Національний університет « Чернігівський колегіум » імені Т.Г.Шевченка
Природничо-математичний факультет
Кафедра інформатики і обчислювальною техніки

Кваліфікаційна робота

освітнього ступеня « бакалавр »

на тему

РОЗРОБКА СИСТЕМИ

ВІЗУАЛІЗАЦІЇ АЛГОРИТМІВ ПОШУКУ ШЛЯХУ В ЛАБІРИНТІ

Виконав:
студент 4 курсу, 44-фмт групи
спеціальності
122 Комп’ютерні науки
Фесько Андрій Антонович
Науковий керівник:
к.п.н., доц. Вінниченко Є.Ф.

Чернігів - 2025

Роботу подано до розгляду «_____ » _____202__року.

Студент _____ _____
(підпис) (прізвище та ініціали)

Науковий керівник _____ _____
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Кваліфікаційна робота розглянута на засіданні кафедри
Інформатики і обчислювальної техніки
протокол № ____ від «_____ » _____ 202__ року
Студент допускається до захисту даної роботи в екзаменаційній комісії

Завідувач кафедри _____ _____
(підпис) (прізвище та ініціали)

Анотація

Кваліфікаційна робота присвячена розробці системи візуалізації алгоритмів пошуку шляху в лабіринті.
В ході роботи були вирішені наступні задачі:

1. Проведено аналіз алгоритмів пошуку шляху.
2. Спроектовано архітектуру програмної системи.
3. Реалізовано візуалізацію алгоритмів Дейкстри, A* та хвильового пошуку.
4. Проведено тестування та аналіз ефективності реалізованих методів.

Ключові слова: пошук шляху, алгоритм Дейкстри, алгоритм A*, візуалізація, лабіринт.

Abstract

The qualification work is dedicated to the development of a system for visualizing pathfinding algorithms in a maze.
The following tasks were completed:

1. Analyzed pathfinding algorithms.
2. Designed the system architecture.
3. Implemented visualization of Dijkstra’s algorithm, A*, and wave propagation.
4. Conducted testing and analyzed the efficiency of implemented methods.

Keywords: pathfinding, Dijkstra’s algorithm, A* algorithm, visualization, maze.

ЗМІСТ

ВСТУП

У сучасному світі інформаційних технологій усе більшої актуальності набуває потреба у створенні наочних, інтуїтивно зрозумілих засобів вивчення алгоритмів і структур даних. Зокрема, алгоритми пошуку шляху є фундаментальними для багатьох галузей, зокрема робототехніки, комп'ютерної графіки, навігаційних систем, інтелектуального аналізу даних, а також створення комп'ютерних ігор. Розуміння принципів їх роботи, а також можливість візуалізувати процес виконання дозволяє ефективніше аналізувати, налагоджувати та оптимізувати складні програмні рішення.

Однією з поширених форм застосування алгоритмів пошуку шляху є побудова маршрутів у лабіринтах. Лабіринт — це умовна модель простору, що представлена у вигляді двовимірної сітки або графа, по якій здійснюється пошук найкоротшого або оптимального шляху від початкової до цільової точки. Саме задача пошуку шляху в лабіринті виступає зручною моделлю для демонстрації та порівняння різних алгоритмів.

Актуальність даної роботи полягає у відсутності універсальних простих у використанні систем, які дозволяють в інтерактивному режимі порівнювати роботу різних алгоритмів пошуку шляхів із візуальним супроводом кожного кроку. Існуючі рішення здебільшого або надто складні у використанні, або обмежені у функціоналі. Тому створення такої системи може бути корисним як для викладачів і студентів ІТ-спеціальностей, так і для розробників, які потребують інструменту для аналізу алгоритмів.

Метою даної кваліфікаційної роботи є розробка програмного забезпечення, яке дозволяє візуалізувати процес пошуку шляху в лабіринті за допомогою різних алгоритмів. Система повинна надати можливість створення випадкового лабіринту, вибору алгоритму, запуску візуалізації та детального спостереження за процесом.

Для досягнення поставленої мети були сформульовані наступні завдання:

- Провести аналітичний огляд алгоритмів пошуку шляху та програм, що реалізують їх візуалізацію;
- Визначити функціональні та нефункціональні вимоги до системи;
- Спроектувати архітектуру програмного продукту;
- Реалізувати користувацький інтерфейс та логіку візуалізації;
- Здійснити тестування та оцінити результати роботи алгоритмів на прикладах;
- Створити інструкції користувача.

Об'єктом дослідження є процес пошуку шляху в дискретному просторі.

Предметом дослідження є алгоритми пошуку шляху та методи їх візуалізації в інтерактивному середовищі.

Методологічну основу роботи становлять методи структурного аналізу, об'єктно-орієнтованого програмування, моделювання, а також візуалізації графів. Основними методами дослідження виступають: аналіз літератури та інтернет-ресурсів, проектування програмної архітектури, розробка інтерфейсу, тестування програмного забезпечення.

Практична значущість результатів полягає в створенні зручного навчального інструменту для вивчення алгоритмів пошуку шляху. Запропоновану систему можна використовувати у закладах освіти як допоміжний засіб при викладанні курсів з алгоритмів, структури даних або штучного інтелекту. Крім того, її можна розширити і адаптувати для інших дослідницьких або комерційних цілей.

РОЗДІЛ 1. Аналітичний склад та постановка задачі

1.1. Поняття лабіринтів, графів та задач пошуку шляху

Задача пошуку шляху є однією з ключових задач у теорії графів і знаходить широке застосування в комп'ютерних науках. Лабіринт у цьому контексті розглядається як двовимірне дискретне середовище, яке може бути подано у вигляді графа. У такому графі вузли відповідають клітинкам лабіринту, а ребра — можливостям переходу між ними. Класично лабіринт визначається як сітка з прохідними та непрохідними комірками, де необхідно знайти шлях від стартової до цільової точки.[1]

У загальному випадку, лабіринт можна подати як неорієнтований граф G(V, E), де V — множина вершин (комірок), E — множина ребер (допустимих переходів між комірками). У простому прямокутному лабіринті із 4-сусідністю (вгору, вниз, вліво, вправо), кожна прохідна клітинка має до чотирьох можливих з'єднань. У разі використання 8-сусідності додаються діагональні переходи.

Задача пошуку шляху полягає в знаходженні найкоротшого або найефективнішого маршруту між двома заданими вершинами графа (в нашому випадку — між стартом і фінішем у лабіринті). У залежності від постановки, це може бути задача знаходження будь-якого шляху, найкоротшого шляху або шляху з урахуванням певних критеріїв (наприклад, ваги переходів).

Пошук шляху в лабіринті широко застосовується в таких галузях:

- Робототехніка (навігація мобільних роботів);
- Комп'ютерні ігри (штучний інтелект персонажів);
- Геоінформаційні системи (маршрутизація);
- Логістика (оптимізація маршрутів);
- Біоінформатика (аналіз структур білків);
- Аналіз графів і мережевих структур.

1.2. Огляд алгоритмів пошуку шляху

Алгоритми пошуку шляху можна умовно поділити на два основних типи: ненаправлені (які просто досліджують простір) і евристичні (які враховують припущення щодо відстані до цілі). Обидва типи використовуються в комп'ютерній науці, візуалізації, ігровому програмуванні та аналізі даних.

Пошук у ширину (Breadth-First Search, BFS) — це алгоритм, який досліджує граф рівнями. Він додає у чергу всі сусідні вершини перед переходом до наступного рівня. Це робить його дуже ефективним для знаходження найкоротшого шляху у невагомих графах. BFS використовує структуру даних чергу (queue) для зберігання вершин, які потрібно відвідати. Його переваги — простота реалізації та гарантія знаходження найкоротшого шляху. Недоліки — неефективність у великих графах через велику кількість пам'яті та часу.[2]

Пошук у глибину (Depth-First Search, DFS) — працює з використанням стека, занурюючись у глибину графа настільки, наскільки можливо, і лише потім повертаючись до попередніх вузлів. DFS добре підходить для перевірки існування шляху або пошуку рішень у лабіринтах з великою глибиною. Однак цей алгоритм не гарантує знаходження найкоротшого шляху, особливо у випадках складних або розгалужених лабіринтів.[2]

Алгоритм Дейкстри (Dijkstra's Algorithm) — один з найвідоміших жадібних алгоритмів пошуку шляху. Він призначений для графів з невід'ємними вагами ребер і шукає шлях з мінімальною сумою ваг. У контексті лабіринтів, де всі переходи мають однакову вагу, цей алгоритм працює схоже на BFS, але з використанням пріоритетної черги. Перевага — оптимальність шляху, недолік — висока складність.[3]

Алгоритм A* — покращення Дейкстри, яке додає до оцінки вартості шляху евристичну функцію. Найчастіше використовуються:

- Манхеттенська відстань (для сітки з 4-сусідністю);
- Евклідова відстань (для графа з можливими діагоналями).

Це дозволяє A* не тільки ефективно знаходити найкоротший шлях, а й робити це швидше за Дейкстру, особливо у великих графах. Недолік — залежність від правильності вибору евристики.

У даній роботі обрано для реалізації BFS, DFS, A* та алгоритм Дейкстри. Це дає змогу порівняти жадібні та евристичні підходи, оцінити їхню продуктивність та поведінку на однакових даних.

Варто зазначити, що навіть прості алгоритми, такі як DFS, можуть демонструвати хороші результати в спеціалізованих задачах або при правильному обмеженні глибини пошуку. У той час як BFS і Дейкстра забезпечують найкоротший шлях, вони не завжди оптимальні з точки зору продуктивності.

У випадку складних, динамічних середовищ (наприклад, рухомих перешкод), можуть застосовуватися адаптивні алгоритми, такі як ***D* Lite***, які дозволяють повторно використовувати попередні обчислення при зміні середовища.

Деякі дослідження фокусуються на модифікаціях A*, наприклад, ***алгоритмі IDA**** (Iterative Deepening A*), який використовує глибину пошуку як обмеження, що дозволяє зменшити витрати пам’яті.

Іншим прикладом є алгоритм ***Jump Point Search*** (JPS), який оптимізує пошук шляхів у сітках, скорочуючи кількість непотрібних перевірок завдяки симетрії.

