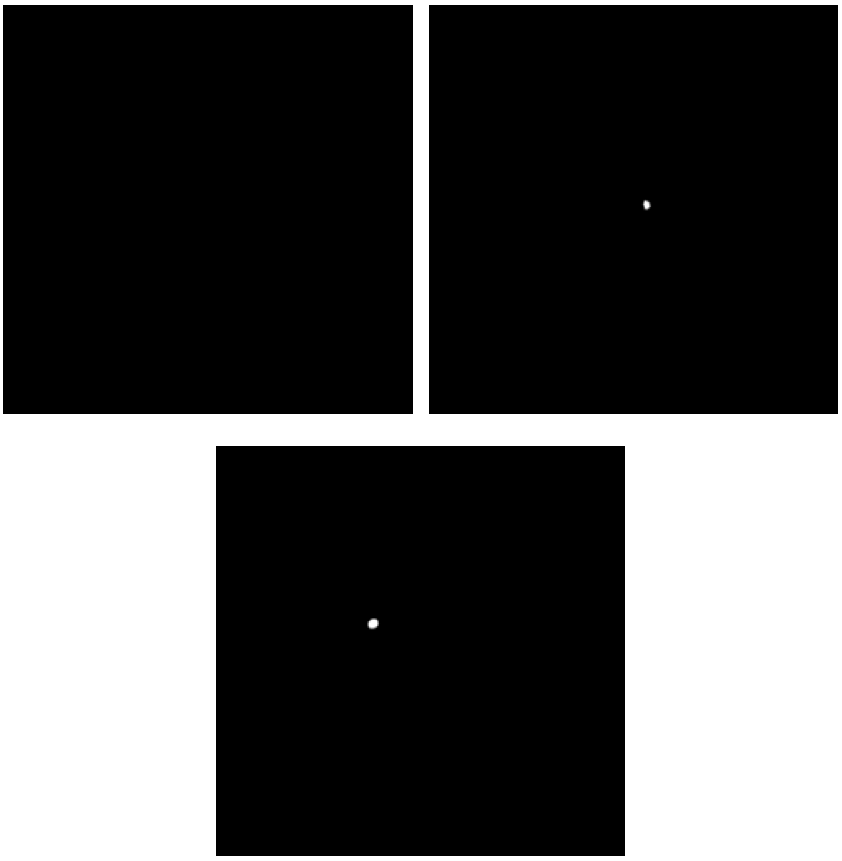
$$u = \lim_{k\to\infty} u_k,$$

$$u_k(x) = h + \int_{\Omega} w(x-y)H\big(u_{k-1}(y)\big)dy + s(x), \;\; x \in \Omega,$$

$$u_0 = \phi. \tag{15}$$

Хотілося б відзначити лише те, що згортка з ваговою функцією обчислюється через фільтр Гауса, причому зображення продовжуються періодично на всю площину (параметр wrap). Демонстрація інтерактивна в тому сенсі, що дозволяє примусово встановити позитивний потенціал у будь-якій точці кліку.

Поведінка рішення, як і очікувалося, залежить від вибору параметра *M* (рис. 4.1).



M=0.05 *M=0.065* *M=0.07*

Рисунок 4. 1. Вибір параметра M

Тепер отримаємо теоретичну оцінку оптимального значення параметра *M*. Воно задовольняє умову:

$$h + \frac{1}{2}W(\infty) + \max_{x\in\Omega} s(x) = 0 \leftrightarrow W(\infty) = 0. \tag{16}$$

Тому його можна оцінити так:

$$W(\infty) \approx \pi\left(\frac{K}{k} - \frac{M}{m}\right) \approx 0 \rightarrow M \approx m\frac{K}{k} = 0.0625$$

(17)

Проте реальне значення *M* трохи вище за теоретичну оцінку. У цьому легко переконатися, поклавши *h=0*. Нарешті, можна змінювати ступінь «розрідженості» лабіринту, змінюючи значення параметра *h* (рис. 4. 2).

Рисунок 4. 2. Залежність розрідженості лабіринту від параметра *h*

4.2. Реалізація алгоритму на мові Python

А ось і довгоочікувана інтерактивна демонстрація на Python:

```
import math
import numpy
import pygame
from scipy.misc import imsave
from scipy.ndimage.filters import gaussian_filter


class AmariModel(object):

    def __init__(self, size):
        self.h = -0.1
        self.k = 0.05
        self.K = 0.125
        self.m = 0.025
        self.M = 0.065

        self.stimulus = -self.h * numpy.random.random(size)
        self.activity = numpy.zeros(size) + self.h
        self.excitement = numpy.zeros(size)
        self.inhibition = numpy.zeros(size)

    def stimulate(self):
        self.activity[:, :] = self.activity > 0

        sigma = 1 / math.sqrt(2 * self.k)
        gaussian_filter(self.activity, sigma, 0, self.excitement, "wrap")
        self.excitement *= self.K * math.pi / self.k

        sigma = 1 / math.sqrt(2 * self.m)
        gaussian_filter(self.activity, sigma, 0, self.inhibition, "wrap")
        self.inhibition *= self.M * math.pi / self.m

        self.activity[:, :] = self.h
        self.activity[:, :] += self.excitement
        self.activity[:, :] -= self.inhibition
        self.activity[:, :] += self.stimulus


class AmariMazeGenerator(object):

    def __init__(self, size):
        self.model = AmariModel(size)

        pygame.init()
        self.display = pygame.display.set_mode(size, 0)
        pygame.display.set_caption("Amari Maze Generator")

    def run(self):
        pixels = pygame.surfarray.pixels3d(self.display)

        index = 0
        running = True
        while running:
            self.model.stimulate()
```

```
            pixels[:, :, :] = (255 * (self.model.activity > 0))[:, :, None]
            pygame.display.flip()

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False
                elif event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_ESCAPE:
                        running = False
                    elif event.key == pygame.K_s:
                        imsave("{0:04d}.png".format(index), pixels[:, :, 0])
                        index = index + 1
                elif event.type == pygame.MOUSEBUTTONDOWN:
                    position = pygame.mouse.get_pos()
                    self.model.activity[position] = 1

        pygame.quit()


def main():
    generator = AmariMazeGenerator((512, 512))
    generator.run()


if __name__ == "__main__":
    main()
```

Я вважаю, що коментарі зайві. Хотілося б відзначити лише те, що згортка з ваговою функцією обчислюється через фільтр Гауса, причому зображення продовжуються періодично на всю площину (параметр wrap). Демонстрація інтерактивна в тому сенсі, що дозволяє примусово встановити позитивний потенціал у будь-якій точці кліку.

Поведінка рішення, як і очікувалося, залежить від вибору параметра M:

4.3. Програмна реалізація алгоритму генерації лабіринтів

Лабіринт може зберігатися у файлі у вигляді певної кількості рядків та стовпців, а також двох матриць, які містять положення вертикальних та горизонтальних стін відповідно. У першій матриці відображається наявність стіни праворуч від кожної комірки, а в другій – знизу (рис. 4. 3).

4 4
0 0 0 1
1 0 1 1
0 1 0 1
0 0 0 1

1 0 1 0
0 0 1 0
1 1 0 1
1 1 1 1

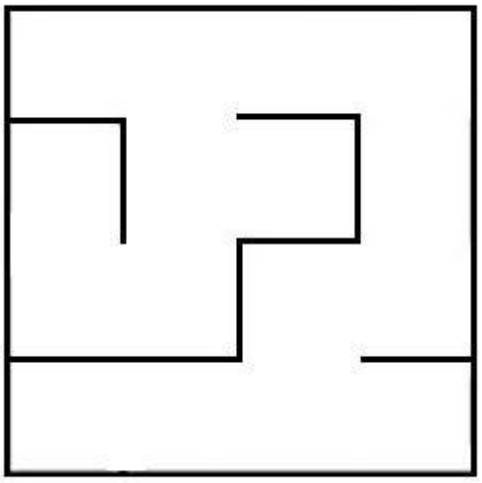


Рисунок 4. 3. Приклад лабіринту із файлу

Створимо перший рядок. Жодна комірка не буде частиною будь якої множини.

Рисунок 4. 4. Створення першого рядка

```
/* Заповнюємо вектор порожнім значенням */
void Maze::fillEmptyValue() {
    for (int i = 0; i < cols_; i++) {
        sideLine_.push_back(kEmpty);
    }
}
```

Присвоїмо коміркам, що не входять до множини, свою унікальну множину.

Рисунок 4. 5. Присвоєння коміркам унікальної множини

```
/* Присвоюємо коміркам свою унікальну множину */
void Maze::assignUniqueSet() {
    for (int i = 0; i < cols_; i++) {
      /* Перевіряємо на пусту комірку */
        if (sideLine_[i] == kEmpty) {
            /* Присвоюємо комірці унікальну множину */
            sideLine_[i] = counter_;
            counter_++;
        }
    }
}
```

Створимо праві границі, рухаючись зліва направо:
(випадково вирішуючи додавати границю чи ні)

1. Якщо поточна комірка і комірка праворуч належать одній множині, то створимо границю між ними (для запобігання зацикленням);

2. Якщо ми вирішимо не додавати границю, то об'єднаємо дві множини в яких знаходиться поточна комірка і комірка праворуч.

Рисунок 4. 6. Права стінка біля комірки під номером 2

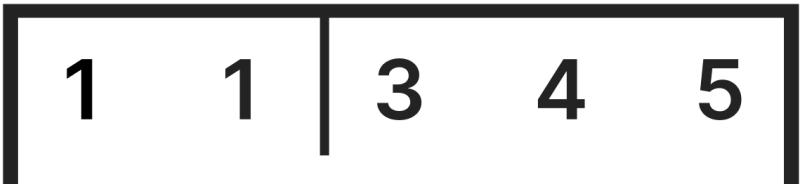
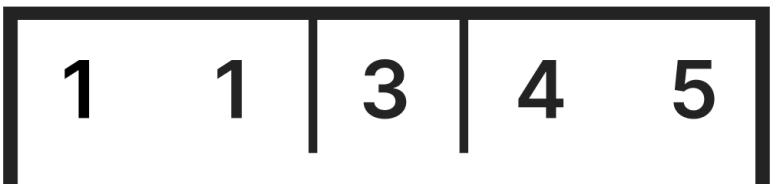
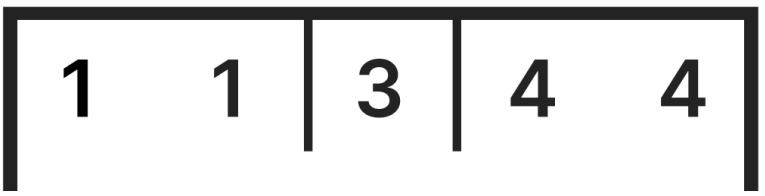