Також варто згадати ***Theta**** — алгоритм, що базується на A*, але дозволяє прокладати прямі шляхи через кілька клітин, якщо між ними немає перешкод, що робить його більш природним з точки зору руху.

Загальна ефективність алгоритмів залежить від типу графа, кількості вершин і ребер, розміру простору пошуку, структури лабіринту та наявності евристики. Тому вибір конкретного алгоритму завжди повинен базуватись на контексті задачі.

Порівняння характеристик різних алгоритмів зображено в Таблиці 1.

Таблиця 1. Характеристика алгоритмів

Алгоритм	Гарантія найкоротшого шляху	Евристика	Складність	Структура даних	Пам’ять	Швидкість	Коментар
BFS	Так	Ні	O(V+E)	Черга	Середня	Середня	Простий у реалізації
DFS	Ні	Ні	O(V+E)	Стек	Низька	Висока	Не оптимальний шлях
Дейкстра	Так	Ні	O((V+E)log V)	Пріоритетна черга	Висока	Низька	Важкий для великих графів
A*	Так	Так	Залежить від евристики	Пріоритетна черга +r heuristics	Висока	Висока	Оптимальний при хорошій евристиці
JPS	Так	Так	Менша за A*	Залежить від реалізації	Середня	Дуже висока	Оптимально для сіток
IDA*	Так	Так	O(b^d) менше пам’яті	Рекурсивна реалізація	Низька	Середня	Знижене споживання пам’яті
Theta*	Так	Так	Залежить від евристики	Лінії прямого виду	Середня	Висока	Природніші маршрути

1.3. Порівняння існуючих рішень для візуалізації алгоритмів

Візуалізація алгоритмів є важливою складовою процесу навчання та розуміння складних структур даних і принципів функціонування програмних систем. У випадку з алгоритмами пошуку шляху вона дозволяє зробити процес більш інтуїтивним і доступним, особливо для студентів і молодих розробників. На сьогоднішній день існує велика кількість як комерційних, так і безкоштовних програмних продуктів, які дозволяють демонструвати роботу алгоритмів пошуку шляху в інтерактивній формі.

Одним з найпопулярніших інструментів є Pathfinding Visualizer, розроблений Клементом Михайлеску. Цей веб-застосунок реалізований на JavaScript з використанням React і дозволяє візуалізувати алгоритми BFS, DFS, Dijkstra та A*. Користувач може самостійно створювати або редагувати лабіринти, розміщувати початкову та цільову точки, а потім запускати вибраний алгоритм. Важливою перевагою є можливість бачити процес пошуку у режимі реального часу.

Інтерфейс Pathfinding Visualizer простий у використанні, що робить його зручним для новачків. Проте функціональні можливості обмежені: користувач не може змінювати параметри алгоритмів, задавати власні правила обходу або евристики. Також немає детальної статистики про перебіг виконання, що ускладнює використання цього ресурсу для глибокого аналізу ефективності алгоритмів.[4]

Іншим цікавим ресурсом є VisuAlgo.net — освітній веб-портал, що містить інтерактивні демонстрації багатьох алгоритмів, включаючи ті, що працюють з графами. Цей ресурс створений з акцентом на навчальну складову: кожен алгоритм супроводжується поясненням, покроковими підказками та контрольними запитаннями. Він особливо корисний для підготовки до іспитів або конкурсів з алгоритмів.

VisuAlgo підтримує лише статичні приклади або обмежені варіанти графів. Користувач не має можливості створювати власні лабіринти або

змінювати розміри сітки. Це значно знижує інструментальну цінність ресурсу для практичного застосування чи дослідницької діяльності. Проте він ідеально підходить для ознайомлення з базовими принципами алгоритмів.[5]

MazeSolver — це приклад іншого підходу до візуалізації алгоритмів пошуку шляху. Цей інструмент дає змогу завантажувати зображення лабіринту, яке система обробляє за допомогою алгоритмів комп'ютерного зору для розпізнавання структури. Після цього запускається алгоритм пошуку шляху, і користувач може спостерігати за результатом його роботи у вигляді візуалізації.

Водночас MazeSolver не підтримує класичну інтерактивність: користувач не має можливості змінювати лабіринт або впливати на хід виконання алгоритму. Таким чином, цей ресурс найкраще підходить для швидкого аналізу та демонстрації роботи алгоритмів на заздалегідь підготовлених прикладах. Це робить MazeSolver корисним інструментом для ілюстрації практичного поєднання алгоритмів пошуку з методами обробки зображень, але менш придатним для глибшого експериментування чи навчання.[6]

Загальний вигляд графічного інтерфейсу MazeSolver показаний на Рис 1.1.

GraphPathFinder — менш популярний, проте гнучкий інструмент, що дозволяє створювати графи довільної топології. Цей застосунок орієнтований на більш досвідчених користувачів, які хочуть дослідити поведінку алгоритмів не лише в сіткових структурах, але й у складних мережах. На жаль, графічна реалізація інтерфейсу не завжди є зручною, а сам ресурс потребує доопрацювання з точки зору використання.

Іншим представником є Algovisor — інструмент, орієнтований на візуалізацію класичних алгоритмів, таких як сортування, пошук у графах, побудова дерев. Він має гарну анімацію та підтримку покрокового виконання, але не фокусується спеціально на лабіринтах або сіткових просторах. Це обмежує сферу його застосування у рамках теми пошуку шляху.

AlgoVis.io — сучасний онлайн-інструмент для візуалізації алгоритмів пошуку шляху. Платформа дозволяє користувачам спостерігати покрокове виконання таких алгоритмів, як BFS, DFS, Дейкстра та A*, на інтерактивній сітці. Користувачі можуть налаштовувати розміри лабіринту, встановлювати стіни та стартові й цільові точки, що робить процес навчання більш гнучким і наочним.

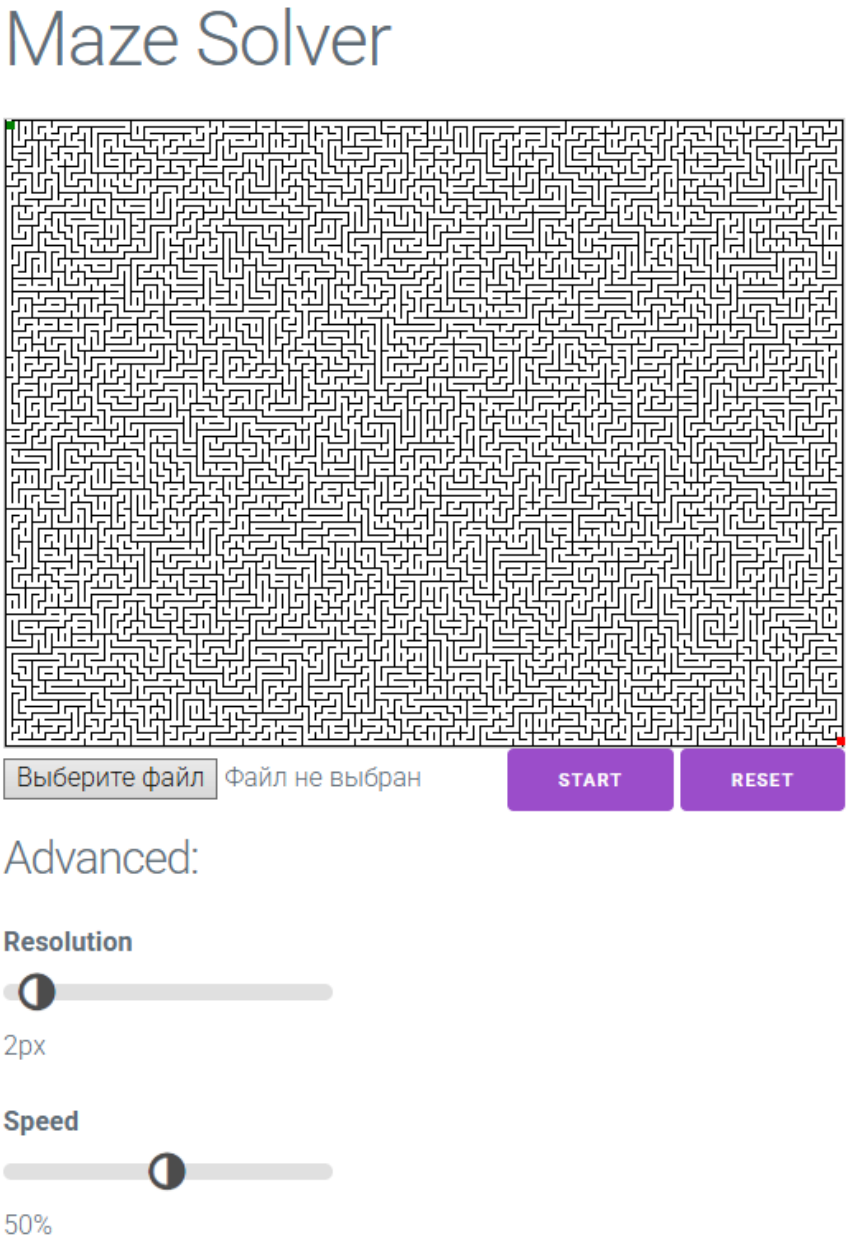

Рис. 1.1. Загальний вигляд графічного інтерфейсу веб-додатку MazeSolver

AlgoVis.io надає можливість порівнювати ефективність різних алгоритмів у реальному часі, демонструючи кількість відвіданих клітинок, тривалість пошуку та якість знайденого шляху. Це робить платформу корисним

інструментом не лише для студентів і викладачів, а й для розробників, які бажають краще зрозуміти особливості роботи алгоритмів. Завдяки відкритому коду та доступності в браузері AlgoVis.io є зручним засобом для вивчення алгоритмів пошуку шляху.[7]

Загальний вигляд графічного інтерфейсу AlgoVis.io показаний на Рис 1.2.

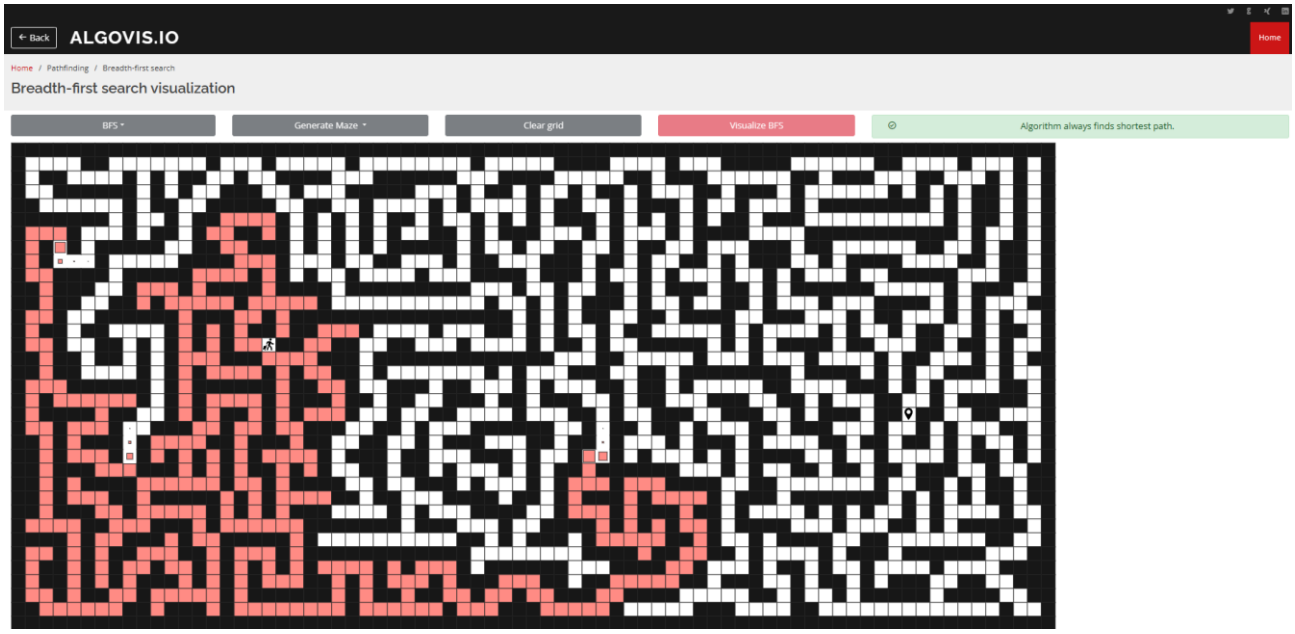

Рис. 1.2. Загальний вигляд графічного інтерфейсу онлайн-інструменту AlgoVis.io

Також існую така бібліотека для JavaScript, як Pathfinding.js. Ця бібліотека дозволяє інтегрувати алгоритми пошуку шляху у веб-додатки. Вона підтримує алгоритмі Dijkstra, A*, BFS та DFS та надає інструменти для налаштування сіткових параметрів. Pathfinding.js широко використовується в інтерактивних навчальних платформах і тестуванні логіки штучного інтелекту у відеоіграх. Основною перевагою є можливість легко інтегрувати її у власні проекти без необхідності використовувати готові веб-застосунки.

Algorithm Visualizer - це онлайн-платформа з відкритим кодом, яка дозволяє користувачам переглядати візуалізацію різних алгоритмів, включаючи пошук шляху, сортування та роботу з графами. Користувачі можуть змінювати параметри алгоритмів, тестувати різні сценарії та переглядати покрокове виконання процесів. Цей інструмент особливо корисний для студентів та

розробників, які хочуть глибше зрозуміти принципи роботи алгоритмів без необхідності писати код самостійно.[8]

Загальний вигляд графічного інтерфейсу веб-додатку Algorithm Visualizer показаний Рис 1.3.

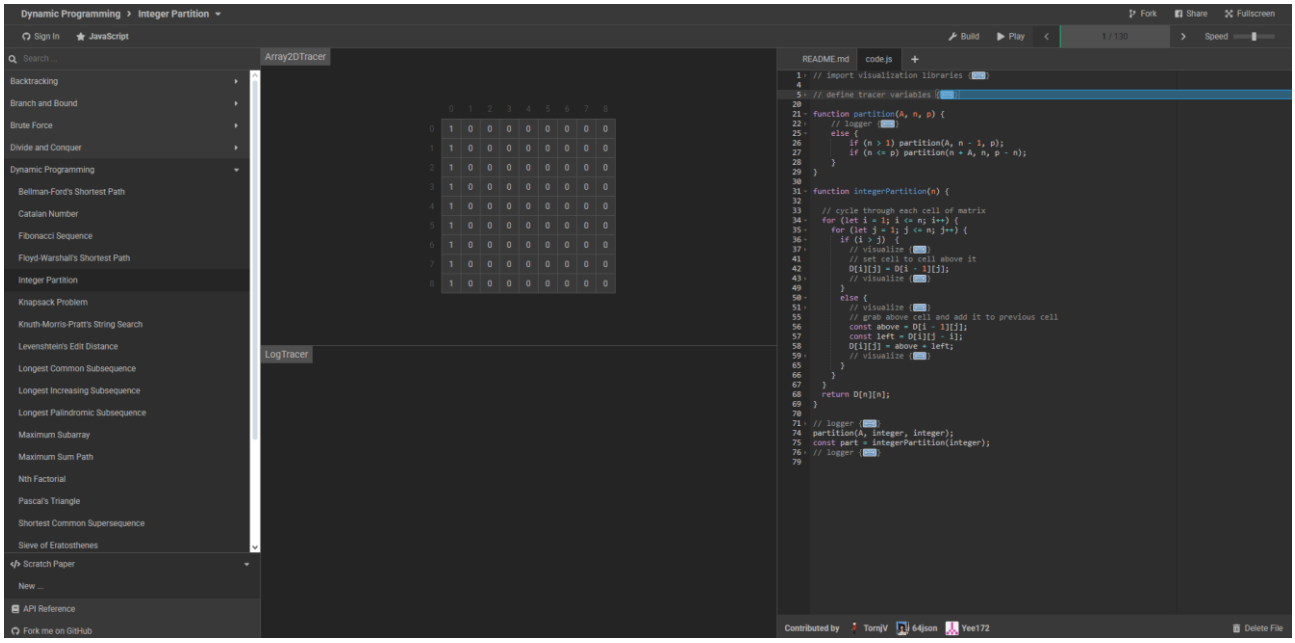

Рис. 1.3. Загальний вигляд графічного інтерфейсу веб-додатку Algorithm Visualizer

Значною перевагою візуалізаторів є можливість роботи у веб-браузері без необхідності встановлення додаткового програмного забезпечення. Це спрощує доступ до ресурсу та дає змогу швидко почати роботу навіть на слабких пристроях.

Проте саме веб-орієнтовані інструменти часто страждають від обмеженої продуктивності при великих обсягах даних. Наприклад, робота з великими лабіринтами (100x100 і більше) може призвести до помітного зниження швидкодії або зависань браузера.

Ще однією поширеною проблемою є відсутність персоналізації: користувачі не мають змоги зберігати свої налаштування, сценарії або результати. Це особливо важливо для викладачів або дослідників, які хочуть використовувати інструмент на постійній основі.

Чимало візуалізаторів орієнтовані виключно на прямокутні сітки, що обмежує можливість дослідження інших форм графів, наприклад, шестикутних,

тривимірних або зважених з’єднань. Таке спрощення, хоча і зручне для новачків, знижує універсальність застосунку.

У перспективі важливо створювати інструменти, які дозволяли б не лише запускати алгоритми, а й модифікувати їх. Можливість редагувати код або логіку обходу у візуальному середовищі відкриває нові горизонти для навчання і досліджень.

Найперспективнішими виглядають ті системи, які поєднують простоту інтерфейсу з можливістю глибокого контролю над процесом: зміна алгоритмів, параметрів пошуку, типу графа, формування звітів і аналізу ефективності.

Отже, огляд наявних рішень дозволяє зробити висновок, що, попри значну кількість ресурсів, лише небагато з них відповідають вимогам універсальності, адаптивності та гнучкості. Більшість систем обмежуються демонстраційною функцією, залишаючи за межами уваги дослідницькі й освітні задачі більш високого рівня.

1.4. Визначення вимог до системи візуалізації алгоритмів пошуку шляху

На основі аналізу предметної області, огляду алгоритмів пошуку шляху та існуючих візуалізаторів можна сформулювати загальні та специфічні вимоги до розроблюваної програмної системи. Ці вимоги поділяються на функціональні, нефункціональні, а також обмеження проекту, які впливають на архітектуру та реалізацію майбутнього програмного продукту.

Формулювання вимог перед початком реалізації проекту є важливим етапом, який визначає межі проекту, основні сценарії використання, необхідні компоненти та очікувану поведінку системи. Правильне визначення вимог дозволяє уникнути непорозумінь між розробником та кінцевим користувачем, а також забезпечити відповідність результату початковим цілям.

Основною метою системи є забезпечення інтерактивної візуалізації роботи пошукових алгоритмів у лабіринті. Вона повинна бути зручною, інтуїтивно зрозумілою, придатною як для навчальних, так і для демонстраційних цілей, та дозволяти змінювати параметри в реальному часі.

Функціональні вимоги.

Функціональні вимоги описують, які саме функції повинна виконувати система. Вони є основою для побудови архітектури програмного забезпечення та визначення модулів. Для розроблюваної системи такі вимоги можуть бути сформульовані наступним чином:

1. Побудова лабіринту:
    - Можливість створення випадкового лабіринту;
    - Підтримка різних розмірів поля.
2. Вибір алгоритму пошуку:
    - Підтримка вибору одного з реалізованих алгоритмів;
    - Відображення короткої інформації про обраний алгоритм перед запуском.
3. Інтерактивна візуалізація:
    - Покрокова візуалізація процесу пошуку;

- Виділення оброблюваних кліток, відвіданих вершин та остаточного шляху;
- Можливість керування швидкістю виконання;
- Можливість призупинення, відновлення, скасування виконання.

4. Керування початковими параметрами:
   - Встановлення стартової та цільової позиції у довільній клітинці;
   - Повне очищення поля.
5. Виведення додаткової інформації:
   - Відображення статистики кількості відвіданих клітинок, довжину знайденого шляху та час виконання;
   - Короткий лог виконання алгоритму.
6. Збереження та відновлення стану:
   - Можливість зберегти поточний стан лабіринту у форматі JSON або іншому структурованому форматі.