Рисунок 4. 7. Об'єднання двох множин в яких знаходиться поточна комірка і комірка праворуч






```
/* Додавання правої вертикальної стінки */
void Maze::addingVerticalWalls(int row) {
    for (int i = 0; i < cols_ - 1; i++) {
      /* Ставимо стінку або ні */
        bool choise = randomBool();
      /* Перевірка умови для попередження зациклювання */
        if (choise == true || sideLine_[i] == sideLine_[i + 1]) {
            v_walls_(row, i) = true;
        } else {
            /* Об єднання комірок в одну множину */
            mergeSet(i, sideLine_[i]);
        }
    }
      /* Додавання правої стінки в останній комірці */
    v_walls_(row, cols_ - 1) = true;
}

/* Об єднання комірок в одну множину */
void Maze::mergeSet(int index, int element) {
    int mutableSet = sideLine_[index + 1];
    for (int j = 0; j < cols_; j++) {
      /* Перевірка комірок на одну множину */
        if (sideLine_[j] == mutableSet) {
            /* Об єднання комірок в множину */
            sideLine_[j] = element;
        }
    }
}
```

Створимо межі знизу, рухаючись зліва направо:

- Випадково вирішуючи додавати межі чи ні. Переконайтеся, що кожна множина має хоча б одну комірку без нижньої межі (для запобігання ізолюванню областей):

1. Якщо комірка у своїй множині одна, то не створюємо межі знизу;

2. Якщо комірка одна у своїй множині без нижньої межі, то не створюємо нижньої границі.

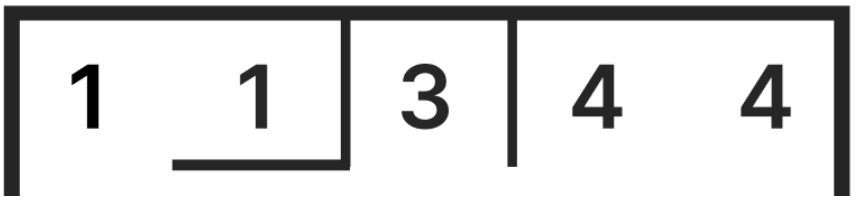


Рисунок 4. 8. Створення нижньої межі в комірці під номером 2, так як 1 множина має хоча б одну комірку без нижньої межі

У комірці під номером 3 не можемо додати нижню стінку, так як комірка у множині 3 одна.

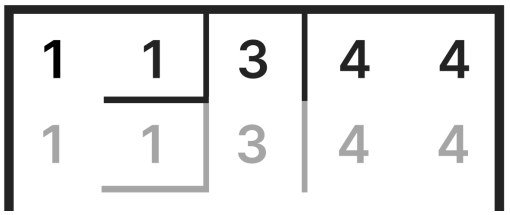
```
void Maze::addingHorizontalWalls(int row) {
    for (int i = 0; i < cols_; i++) {
        bool choise = randomBool();
             if (calculateUniqueSet(sideLine_[i]) != 1 && choise == true) {
                    h_walls_(row, i) = true;
        }
    }
}

int Maze::calculateUniqueSet(int element) {
    int countUniqSet = 0;
    for (int i = 0; i < cols_; i++) {
        if (sideLine_[i] == element) {
            countUniqSet++;
        }
    }
    return countUniqSet;
}
void Maze::checkedHorizontalWalls(int row) {
    for (int i = 0; i < cols_; i++) {
        if (calculateHorizontalWalls(sideLine_[i], row) == 0) {
            h_walls_(row, i) = false;
        }
```

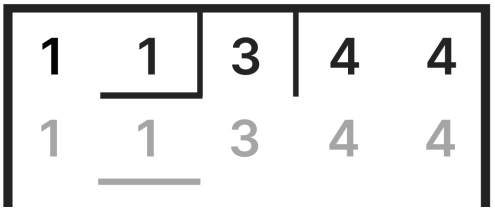
```
    }
}
int Maze::calculateHorizontalWalls(int element, int row) {
    int countHorizontalWalls = 0;
    for (int i = 0; i < cols_; i++) {
        if (sideLine_[i] == element && h_walls_(row, i) == false) {
            countHorizontalWalls++;
        }
    }
    return countHorizontalWalls;
}
```

Якщо нам потрібно додати ще один рядок, то необхідно зробити наступні кроки:

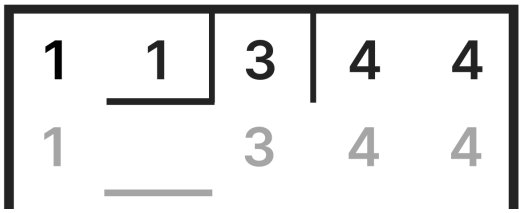
1. Вивести поточний рядок;
2. Видалити усі праві межі;
3. Видалити комірки з нижньою межею з їхньої множини;
4. Видалити усі нижні межі;
5. Продовжити з кроку 2;



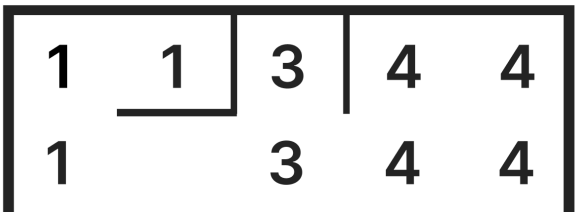
Копіювання поточного рядка



Видалення правої стінки



Видалення комірки з нижньою межею



Видалення усіх нижніх меж

```
void Maze::preparatingNewLine(int row) {
    for (int i = 0; i < cols_; i++) {
        if (h_walls_(row, i) == true) {
                        sideLine_[i] = kEmpty;
        }
    }
}
```

Продовжуємо алгоритм до останнього рядка.

Якщо ви хочете закінчити лабіринт, то:

Додайте нижню межу до кожної комірки:

Рухаючись зліва направо:

1. Якщо поточна комірка і комірка праворуч члени різних множин, то:

1.1 Видаліть правий кордон

1.2 Об'єднайте множину поточної комірки та комірки праворуч

1.3 Виведіть завершальний рядок.

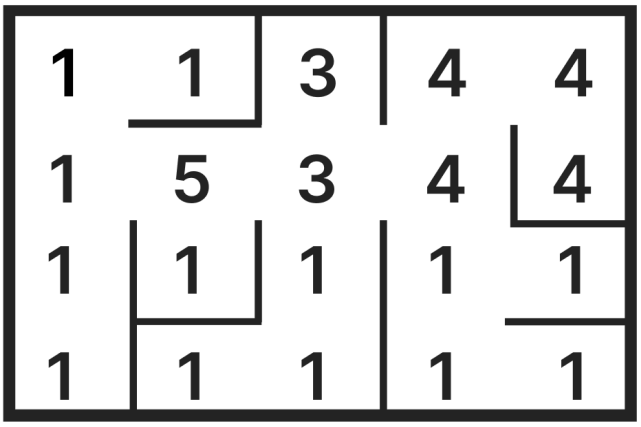


Рисунок 4. 9. Завершення лабіринту

```
void Maze::addingEndLine() {
    assignUniqueSet();
    addingVerticalWalls(rows_ - 1);
    checkedEndLine();
}
void Maze::checkedEndLine() {
    for (int i = 0; i < cols_ - 1; i++) {
            if (sideLine_[i] != sideLine_[i + 1]) {
                    v_walls_(rows_ - 1, i) = false;
```

```
            mergeSet(i, sideLine_[i]);
        }
            h_walls_(rows_ - 1, i) = true;
    }
        h_walls_(rows_ - 1, cols_ - 1) = true;
}
```