Нефункціональні вимоги.

Нефункціональні вимоги визначають якість роботи системи, її продуктивність, масштабованість, зручність використання та інші аспекти, які не пов’язані безпосередньо з її функціоналом, але значною мірою впливають на користувацький досвід.

1. Продуктивність:
   - Обробка пошуку шляху повинна бути швидка;
   - Анімація повинна бути плавною, без ривків і затримок.
2. Масштабованість:
   - Система повинна підтримувати розширення - можливість додавання нових алгоритмів без значної модифікації коду;
   - Можливість змінити структури даних.

3. Кросплатформеність:
    - Програмне забезпечення має коректно працювати в різних операційних системах;
    - Забезпечення адаптивного інтерфейсу для різних розмірів екрану.
4. Зручність у використанні:
    - Інтерфейс повинен бути простим і зручним без необхідності вивчення інструкції;
    - Логічне розташування кнопок, підсказки при наведенні, графічні позначення для елементів.

Обмеження та припущення

1. У даній реалізації передбачається використання прямокутної сітки з 4-сусідністю. Підтримка діагоналей або тривимірного простору не планується.
2. Розміри поля обмежуються до 100x100 клітинок для збереження стабільної роботи.
3. Застосунок реалізується як програмне забезпечення. Усі обчислення виконуються на стороні користувача програмного забезпечення.
4. Передбачається, що цільова аудиторія - це студенти, викладачі, розробники-початківці, тобто користувачі з базовими знаннями у сфері алгоритмів.
5. На етапі розробки система не передбачає інтеграцію з іншими платформами або застосунками.

Отже, чітке визначення вимог дозволяє сформувати основу технічного завдання та визначити функціональні блоки системи. Такий підхід забезпечить відповідність реалізації очікуванням користувачів і дозволить створити ефективний інструмент для візуалізації алгоритмів пошуку шляху.

РОЗДІЛ 2. Проектні рішення

2.1. Формування задачі та постановка проблеми

У сучасному світі візуалізація алгоритмів є важливим інструментом для навчання, демонстрації та аналізу складних обчислювальних процесів. Зокрема, алгоритми пошуку шляху відіграють ключову роль у багатьох сферах — від комп’ютерних ігор до навігаційних систем і робототехніки. Саме тому актуальною є задача створення програмного засобу, який дозволяє не лише реалізувати відомі алгоритми пошуку, а й візуалізувати їхню роботу в реальному часі.

Основною метою проекту є розробка програмної системи візуалізації алгоритмів пошуку шляху в лабіринті. Така система повинна забезпечити користувачу змогу обрати один з кількох класичних алгоритмів (BFS, DFS, Dijkstra, A*), обрати конфігурацію лабіринту та переглянути, як саме алгоритм будує маршрут до цільової точки або повідомляє про відсутність такого.

Для точного формулювання задачі доцільно подати її у вигляді математичної моделі. Вхідні дані — це прямокутна сітка розміром $n \times m$, де кожна клітинка відповідає вершині графа. Ребра існують між суміжними клітинками, які не є стінами. Лабіринт можна подати як неорієнтований граф $G = (V, E)$, де:

- V — множина вершин (клітинки лабіринту);
- E — множина ребер між сусідніми клітинками;
- s — стартова вершина;
- t — цільова вершина.

Задача полягає у пошуку шляху з s до t з урахуванням умов:

- обхід можливий лише через прохідні клітинки;
- при використанні A* або Dijkstra шлях повинен бути оптимальним за довжиною;

- при використанні BFS — гарантується найкоротший шлях у ненавантаженому графі;
- при використанні DFS — шлях може бути довший, але обхід завершується при знаходженні цілі.

Крім пошуку шляху, система має забезпечувати візуальний супровід процесу, що включає:

- відображення відвіданих клітинок у процесі роботи алгоритму;
- поступове малювання знайденого шляху;
- оновлення статистики (час виконання, назва алгоритму, результат проходження).

У рамках постановки задачі також потрібно врахувати обмеження:

- алгоритм повинен працювати на обмеженому полі (21×21 клітинка);
- час візуалізації не повинен перевищувати 1–2 хвилин у найскладніших випадках;
- відсутність шляху має бути коректно оброблена без зависання системи.

Для забезпечення якісної реалізації проекту необхідно також продумати інтерфейс користувача, який має бути інтуїтивно зрозумілим та зручним. Важливо надати можливість гнучкого налаштування параметрів алгоритмів та лабіринту, а також швидко отримувати зворотний зв’язок про хід виконання пошуку.

Таким чином, формалізована задача поєднує в собі класичну обчислювальну проблему (пошук у графі) та вимоги до її інтерактивного представлення для користувача. Це створює умови як для поглибленого вивчення алгоритмів, так і для побудови програмної архітектури, що підтримує гнучкість, стабільність і візуальну наочність.

2.2. Обґрунтування вибору алгоритмічного підходу

Для реалізації задачі пошуку шляху в лабіринті було обрано чотири класичних алгоритми: пошук у ширину (BFS), пошук у глибину (DFS), алгоритм Дейкстри та алгоритм A*. Кожен із них має свої унікальні особливості та підходить для вирішення задач із різною складністю, топологією та вимогами до оптимальності маршруту.

Різноманітність алгоритмів дозволяє реалізувати гнучку систему, яка охоплює як найпростіші методи, так і більш ефективні, але складніші евристичні підходи. Такий вибір є обґрунтованим з точки зору навчального, експериментального й демонстраційного застосування.

BFS (Breadth-First Search).

Цей алгоритм гарантує знаходження найкоротшого шляху в ненавантаженому графі, яким і є лабіринт у вигляді сітки. BFS розглядає всі вершини на одному рівні перш ніж переходити до наступного. Його перевагами є простота реалізації, передбачувана поведінка та можливість легкої візуалізації. BFS добре демонструє принципи рівномірного розширення області пошуку та дозволяє студентам легко спостерігати за логікою дій алгоритму.

У практичному сенсі, BFS є корисним для обробки рівномірно зважених графів, де основним критерієм є мінімальна кількість кроків до цілі. У випадку, якщо лабіринт містить багато перешкод, BFS може витратити більше часу через необхідність послідовного розширення фронту у всіх напрямках.[9]

DFS (Depth-First Search).

Пошук у глибину працює за принципом заглиблення до максимально можливої глибини, а потім повернення назад. Він швидко знаходить будь-який допустимий шлях, однак не гарантує його оптимальність. Цей підхід дозволяє дослідити структуру лабіринту на глибину й краще підходить для візуалізації жадібного просування алгоритму. Візуально він виглядає як вузький тунель, що рухається до цілі або в тупик.

Однією з важливих переваг DFS є економія пам’яті: він потребує лише стек для зберігання шляху. Це дозволяє використовувати його навіть у складніших топологіях. У навчальному контексті DFS добре ілюструє процес рекурсивного обходу графа та демонструє, як алгоритм поводиться при наявності численних розгалужень.[9]

Алгоритм Дейкстри.

Цей алгоритм призначено для знаходження найкоротшого шляху у графах з не від’ємними вагами. У нашому випадку, вага кожного переходу однакова, тому Дейкстра поводиться подібно до BFS, але використовує пріоритетну чергу, що дозволяє гнучко керувати порядком обробки вершин.

Дейкстра є класичним прикладом точного методу, який не потребує евристик. Його використання виправдане в ситуаціях, де гарантовано необхідно знайти найкоротший шлях незалежно від структури графа. Важливою особливістю є універсальність: алгоритм можна легко адаптувати під вагові графи, у тому числі — для побудови транспортних, логістичних і комунікаційних мереж.[9]

Алгоритм A*.

Цей алгоритм поєднує переваги пошуку за вартістю (як у Дейкстрі) та евристики, що значно пришвидшує пошук у великих і складних графах. A* використовує функцію оцінки $f(n) = g(n) + h(n)$, де $g(n)$ — фактична вартість шляху до поточної вершини, а $h(n)$ — евристична оцінка до цілі. У даному проекті для $h(n)$ використовується манхеттенська відстань між поточною точкою та виходом.

A* забезпечує найвищу ефективність серед представлених алгоритмів при правильному підборі евристики. Його робота особливо помітна на великих сітках або лабіринтах із довгими коридорами. A* дозволяє значно скоротити кількість оброблених клітинок, що має практичну цінність для задач із великим обсягом даних або обмеженими обчислювальними ресурсами.[9]

Запропонований набір алгоритмів охоплює усі основні стратегії обходу графів: від повного обстеження (BFS), до глибокого занурення (DFS), до

точного методу (Dijkstra) і евристичного підходу (A*). Це дозволяє системі не лише демонструвати алгоритми, але й порівнювати їхню поведінку в однакових умовах, що є важливим для розуміння принципів побудови маршрутів у складних структурах.

Крім того, така комбінація підходів сприяє глибшому засвоєнню алгоритмічного мислення. Користувач може на практиці переконатися, як змінюється ефективність пошуку залежно від вибору методу, і як різні стратегії впливають на кількість відвіданих клітинок, швидкість виконання та результат пошуку.

З точки зору реалізації, обрані алгоритми легко інтегруються у єдину платформу, оскільки базуються на однакових принципах проходження графа. Це спрощує як структурування коду, так і підтримку системи, дає змогу уникнути дублювання логіки та підвищити масштабованість рішення.

Важливо також відзначити, що інтеграція цих алгоритмів у єдину систему дозволяє не лише порівнювати їхню ефективність, а й підвищувати розуміння користувачем ключових концепцій пошуку шляхів у графах. Такий підхід сприяє глибшому засвоєнню матеріалу та розвитку аналітичних навичок, що є важливим для подальшого застосування отриманих знань у практичних задачах.

Таке поєднання робить програмну систему універсальним інструментом для навчання, тестування, дослідження й порівняльного аналізу методів пошуку шляху. Вибір саме цих чотирьох алгоритмів базується на їх науковій визнанності, зрозумілій логіці та широкому практичному застосуванню в різних галузях: від програмування роботів до оптимізації маршрутів і комп’ютерних ігор.

2.3. Архітектура та структура програмного рішення

Розроблена система візуалізації алгоритмів пошуку шляху побудована за модульним принципом із чітким розділенням логіки, візуального представлення та взаємодії з користувачем. Такий підхід дозволяє забезпечити стабільність, масштабованість та гнучкість програми при її розширенні чи модифікації.

Загальна архітектура відповідає принципам шаблону Model–View–Controller (MVC), який забезпечує логічне розділення коду на три основні частини:

- Model — частина, що відповідає за обчислення та реалізацію алгоритмів (BFS, DFS, Dijkstra, A*);
- View — графічне відображення лабіринту, шляху та анімації пошуку;
- Controller — обробка дій користувача (натискання кнопок, вибір алгоритму, запуск тощо).

Основні компоненти програми:

1. ***Модуль візуалізації*** (View):
   - Відповідає за побудову сітки лабіринту на основі масиву символів.
   - Здійснює оновлення кольору клітинок під час анімації алгоритмів.
   - Виводить кінцевий шлях та результати проходження.
2. ***Модуль алгоритмів*** (Model):
   - Містить окремі методи для реалізації кожного алгоритму пошуку.
   - Використовує делегати або callback-функції для передачі інформації у візуалізацію.
   - Працює з копією лабіринту, не змінюючи оригінальний стан масиву.
3. ***Інтерфейс користувача*** (Controller):
   - Складається з кнопок, випадаючих списків та таблиці результатів.

- Керує послідовністю дій: вибір → генерація → запуск → вивід результатів.
- Перевіряє правильність введених даних перед запуском алгоритмів.

4. ***Модуль генерації лабіринтів***:
   - Містить набір готових шаблонів лабіринтів.
   - Реалізує функцію створення випадкового лабіринту розміром 21×21.
   - Створює вхідні дані для подальшої обробки алгоритмами.
5. ***Модуль збереження результатів***:
   - Формує рядок статистики з назвою алгоритму, типом лабіринту, часом проходження та статусом успіху.
   - Додає новий рядок у таблицю після завершення кожного запуску алгоритму.
   - Допомагає візуально оцінити ефективність методів.

Взаємодія модулів.

Після вибору лабіринту та алгоритму користувач натискає кнопку запуску.

- Контролер ініціює генерацію або завантаження лабіринту.
- Потім викликається відповідна функція алгоритму з передачою поточної карти.
- Модуль алгоритму виконує пошук та передає координати для візуалізації.
- Після завершення пошуку результат відображається на панелі та у таблиці.

Переваги архітектурного підходу полягають у його масштабованості, що дозволяє легко додавати нові алгоритми або типи лабіринтів, модульності, завдяки якій можна оновлювати частини програми незалежно одна від одної, зрозумілості, оскільки логічна структура спрощує супровід та тестування, а

також гнучкості, що дає можливість адаптувати систему під різні графічні платформи або середовища.

2.4. Вибір технології та середовище розробки

Для реалізації програмної системи було обрано середовище розробки Visual Studio Community у поєднанні з мовою програмування C++/CLI та бібліотекою Windows Forms. Такий вибір зумовлений кількома ключовими факторами, серед яких: доступність інструментів, простота створення графічного інтерфейсу та наявність підтримки об’єктно-орієнтованого підходу.

Visual Studio є одним з найпотужніших середовищ для розробки застосунків під Windows. Його інтегрована підтримка проектів Windows Forms дозволяє зручно створювати візуальні елементи інтерфейсу за допомогою конструктора форм. Це значно прискорює процес реалізації GUI, знижуючи потребу в ручному розміщенні елементів через код. Крім того, Visual Studio має зручні інструменти налагодження (debugging), автодоповнення коду (IntelliSense) та систему керування проектами, що робить його надзвичайно ефективним у навчальних і практичних цілях.[10]

Мова C++/CLI була обрана завдяки можливості поєднання переваг традиційного C++ із сумісністю з .NET-платформою. Такий підхід дозволив реалізувати високопродуктивну логіку (наприклад, обчислення шляху в алгоритмах) при збереженні зручного доступу до графічних компонентів, подій, таблиць, кнопок та інших елементів Windows Forms. Крім того, використання C++/CLI дозволяє гнучко маніпулювати масивами, структурами та делегатами, що важливо для реалізації алгоритмів пошуку. Можливість створення окремих модулів логіки і подальша інтеграція з інтерфейсом через managed-код забезпечує зручну організацію архітектури проекту.

Середовище Windows Forms було обрано як компроміс між простотою використання та функціональністю. На відміну від інших рішень (наприклад, WPF або Qt), Windows Forms дозволяє швидко отримати стабільний графічний інтерфейс із мінімальним навантаженням на систему. Це ідеальне рішення для навчального проекту, де важлива швидкість розробки та зрозумілість коду.[11] Завдяки своїй простоті, Windows Forms також дозволяє легко розміщувати

візуальні елементи — наприклад, поле для відображення лабіринту, комбіновані списки для вибору, кнопки запуску, таблиці для результатів тощо. Для цього не потрібно мати складних знань із UI-дизайну, що дозволяє зосередитися на алгоритмах.

У рамках реалізації проекту було створено кілька окремих файлів-класів, які відповідають за конкретну функціональність: логіку руху, генерацію лабіринтів, зберігання шаблонів та реалізацію пошукових алгоритмів. Сам графічний інтерфейс побудовано за допомогою Windows Forms у візуальному редакторі, що дозволило значно скоротити час на верстку й розміщення елементів. Кожна кнопка, список або панель прив’язана до відповідного обробника подій, що дає змогу забезпечити чітку послідовність взаємодії між користувачем і програмою.

Однак під час вибору технологій було розглянуто й альтернативні варіанти. Одним із них був C# у поєднанні з WPF (Windows Presentation Foundation). Цей підхід дозволяє створювати сучасні, адаптивні інтерфейси з розширеною підтримкою графіки та анімацій. Проте, його використання потребує додаткового вивчення XAML та значно більших ресурсів. Для навчального проекту WPF виявився надлишковим за функціональністю, складнішим у реалізації та повільнішим у первинному освоєнні.

Ще одним можливим варіантом був Python з Tkinter або бібліотекою PyQt. Ці рішення мають відкритий код, є кросплатформеними, але менш продуктивні при обробці великих обсягів графіки, а реалізація алгоритмів потребує окремих структур і класів без вбудованої підтримки типізації. Крім того, GUI на базі Tkinter виглядає застарілим, а PyQt — складніший у налаштуванні та розповсюдженні.[11]

Також розглядалась можливість використання C++ з бібліотекою Qt. Це рішення мало б сенс у разі необхідності кросплатформеності або складніших графічних інтерфейсів. Проте Qt значно перевищує потреби даного навчального проекту за складністю, потребує окремого середовища (Qt Creator), та не

забезпечує такої швидкої інтеграції з візуальним редактором, як Windows Forms.[11]

Крім основних технологій, у системі було використано стандартні компоненти .NET Framework, що гарантує стабільність і сумісність із більшістю версій Windows. Весь проект не потребує додаткових зовнішніх бібліотек, що спрощує його розгортання та тестування. Таке рішення робить систему максимально незалежною від зовнішніх середовищ і дозволяє запускати її на будь-якому комп’ютері з базовими компонентами .NET.[12]

Особливу увагу також було приділено зручності користувача. Всі елементи інтерфейсу оптимізовано під фіксований розмір форми, що дозволяє уникнути спотворення вигляду при запуску на різних комп’ютерах. Елементи керування мають логічне розташування: вибір лабіринту та алгоритму, запуск, відображення таблиці результатів. Всі компоненти відображаються з врахуванням системного масштабу та дозволу екрана. Інтерфейс залишився стабільним навіть при запуску в середовищах із різними DPI або шрифтами.

Ще одним аргументом на користь обраного підходу є наявність широкої бази підтримки та документації. Всі інструменти, використані у проекті, активно підтримуються спільнотою розробників, мають приклади реалізацій і дозволяють легко шукати рішення можливих проблем. Це особливо важливо в навчальних проектах, де важливо не лише створити продукт, а й зрозуміти кожен етап його реалізації.

Окрім технічних переваг, обране середовище має також важливе значення з погляду зручності налагодження та тестування. Наявність інтегрованих інструментів для виявлення помилок у Visual Studio дозволяє зменшити час на діагностику проблем і підвищити якість розробки. Розробник може відслідковувати змінні, точки зупинки та потік виконання програми в режимі реального часу, що дуже важливо для систем із анімацією.

У середовищі C++/CLI досить легко реалізується взаємодія між логікою обробки та графічною частиною, завдяки використанню делегатів та подій. Цей механізм дозволяє передавати зворотні виклики з алгоритмів без необхідності

прив'язувати їх до візуального коду, що підтримує принцип розділення відповідальностей. Така архітектура значно полегшує як модульне тестування, так і майбутнє розширення програми.

Ще одним фактором, що вплинув на вибір Windows Forms, є наявність великої кількості навчальних ресурсів українською та англійською мовами. Завдяки цьому навіть користувачі-початківці можуть швидко ознайомитися з основами проектування графічного інтерфейсу, працювати з подіями, формами, компонентами таблиць і кнопок. Це підвищує загальну доступність технології для широкої аудиторії.

Завдяки використанню знайомих інструментів, уся система може бути легко перенесена, модифікована або навіть перероблена в іншу архітектуру без суттєвих витрат часу. Наприклад, за потреби її можна перевести в WPF або навіть веб-інтерфейс, залишивши основну логіку обробки без змін. Така гнучкість забезпечує довгострокову життєздатність розробки в умовах швидко змінних технологій.

Таким чином, зроблений вибір технологій є обґрунтованим з точки зору поєднання простоти, швидкодії та стабільності. Він ідеально підходить для реалізації системи навчального характеру, де головним пріоритетом є демонстрація логіки алгоритмів, а не складність дизайну або зовнішній вигляд програми. У наступному пункті буде розглянуто приклад застосування цієї системи в практичних сценаріях.

2.5. Практичне значення та область застосування

Розроблена програмна система має практичну цінність у контексті навчального процесу, самостійного вивчення алгоритмів, демонстрації їх роботи, а також як приклад інтеграції теоретичних знань у реальне застосування. Її головне призначення — надати користувачеві можливість візуально спостерігати за процесом пошуку шляху різними алгоритмами в інтерактивному середовищі.