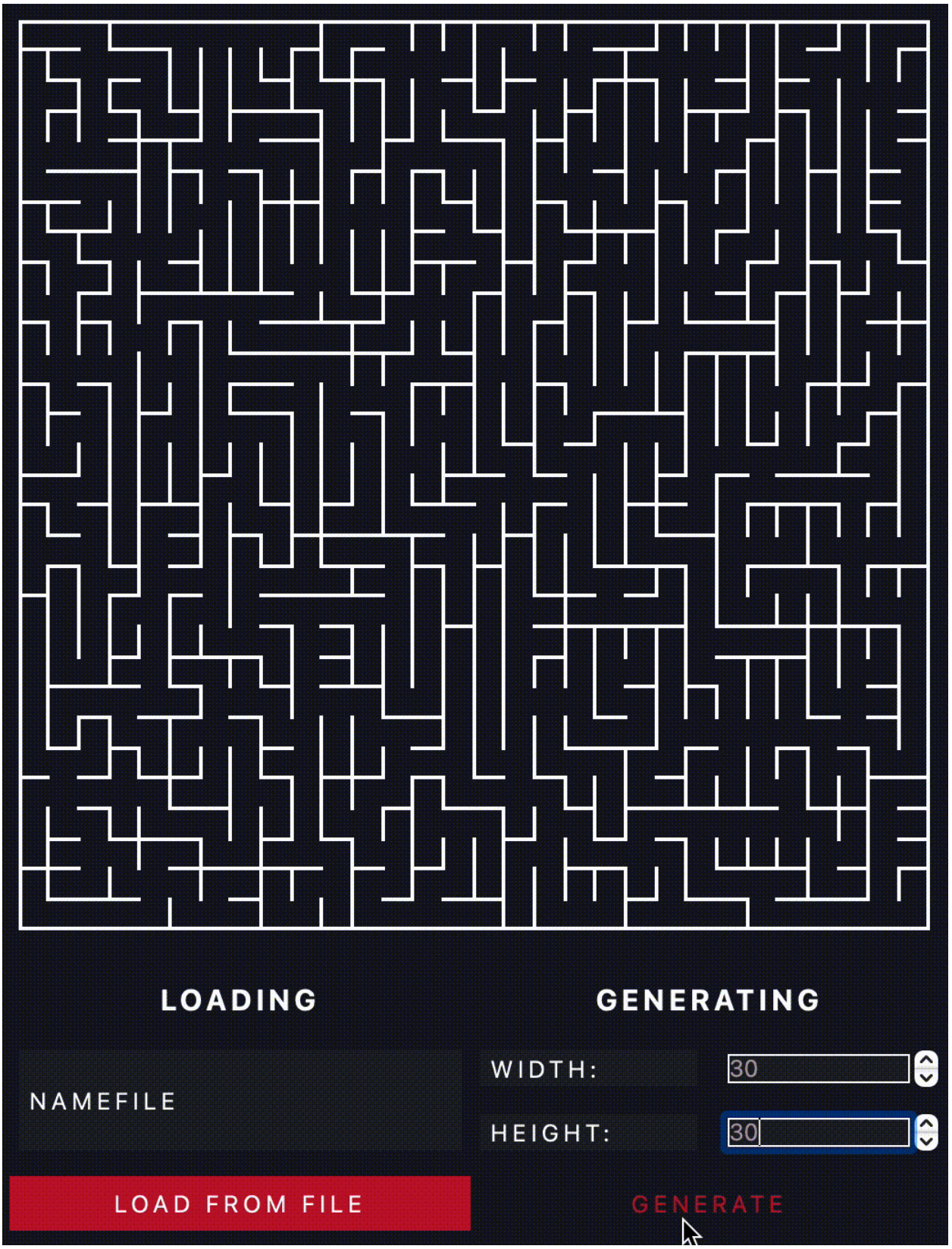


Рисунок 4. 10. Інтерфейс програмного застосунку для генерації лабіринтів

Основні функції застосунку:

- Область малювання, де можна вибрати початкову та кінцеву точки шляху.
- Завантаження файлу з txt-файлу (показати лабіринт).
- Налаштування висоти та ширини створеного лабіринту.
- Пошук найкоротшого шляху.

По-перше, коли ви запускаєте програму, ви можете вибрати завантажити лабіринт із файлу або «ГЕНЕРУВАТИ». Ви можете змінювати висоту і ширину лабіринту. Налаштування висоти і ширини вікон. Виберіть початкову точку лівою кнопкою миші, а кінцеву — правою кнопкою миші.

Висновки до розділу

Якщо запускати демо версію з параметрами, що вказані в роботі (M = 0.065), то білі «звивини» (стіни) ніколи не будуть зливатися. Оскільки спочатку чорна область була зв'язана (з будь-якої чорної точки можна було дійти до будь-якої іншої чорної точки, пересуваючись лише чорними точками), то і в будь-який інший момент часу вона залишиться зв'язковою. Інакше кажучи, проходження лабіринту «by design» можна домогтися правильним вибором параметрів моделі (точніше, одного параметра M).

У представленому варіанті зв'язаним буде лише нескінченне періодичне продовження лабіринту на всю площину. Це пов'язано з використанням режиму wrap у фільтрі Гауса. Щоб отримати кінцевий зв'язаний лабіринт, досить поміняти «wrap» на «reflect».

Дані лабіринти можна використовувати в гейм-дизайні для створення цікавих рівнів.

РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ

5.1. Опис ідеї проекту

Виявилося, що тема генерації лабіринтів не сильно розкрита в українському та англомовному науковому співтоваристві. В інтернеті наявні декілька статтей про алгоритми для створення лабіринтів. Деякі статті є перекладом англомовної статті з описом алгоритмів по кроках. У своєму проекті реалізації я спирався на власний унікальний алгоритм на основі моделі Амарі. У процесі реалізації ідеї проекту ми зіткнулися з деякими труднощами і особливими випадками.

Ідеальний лабіринт - це лабіринт у якому немає циклів (між двома комірками є лише один шлях) та ізольованих частин (комірки або групи комірок, які не пов'язані з іншими частинами лабіринту).

Ідея даного проекту полягала у реалізації одного незвичайного підходу до генерації лабіринтів. Він заснований на моделі Амарі нейронної активності кори головного мозку, що є безперервним аналогом нейронних мереж. За певних умов вона дозволяє створювати красиві лабіринти дуже складної форми, подібні до того, що наведено на рисунку 5.1.

Рисунок 5. 1. Ідеальний лабіринт заснований на моделі Амарі

Як таке завдання переросло в проект із генератором лабіринту — сказати складно. Заздалегідь зазначу, що протягом усієї розробки не розглядався досить сучасний і ефективний варіант в особі нейромереж, і це

було зроблено навмисно, щоб отримати хороші результати із цілком класичних методів та алгоритмів.