У першу чергу система орієнтована на використання в освітньому процесі. Вона може бути корисною для викладання таких тем, як «графи», «алгоритми пошуку в ширину та глибину», «евристичні методи», «оптимальні маршрути» тощо. У навчальних закладах програму можна використовувати як ілюстративний матеріал під час лекцій або лабораторних робіт, що підвищує наочність і ефективність засвоєння теоретичного матеріалу.

Програма також буде корисною студентам і школярам при самостійному вивченні алгоритмів. Вона дозволяє не лише побачити, як працює той чи інший метод, але й порівняти результати (час виконання, ефективність маршруту) між алгоритмами. Таким чином, користувач може на практиці переконатися у відмінностях між DFS і BFS, перевагах евристики в A*, чи точності алгоритму Дейкстри.

Ще однією сферою застосування роботи, у яких необхідно проілюструвати чи проаналізувати роботу алгоритмів у графах. Система може бути використана як база для подальших досліджень, експериментів або розширень — наприклад, додавання нових алгоритмів, ускладнення лабіринтів, введення ваг на ребрах тощо.

У професійній діяльності (зокрема, в ігровій індустрії або розробці програмного забезпечення для логістики чи робототехніки) реалізація алгоритмів пошуку шляху є основою багатьох функціональних систем. Представлена розробка є прикладом базової системи навігації, яку можна масштабувати та адаптувати до конкретних задач. Зокрема, модульна структура

дозволяє швидко замінити або удосконалити один з компонентів, не змінюючи решту системи.

З практичної точки зору програму можна використовувати як демонстраційний матеріал під час хакатонів, STEM-івентів, конкурсів інформатики або олімпіад, де важливо не лише реалізувати алгоритм, але й представити його в наочному вигляді. Завдяки простоті інтерфейсу та легкості запуску, програму можна використовувати без спеціальної підготовки навіть на стандартному комп’ютерному обладнанні.

Варто також відзначити, що система може бути адаптована для інтеграції в онлайн-платформи навчання, наприклад, у вигляді веб-додатку чи модуля до системи дистанційного навчання (LMS). При незначному доопрацюванні можливо створити багатокористувацький режим або режим тестування, де користувач обирає алгоритм і перевіряє знання на практиці.

У перспективі розроблена система може стати базою для побудови повноцінного навчального середовища з аналізом помилок, рекомендаціями з вибору оптимального алгоритму та підтримкою гнучкого конфігурування середовища під потреби користувача.

Таким чином, практична значущість розробленої системи полягає в її універсальності, доступності та гнучкості для широкого кола користувачів: від школярів і студентів до викладачів, розробників та дослідників. Програма ефективно поєднує теоретичну основу з прикладним використанням, що дозволяє застосовувати її у навчанні, дослідженнях і як базову платформу для подальшої розробки більш складних інтелектуальних систем навігації.

2.6. Побудова сценаріїв використання

Для даної системи візуалізації алгоритмів пошуку шляху було визначено кілька основних сценаріїв, що охоплюють ключові функціональні можливості програми.

Сценарій 1: Побудова лабіринту з шаблону

Мета: Створити лабіринт за допомогою одного з вбудованих шаблонів.

Попередні умови: Програма запущена, головне вікно завантажене.

Основний сценарій:

1. Користувач відкриває головне вікно програми.
2. У випадаючому списку «Лабіринт» обирає один із шаблонів: «Лабіринт 1», «Без виходу», «Демо».
3. Натискає кнопку «Створити лабіринт».
4. Програма завантажує відповідний масив лабіринту та візуалізує його на панелі.
5. Лабіринт відображається у вигляді кольорової сітки.

Результат: Статичний лабіринт готовий до виконання пошуку.

Сценарій 2: Генерація випадкового лабіринту

Мета: Створити унікальний лабіринт випадковим чином.

Основний сценарій:

1. Користувач обирає опцію «Випадковий» у випадаючому списку лабіринтів.
2. Натискає кнопку створення лабіринту.
3. Вбудований генератор створює зв'язну сітку з початковою та кінцевою точкою.
4. Лабіринт виводиться на екран у графічному вигляді.

Результат: Лабіринт з унікальною конфігурацією готовий до обробки.

Сценарій 3: Запуск алгоритму пошуку

Мета: Побачити візуалізацію процесу пошуку шляху від старту до цілі.

Основний сценарій:

1. Користувач обирає алгоритм у випадаючому списку: BFS, DFS, A*, Dijkstra.
2. Переконується, що лабіринт уже створений або згенерований.
3. Натискає кнопку «Створити бігуна».
4. Програма виконує обраний алгоритм, викликаючи функції візуалізації клітинок.
5. Після завершення шлях відображається синім кольором.

Результат: Користувач бачить покрокову візуалізацію обраного алгоритму.

Сценарій 4: Перегляд результатів виконання

Мета: Ознайомитись зі статистикою після виконання алгоритму.

Основний сценарій:

1. Після виконання пошуку дані автоматично додаються до таблиці результатів.
2. Користувач бачить час виконання, тип лабіринту, назву алгоритму та результат (успіх/невдача).

Результат: Створюється журнал виконання, який можна переглядати без втрати попередніх даних.

Сценарій 5: Отримання інформації про програму

Мета: Дізнатися базові відомості про програму.

Основний сценарій:

1. Користувач натискає кнопку «Про розробника».
2. З’являється вікно з інформацією про автора, версію та рік створення програми.

Результат: Користувач ознайомлений із мета-інформацією про застосунок.

Сценарій 6: Спроба створити лабіринт без вибору алгоритму

Мета: Перевірити реакцію системи при відсутності вибраного алгоритму.

Основний сценарій:

1. Користувач відкриває програму.
2. Обирає тип лабіринту зі списку.
3. Натискає кнопку «Створити бігуна», не обравши алгоритм у списку.
4. Система перевіряє вибір і видає повідомлення про помилку: «Виберіть алгоритм і лабіринт!».

Результат: Програма не починає візуалізацію й запобігає некоректним діям.

Сценарій 7: Спроба запуску без вибору лабіринту

Мета: Побачити, як система реагує, коли обрано алгоритм, але не задано лабіринт.

Основний сценарій:

1. Користувач відкриває програму.
2. Обирає алгоритм у списку (наприклад, A*).
3. Не обирає тип лабіринту.
4. Натискає кнопку запуску алгоритму.
5. Програма виводить повідомлення про необхідність вибору лабіринту.

Результат: Користувач отримує підказку, що обидва параметри мають бути обрані перед запуском.

Сценарій 8: Послідовний запуск різних алгоритмів у різних лабіринтах

Мета: Провести кілька тестів із різними алгоритмами та лабіринтами для наповнення таблиці результатів.

Основний сценарій:

1. Користувач запускає програму.
2. Обирає шаблон «Лабіринт 1» і алгоритм BFS.
3. Натискає кнопку запуску — результат записується в таблицю.
4. Змінює вибір лабіринту на «Без виходу» і запускає DFS.
5. Обирає «Демо» і запускає A*.
6. Обирає «Випадковий» і запускає Dijkstra.
7. Після кожного запуску новий рядок з’являється в таблиці статистики.

Результат: Таблиця результатів містить повний журнал запусків для різних комбінацій алгоритмів і лабіринтів. Це дає змогу порівняти ефективність пошуку за різних умов.

Визначення сценаріїв використання дозволяє зрозуміти логіку дій користувача, забезпечити покриття основних функціональних вимог і виявити потенційні точки вдосконалення. Усі сценарії реалізовані відповідно до принципів зручності та послідовності, що підвищує загальну якість взаємодії користувача із системою.

2.7. Висновки до другого розділу.

У межах розділу 2 було здійснено комплексне обґрунтування вибору підходів, інструментів і рішень, необхідних для реалізації системи візуалізації алгоритмів пошуку шляху. Було сформульовано задачу з урахуванням вимог до точності, ефективності та наочності, а також детально розглянуто вибрані алгоритми (BFS, DFS, A*, Дейкстри) і пояснено причини їх включення до проєкту. Архітектура програмного забезпечення побудована за принципами модульності та розділення відповідальності, що забезпечує її гнучкість, масштабованість і зручність у підтримці.

Також проведено обґрунтування вибору технологій: середовище Visual Studio, мова програмування C++/CLI та платформа Windows Forms були обрані через оптимальне поєднання доступності, зручності реалізації GUI та підтримки об'єктно-орієнтованого підходу. Окрему увагу приділено практичній цінності системи, яка проявляється у можливості використання в освіті, дослідженнях, самоосвіті, а також як основи для більш складних навігаційних рішень.[13]

Побудовані сценарії використання дозволили охопити усі основні можливості програми та продемонструвати логіку взаємодії користувача з системою. Усі підрозділи розділу 2 є взаємопов’язаними, що дозволяє вважати проектне рішення цілісним, логічно обґрунтованим і придатним для практичної реалізації в обраній предметній області. Таким чином, сформоване програмне рішення відповідає вимогам до систем візуалізації алгоритмів та має перспективи для подальшого вдосконалення.

У подальшому можливе розширення функціоналу системи: зокрема, додавання налаштувань генератора, підтримка вагових графів або створення веб-версії інтерфейсу. Це відкриває нові напрями для розвитку розробки в освітньому та науковому контексті.

РОЗДІЛ 3. Реалізація та тестування

3.1. Реалізація алгоритмів пошуку шляху

У програмі реалізовано чотири алгоритми: пошук у ширину (BFS), пошук у глибину (DFS), алгоритм Дейкстри та алгоритм A*. Кожен з них має власні особливості реалізації та відповідає за виконання логічного пошуку у межах заданого лабіринту. Для всіх алгоритмів забезпечено візуалізацію процесу, що дозволяє користувачу бачити проміжні кроки та результат.

Алгоритми реалізовано як окремі методи у класі CubeMovement, кожен із яких приймає двовимірний масив символів, що представляє лабіринт, а також функцію зворотного виклику для оновлення візуального представлення. Під час виконання алгоритм змінює значення клітинок і викликає візуалізацію через переданий callback. Результатом роботи кожного алгоритму є список координат (об’єктів типу Position), що складають знайдений шлях.

Реалізація BFS базується на використанні черги (Queue). Спочатку алгоритм додає стартову клітинку до черги, після чого поступово обходить суміжні клітинки, перевіряючи їх на прохідність. Якщо знаходиться кінцева точка, алгоритм припиняє виконання й починає відновлення шляху за допомогою таблиці попередників.

Кожна відвідана клітинка позначається символом *, а її координати передаються у функцію оновлення кольору. Візуально це виглядає як поступове заповнення поля світло-блакитними клітинками.

Алгоритм DFS реалізовано з використанням стеку (Stack). Його логіка полягає в дослідженні одного шляху наскрізь, поки не буде досягнуто глухого кута або кінцевої точки. У випадку глухого кута відбувається відкат назад.

У реалізації також ведеться таблиця батьків для подальшого відновлення шляху. Всі відвідані клітинки маркуються аналогічно BFS. Через специфіку алгоритму можна спостерігати глибокі проходи в одному напрямку з подальшим поверненням.

Для реалізації алгоритму Дейкстри використано чергу з пріоритетом, яка впорядковує клітинки за відстанню від старту. Кожній клітинці призначається вага (всі однакові — 1), і зберігається таблиця мінімальних відстаней.

У процесі роботи клітинки обробляються в порядку зростання вартості. Завдяки цьому шлях, який знайде алгоритм, буде найкоротшим за кількістю кроків. Результат — плавне розширення зони досяжності навколо старту, подібне до кругових хвиль.

A* поєднує принципи Дейкстри з евристикою. У реалізації використовується функція оцінки — відстань за Манхеттеном від поточної клітинки до цілі. Ця оцінка додається до вже пройденої відстані, і результат використовується для впорядкування черги.

A* працює ефективніше за Дейкстру, оскільки концентрується в напрямку цілі. У реалізації клітинки вибираються із черги за мінімальним значенням функції f = g + h. Відвідані клітинки оновлюються візуально в тому ж стилі, що й в інших алгоритмах.

У кожному алгоритмі, після досягнення цільової точки, використовується таблиця батьків для відновлення повного шляху. Зворотне проходження виконується від кінцевої клітинки до стартової, після чого координати додаються до списку шляху. Клітинки шляху відображаються синім кольором.

Кожен крок алгоритму викликає UpdateCellColor, що змінює колір відвіданої клітинки. Між викликами вставляється затримка, яка дозволяє користувачу бачити процес у динаміці. По завершенню, весь знайдений шлях виводиться поступово.

3.2. Реалізація графічного інтерфейсу користувача

Графічний інтерфейс користувача (GUI) розроблений на базі Windows Forms у середовищі Visual Studio з використанням мови програмування C++/CLI. GUI реалізовано у вигляді форми з кількома графічними елементами, які розміщені відповідно до логіки використання. Всі елементи ініціалізуються в методі InitializeComponent().

Основні елементи інтерфейсу:

1. Панель візуалізації (panel1) — призначена для відображення сітки лабіринту. Кожна клітинка генерується як об’єкт типу Label, який додається на панель. Колір клітинки змінюється залежно від її типу: стіна, шлях, старт, фініш, відвідана клітинка тощо.
2. Комбіновані списки (comboBox1, comboBox2) — дозволяють користувачу обрати варіант лабіринту та тип алгоритму відповідно. До списків додаються значення: «Лабіринт 1», «Без виходу», «Демо», «Випадковий» — для лабіринтів, та «BFS», «DFS», «A*», «Dijkstra» — для алгоритмів.
3. Кнопки (button1, button3, buttonInfo) — відповідають за створення лабіринту, запуск пошуку та відкриття інформації про програму. Всі кнопки мають зрозумілі назви та стилізовані відповідно до системної теми.
4. Таблиця результатів (dataGridView1) — використовується для відображення результатів запуску алгоритмів: назва лабіринту, алгоритм, час виконання та успішність проходження.
5. Інформаційне вікно — активується кнопкою «Про розробника» і відображає базову інформацію про програму.

Загальний вигляд графічного інтерфейсу показаний на Рис. 3.1.

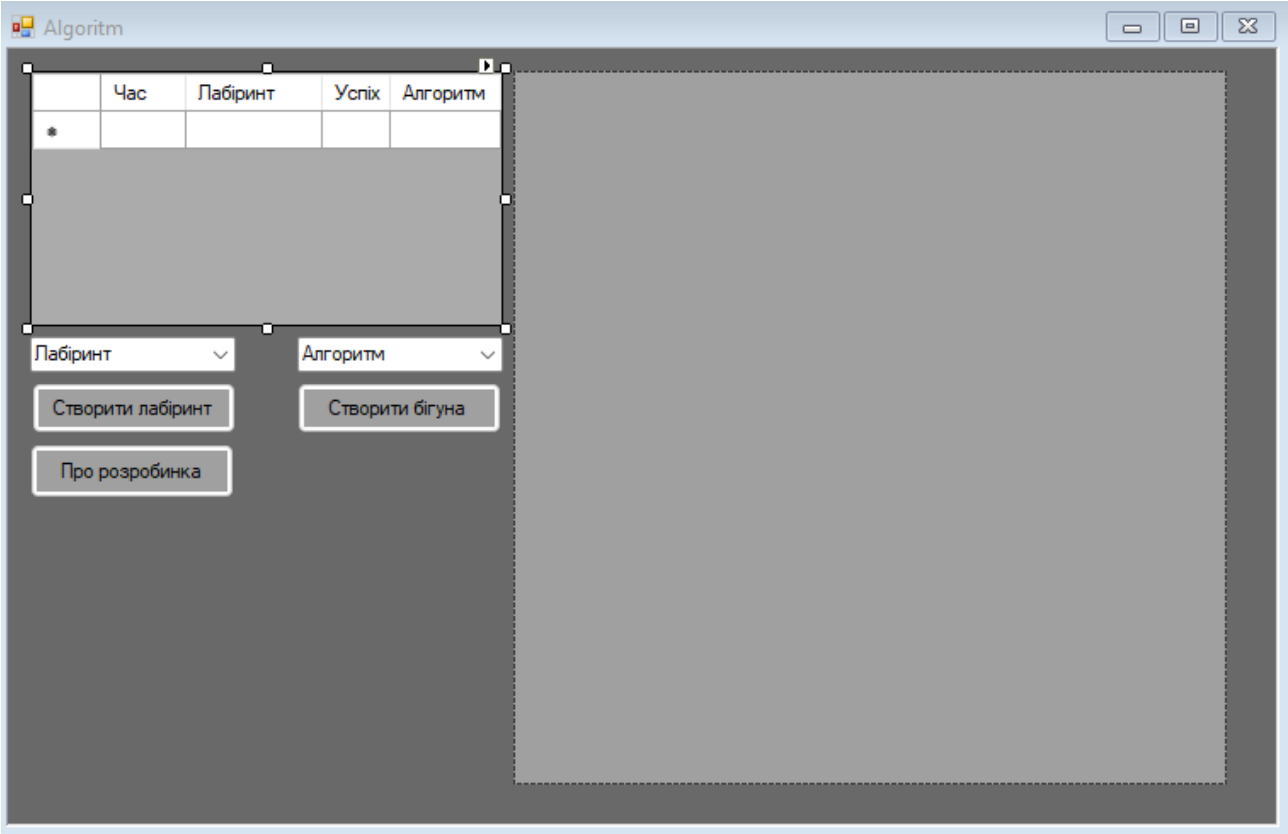

Рис. 3.1. Загальний вигляд графічного інтерфейсу програми

Кожна кнопка пов’язана з відповідним обробником подій:

- button3_Click() — створення лабіринту;
- button1_Click() — запуск пошуку з візуалізацією;
- buttonInfo_Click() — виведення вікна з інформацією.

Обробники містять логіку перевірки вибору, виклику генерації або завантаження лабіринту, запуску алгоритмів, а також взаємодії з графічним відображенням.

Після вибору типу лабіринту користувач натискає кнопку створення, що ініціює побудову сітки. Метод DisplayMaze() очищає попередній вміст панелі та формує нову сітку з Label-елементів відповідно до структури масиву maze. Розмір клітинки зафіксовано (20x20 пікселів), а координати визначаються з урахуванням позиції в масиві.

Додатково для кожної клітинки перед її створенням у циклі перевіряється її логічне призначення, що дозволяє одразу встановити потрібний колір і

уникнути зайвих обчислень у подальшому. Всі елементи додаються до панелі за допомогою функції panel1->Controls->Add(), що забезпечує динамічність і повний контроль над процесом формування лабіринту.

Після запуску алгоритму відбувається візуальне оновлення клітинок у реальному часі: змінюється колір відповідно до відвіданих позицій. Наприкінці синім кольором виводиться знайдений шлях. Час виконання та результати записуються в DataGridView, що дозволяє користувачу аналізувати ефективність різних алгоритмів.

Інтерфейс розроблено з урахуванням простоти розширення: можна легко додати нові типи лабіринтів або алгоритмів, не змінюючи основну структуру форми. Компоненти форми мають стабільні розміри та не залежать від зовнішнього середовища.

Реалізація графічного інтерфейсу користувача в системі забезпечує інтуїтивно зрозумілу, стабільну та візуально привабливу взаємодію з функціональністю програми. Використання Windows Forms дозволило швидко та ефективно побудувати необхідні компоненти, що є оптимальним рішенням у навчальному програмному проєкті. Програмний код графічного інтерфейсу наведено в ДОДАТКУ А, що дає змогу ознайомитися зі структурою та логікою побудови візуальних елементів.

3.3. Реалізація алгоритму побудови лабіринтів

Для підвищення варіативності та навчальної цінності програмної системи, окрім використання готових шаблонів лабіринтів, реалізовано механізм генерації випадкових лабіринтів. Це дозволяє кожного разу створювати унікальну структуру з обов’язковим зв’язком між початковою та кінцевою точкою. Генерація здійснюється за допомогою рекурсивного алгоритму прорізання проходів, адаптованого під прямокутну сітку.

Основою генератора слугує алгоритм, що нагадує модифікований DFS (пошук у глибину) з поверненням назад. Він гарантує створення єдиного з’єднаного шляху, а також заповнення простору лабіринту відповідно до правил прохідності. Результатом є зв’язна карта, де кожна відкрита клітинка потенційно досяжна зі старту.