Алгоритм представлений у форматі інтерактивних форм на яких можна змінювати параметри моделі Амарі та характеристики лабіринту. Даний алгоритм може бути використаний у сфері ігрової індустрії для створення перешкод і генерації рівнів у відеоіграх.

5.2. Порівняльний аналіз алгоритмів створення ідеальних лабіринтів

Існує безліч способів створення ідеальних лабіринтів, і кожен із них має власні характеристики. Нижче в таблиці 5. 1 наведено список конкретних алгоритмів. У всіх них існує можливість створення лабіринту вирізанням проходів, проте якщо не вказано інше, кожен також можна реалізувати додаванням стін. У таблиці 5.1 коротко представлені характеристики описаних вище алгоритмів створення ідеальних лабіринтів. Для порівняння додано власний алгоритм генерації лабіринтів на основі моделі Амарі (теоретично створені лабіринти за допомогою даного алгоритму є ідеальними).

Таблиця 5. 1. Порівняльна характеристика алгоритмів

Алгоритм	**% глухих кутів**	**Тип**	**Пріоритет**	**Зміщеність відсутня?**	**Однорідний?**	**Пам'ять**	**Час**	**% рішення**
Алгоритм на основі моделі Амарі	0	Множина	Стіни, проходи	Так	Ніколи	N^2	37	99.0
Recursive Backtracker	10	Дерево	Проходи	Так	Ніколи	N^2	27	19.0
Hunt and Kill	11	Дерево	Проходи	Так	Ніколи	0	100	9.5
Рекурсивний поділ	23	Дерево	Стіни	Так	Ніколи	N*	10	7.2
Двійкове дерево	25	Множина	Обидва	Ні	Ніколи	0*	10	2.0
Алгоритм Еллера	28	Множина	Обидва	Ні	Ні	N*	20	4.2

Алгоритм Вілсона	29	Дерево	Обидва	Так	Так	N^2	48	4.5

Пояснення стовпців таблиці:

- **Глухий кут** - це приблизний відсоток комірок, що є глухими кутами в лабіринті, створеному за допомогою даного алгоритму, у разі ортогонального 2D-лабіринту.
- **Тип** - існує два типи алгоритмів створення ідеальних лабіринтів: алгоритм на основі дерева вирощує лабіринт подібно до дерева, завжди додаючи до того, що вже є і на кожному етапі маючи правильний ідеальний лабіринт. Алгоритм на основі множин виконує побудови там, де йому хочеться, відстежуючи частини лабіринту, з'єднані один з одним, щоб з'єднати все та створити правильний лабіринт на момент завершення роботи.
- **Пріоритет** - більшість алгоритмів можна реалізувати за допомогою вирізання проходів або додаванням стін. Дуже небагато можна реалізувати лише як один чи інший підхід.
- **Відсутність зміщеності** - чи однаково сприймає алгоритм усі напрями та сторони лабіринту так, що подальший аналіз лабіринту не може виявити жодної зміщеності проходів.
- **Однорідність** - чи генерує алгоритм всі можливі лабіринти з рівною ймовірністю.
- **Пам'ять** - обсяг додаткової пам'яті чи стека, необхідний для реалізації алгоритму.
- **Час** - цей параметр дає уявлення про те, скільки часу потрібно для створення лабіринту з допомогою даного алгоритму, що менше число, тим він працює швидше.
- **Рішення** - це відсоток комірок лабіринту, якими проходить його розв’язання для типового лабіринту, створюваного алгоритмом.

Висновки до розділу

Моделі нейронних полів описують грубозернисту активність популяцій взаємодіючих нейронів. Через ламінарну структуру справжньої кортикальної тканини їх часто вивчають у двох просторових вимірах, де вони, як відомо, створюють багаті моделі просторово-часової активності. Такі моделі інтерпретувалися в різних контекстах, починаючи від розуміння зорових галюцинацій і закінчуючи генерацією електроенцефалографічних сигналів. Типові візерунки включають локалізовані рішення у вигляді мандрівних плям, а також складні лабіринтові структури. Ці закономірності природно визначаються межею між низьким і високим станами нейронної активності.

ВИСНОВКИ

У результаті виконання дипломного проекту було проведено ознайомлення з предметною областю, пошук аналогів алгоритмів генерації лабіринтів, які використовують в сфері ігрової індустрії для створення перешкод у відеоіграх.

Результатом проведеної роботи є розроблений унікальний алгоритм, який заснований на моделі Амарі (нейронної активності кори головного мозку), що являється безперервним аналогом нейронних мереж. За певних умов дана модель дозволяє створювати гарні лабіринти дуже складної форми.

Алгоритм представлений у форматі інтерактивних форм на яких можна змінювати параметри моделі Амарі та характеристики лабіринту.

Для написання програмного коду були використані високорівневі мови програмування Python та C++.

В кінцевому рахунку, була досягнута мета проекту (розроблення та тестування алгоритму генерації лабіринтів на основі моделі Амарі).

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Konstantin Doubrovinski, *Dynamics, Stability and Bifurcation Phenomena in the Nonlocal Model of Cortical Activity*, 2005.
2. Dequan Jin, Dong Liang, Jigen Peng, *Existence and Properties of Stationary Solution of Dynamical Neural Field*, 2011.
3. Stephen Coombes, Helmut Schmidt, Ingo Bojak, *Interface Dynamics in Planar Neural Field Models*, 2012.
4. W. Erlhagen, E. Bicho, The dynamic neural field approach to cognitive robotics, J. Neural Eng. 3 (3) (2006) 36–54.
5. C. Faubel, G. Schöner, Learning to recognize objects on the fly: A neurally based dynamic field approach, Neural Netw. 21 (4) (2008) 562–576.
6. J.S. Johnson, J.P. Spencera, G. Schöner, Moving to higher ground: The dynamic field theory and the dynamics of visual cognition, New Ideas Psychol. 26 (2) (2008) 227–251.
7. J.S. Johnson, J.P. Spencera, G. Schöner, A layered neural architecture for the consolidation, maintenance, and updating of representations in visual working memory, Brain Res. 1299 (3) (2009) 17–32.
8. V.R. Simmering, A.R. Schuttea, J.P. Spencer, Generalizing the dynamic field theory of spatial cognition across real and developmental time scales, Brain Res. 1202 (2) (2008) 68–86.
9. S. Kubota, K. Hamaguchi, K. Aihara, Local excitation solutions in one-dimensional neural fields by external input, Neural Comput. Appl. 18 (6) (2009) 591–602.
10. R. Potthast, P.B. Graben, Existence and properties of solutions for neural field equations, Math. Methods Appl. Sci. 33 (8) (2010) 935–949.
11. Coombes S: **Large-scale neural dynamics: simple and complex.** *NeuroImage* 2010, **52:** 731–739. 10.1016/j.neuroimage.2010.01.045

12. Faye G, Chossat P, Faugeras O: **Analysis of a hyperbolic geometric model for visual texture perception.** *J Math Neurosci* 2011.
13. Lu Y, Amari S: **Traveling bumps and their collisions in a two-dimensional neural field.** *Neural Comput* 2011
14. Bressloff PC, Kilpatrick ZP: **Two-dimensional bumps in piecewise smooth neural fields with synaptic depression.** *SIAM J Appl Math* 2011.
15. Sadaghiani S, Hesselmann G, Friston KJ, Kleinschmidt A: **The relation of ongoing brain activity, evoked neural responses, and cognition.** *Front Syst Neurosci* 2010.

ДОДАТКИ

ДОДАТОК А

Лістинг програмного коду інтерактивної форми

```
import math
import numpy
import pygame
from scipy.misc import imsave
from scipy.ndimage.filters import gaussian_filter


class AmariModel(object):

    def __init__(self, size):
        self.h = -0.1
        self.k = 0.05
        self.K = 0.125
        self.m = 0.025
        self.M = 0.065

        self.stimulus = -self.h * numpy.random.random(size)
        self.activity = numpy.zeros(size) + self.h
        self.excitement = numpy.zeros(size)
        self.inhibition = numpy.zeros(size)

    def stimulate(self):
        self.activity[:, :] = self.activity > 0

        sigma = 1 / math.sqrt(2 * self.k)
        gaussian_filter(self.activity, sigma, 0, self.excitement, "wrap")
        self.excitement *= self.K * math.pi / self.k

        sigma = 1 / math.sqrt(2 * self.m)
        gaussian_filter(self.activity, sigma, 0, self.inhibition, "wrap")
        self.inhibition *= self.M * math.pi / self.m

        self.activity[:, :] = self.h
        self.activity[:, :] += self.excitement
        self.activity[:, :] -= self.inhibition
        self.activity[:, :] += self.stimulus


class AmariMazeGenerator(object):

    def __init__(self, size):
        self.model = AmariModel(size)

        pygame.init()
        self.display = pygame.display.set_mode(size, 0)
        pygame.display.set_caption("Amari Maze Generator")

    def run(self):
        pixels = pygame.surfarray.pixels3d(self.display)

        index = 0
        running = True
```

```
        while running:
            self.model.stimulate()

            pixels[:, :, :] = (255 * (self.model.activity > 0))[:, :, None]
            pygame.display.flip()

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False
                elif event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_ESCAPE:
                        running = False
                    elif event.key == pygame.K_s:
                        imsave("{0:04d}.png".format(index), pixels[:, :, 0])
                        index = index + 1
                elif event.type == pygame.MOUSEBUTTONDOWN:
                    position = pygame.mouse.get_pos()
                    self.model.activity[position] = 1

        pygame.quit()


def main():
    generator = AmariMazeGenerator((512, 512))
    generator.run()


if __name__ == "__main__":
    main()
```

ДОДАТОК Б

Лістинг програмного коду тестування застосунку

```
#include <gtest/gtest.h>

#include <vector>

#include "model/cave.h"
#include "model/matrix.h"
#include "model/maze.h"
#include "model/pacman.h"

TEST(generateMaze, checkSize) {
    s21::Maze maze;
    maze.setSizes(10, 10);
    ASSERT_EQ(maze.getRows(), 10);
    ASSERT_EQ(maze.getColumns(), 10);
}

TEST(generateMaze, checkVerticlal) {
    s21::Maze maze;
    maze.setSizes(10, 10);
    maze.generateMaze();
    auto matrix = maze.getVerticalWalls();
    for (int i = 0; i < maze.getRows(); i++) {
        ASSERT_EQ(matrix(i, maze.getColumns() - 1), true);
    }
}

TEST(generateMaze, checkHorizontal) {
    s21::Maze maze;
    maze.setSizes(10, 10);
    maze.generateMaze();
    auto matrix = maze.getHorizontalWalls();
    for (int i = 0; i < maze.getColumns(); i++) {
        ASSERT_EQ(matrix(maze.getRows() - 1, i), true);
    }
}

TEST(readFromFile, checkMaze) {
    s21::Maze maze;
    maze.setFromFile("datasets/10x10.txt");
}

TEST(findWay10x10, checkPacman) {
    s21::Maze maze;
    maze.setFromFile("datasets/10x10.txt");
    s21::Pacman pacman;
    pacman.getMaze(&maze);
    auto way = pacman.findWay({0, 0}, {5, 5});
    int result[]{5, 5, 5, 4, 6, 4, 6, 3, 5, 3, 4, 3, 4, 4, 3, 4, 3,
                 3, 2, 3, 1, 3, 1, 2, 2, 2, 2, 1, 2, 0, 1, 0, 0, 0};
    for (int i = 0, pos = 0; i < way.size(); i++) {
        ASSERT_EQ(way[i].x, result[pos++]);
        ASSERT_EQ(way[i].y, result[pos++]);
    }
}
```

```
TEST(findWay20x20, checkPacman) {
    s21::Maze maze;
    maze.setFromFile("datasets/20x20.txt");
    s21::Pacman pacman;
    pacman.getMaze(&maze);
    auto way = pacman.findWay({10, 15}, {8, 3});
     int result[]{8,  3,  8,  4,  7,  4,  6,  4,  5,  4,  4,  4,  3,  4,  2,  4,
2,  5,  3,  5,  3,
                   6,  2,  6,  1,  6,  1,  5,  1,  4,  1,  3,  1,  2,  2,  2,  3,
2,  3,  1,  2,  1,
                   1,  1,  0,  1,  0,  0,  1,  0,  2,  0,  3,  0,  4,  0,  4,  1,
4,  2,  5,  2,  6,
                   2,  7,  2,  8,  2,  9,  2,  10, 2,  10, 3,  9,  3,  9,  4,  9,
5,  9,  6,  8,  6,
                   7,  6,  6,  6,  5,  6,  4,  6,  4,  7,  5,  7,  5,  8,  5,  9,
5,  10, 6,  10, 6,
                   9,  6,  8,  6,  7,  7,  7,  7,  8,  7,  9,  8,  9,  8,  8,  8,
7,  9,  7,  10, 7,
                  11, 7,  12, 7,  12, 8,  11, 8,  11, 9,  10, 9,  10, 10, 11, 10,
12, 10, 12, 11, 12,
                  12, 12, 13, 11, 13, 10, 13, 10, 14, 11, 14, 12, 14, 12, 15, 11,
15, 10, 15};
    for (int i = 0, pos = 0; i < way.size(); i++) {
        ASSERT_EQ(way[i].x, result[pos++]);
        ASSERT_EQ(way[i].y, result[pos++]);
    }
}

TEST(readFromFile50x50, checkCave) {
    s21::Cave cave;
    cave.readFile("datasets/cave.txt");
    ASSERT_EQ(cave.getRows(), 50);
    ASSERT_EQ(cave.getCols(), 50);
}

TEST(readFromFile10x10, checkCave) {
    s21::Cave cave;
    cave.readFile("datasets/10x10.txt");
    ASSERT_EQ(cave.getRows(), 10);
    ASSERT_EQ(cave.getCols(), 10);
}

TEST(generateCave, checkSize) {
    s21::Cave cave;
    cave.generateMap(30, 17, 6);
    ASSERT_EQ(cave.getRows(), 17);
    ASSERT_EQ(cave.getCols(), 6);
}

TEST(generateCave, update) {
    s21::Cave cave;
    cave.readFile("datasets/10x10.txt");
     bool map[] = {0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1,
0, 1, 0, 0,
                   1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0,
1, 0, 1, 1,
```

```
                    1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0,
0, 0, 0, 1,
                    0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1,
0, 0, 0, 1};
    for (int i = 0, pos = 0; i < cave.getRows(); i++)
        for (int j = 0; j < cave.getCols(); j++)
            ASSERT_EQ(cave.getMap()(i, j), map[pos++]);
     bool result[] = {1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1,
1, 1, 1, 1, 0,
                     0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
1, 0, 1, 1, 1,
                     0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1,
1, 0, 0, 1, 1,
                     1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1};
    cave.setLimits(3, 4);
    cave.updateMap();
    for (int i = 0, pos = 0; i < cave.getRows(); i++)
        for (int j = 0; j < cave.getCols(); j++)
            ASSERT_EQ(cave.getMap()(i, j), result[pos++]);
}

int main(int argc, char* argv[]) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```