Алгоритм реалізовано у вигляді методу GenerateRandomMaze(int size) у спеціальному класі MazeRandomizer. Метод створює двовимірний масив символів, розміру size x size, де всі клітинки ініціалізуються як стіни (x). Після цього запускається рекурсивна функція CarveMaze(int x, int y), яка проходить по масиву й вирізає шляхи.

Для кожної клітинки перевіряються потенційні напрямки (вгору, вниз, вліво, вправо), які перемішуються у випадковому порядку для забезпечення унікальності конфігурації. Якщо обрана клітинка та її сусіди відповідають умовам (не вийдуть за межі, ще не оброблені), то прорізається прохід.

Після завершення генерації зліва вгорі встановлюється стартова позиція (#), а праворуч знизу — вихід (z). Готовий масив повертається у вигляді об'єкта, що потім передається в інтерфейс для візуалізації.

Випадковість генерації забезпечується використанням генератора псевдовипадкових чисел (std::mt19937), що змінює порядок перевірки напрямків у кожному виклику.

Генерація випадкового лабіринту викликається після вибору опції «Випадковий» у списку типів лабіринтів. При натисканні кнопки «Створити

лабіринт» форма визначає, що слід запустити метод GenerateRandomMaze, і передає результат до візуалізатора (DisplayMaze). Користувач отримує нову структуру лабіринту, не повторюючи попередні спроби.

Приклад випадково згенерованого лабіринту показаний на Рис. 3.2.

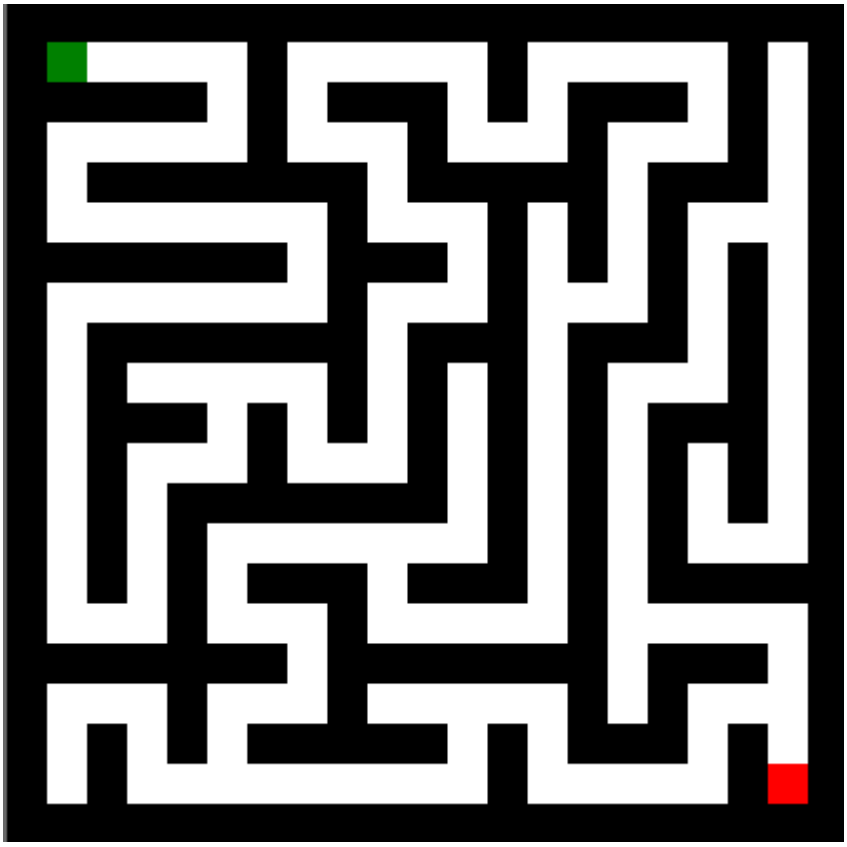

Рис 3.2. Приклад випадково згенерованого лабіринту

Лабіринти можуть мати різну складність і довжину шляху в залежності від випадковості, однак завжди мають логічно побудовану топологію. Це робить систему цікавою для багаторазового використання та тестування алгоритмів пошуку. Програмний код алгоритму генерації лабіринту наведено в ДОДАТКУ Б, що дозволяє детальніше ознайомитися з принципами його реалізації.

3.4. Тестові приклади для перевірки роботи алгоритмів

Для повноцінного тестування реалізованих алгоритмів пошуку шляху було проведено 16 запусків: по одному для кожної комбінації 4 алгоритмів і 4 лабіринтів, включаючи дві різні спроби з випадковими лабіринтами. Усі спостереження були зафіксовані вручну та перевірені за допомогою таблиці результатів, яка автоматично формується в інтерфейсі програми (елемент DataGridView).

1. Тест BFS на лабіринті «Демо»

Алгоритм швидко проходить простір, що не містить перешкод. Візуалізація розгортається у вигляді хвилі, яка рівномірно заповнює поле. Шлях знайдено за 37.9 с. Кількість клітинок мінімальна, маршрут — прямий.

2. Тест BFS на «Лабіринті 1»

BFS послідовно заповнює доступні ділянки, рухаючись від старту до фінішу найкоротшим маршрутом. Шлях знайдено, час виконання — 16 с. Видно характерне «розширення фронту» по лабіринту.

3. Тест BFS на «Без виходу»

Алгоритм послідовно досліджує всю доступну область, але не може знайти шлях. Після повного обходу видає результат: "Ні". Програма не зависає, повідомлення коректне. Час виконання — 20.4 с.

4. Тест BFS на «Випадковому» лабіринті

Створений лабіринт майже не має суттєвих розгалужень, перше довге розгалуження починається біля кінця лабіринту. BFS швидко знайшов шлях, тому що, шлях був майже прямим. Шлях знайдено, в таблиці результатів — "Так", час 7.8 с.

Створений лабіринт під час другої спроби проходив майже через все поле, ближче до центру шляху появилась велика кількість розгалужень, тому було витрачено більше часу. Шлях знайдено, в таблиці результатів — “Так”, час 16.5 с.

Результати проходження лабіринтів за алгоритмом BFS можна переглянути в Таблиці 3.1.

Час	Лабіринт	Успіх	Алгорит
37,9 с	Демо	Так	BFS
16,0 с	Лабіринт 1	Так	BFS
20,4 с	Без виходу	Ні	BFS
7,8 с	Випадковий	Так	BFS
16,5 с	Випадковий	Так	BFS

Таблиця 3.1.

5. Тест DFS на «Демо»

Шлях знайдено, у візуалізації видно «глибоке занурення» в один напрямок. В таблиці результатів — "Так", час 3.8 с.

6. Тест DFS на «Лабіринті 1»

DFS поводиться менш передбачувано. Алгоритм обирає один із проходів і досліджує його до кінця. Шлях знайдено, однак він довший і не оптимальний. Час — 7.7 с.

7. Тест DFS на «Без виходу»

DFS рухається у випадковому напрямку, поки не досягає тупика, потім повертається назад. У підсумку — шлях не знайдено. Таблиця результатів оновлюється коректно, програма не зависає. Час — 20.9 с.

8. Тест DFS на «Випадковому» лабіринті

Генератор створив складний лабіринт. DFS швидко проходив у глибину, відкатів не було і алгоритм відразу знайшов шлях. Вивід у таблицю — "Так", час — 14.4 с.

Під час другої спроби був створений довгий лабіринт з невеликою кількістю коротких розгалужень, алгоритм зробив декілька відкатів. Шлях знайдено, в таблиці результатів — “Так”, час 17.9 с.

Результати проходження лабіринтів за алгоритмом DFS можна переглянути в Таблиці 3.2.

Час	Лабіринт	Успіх	Алгорит
3,8 с	Демо	Так	DFS
7,7 с	Лабіринт 1	Так	DFS
20,9 с	Без виходу	Ні	DFS
14,4 с	Випадковий	Так	DFS
17,9 с	Випадковий	Так	DFS

Таблиця 3.2.

9. Тест A* на «Демо»

A* довго шукав шлях, але пошук шов по діагоналі до точки кінця. Час — 61.5 с. Результат — "Так".

10. Тест A* на «Лабіринті 1»

Шлях знайдено ефективно, з мінімальними відхиленнями. A* обирає найкращі напрямки, що видно по візуалізації — заповнення спрямоване до цілі. Час — 8.2 с. Результат — "Так".

11. Тест A* на «Без виходу»

Алгоритм обчислює напрямки, однак, не маючи змоги дістатись до цілі, він продовжує пошук шляху, пройшовши весь лабіринт, алгоритм зупинився. Результат у таблиці — "Ні". Поведінка стабільна, затримки відсутні. Час — 21.5 с.

12. Тест A* на «Випадковому» лабіринті

A* впевнено рухався в бік виходу, обираючи ефективні напрямки. Шлях знайдено, час — 17.2 с. Анімація плавна.

Під час другої спроби алгоритм рухався також в бік виходу ігноруючи розгалуження які віддалялись від кінця лабіринту. Шлях знайдено, час — 11.1 с.

Результати проходження лабіринтів за алгоритмом A* можна переглянути в Таблиці 3.3.

Час	Лабіринт	Успіх	Алгорит
61,5 с	Демо	Так	A*
8,2 с	Лабіринт 1	Так	A*
21,5 с	Без виходу	Ні	A*
17,2 с	Випадковий	Так	A*
11,1 с	Випадковий	Так	A*

Таблиця 3.3.

13. Тест Dijkstra на «Демо»

Результат аналогічний BFS: алгоритм знаходить шлях, але витратив трохи менше часу на обчислення. Шлях знайдено, час — 35.7 с.

14. Тест Dijkstra на «Лабіринті 1»

Поступово заповнює простір, створюючи ефект розширення по радіусу. Знаходить оптимальний шлях. У таблиці — "Так", час — 17.2 с.

15. Тест Dijkstra на «Без виходу»

Алгоритм коректно завершує пошук після вичерпання всіх варіантів. Повідомляє про відсутність шляху, що підтверджено таблицею. Час — 21.3 с.

16. Тест Dijkstra на «Випадковому» лабіринті

Створився легкий лабіринт з невеликою кількістю розгалужень. Dijkstra стабільно знаходить шлях. Результат — "Так", час — 13.1 с.

Під час другої спроби було створено також легкий лабіринт тому шлях було знайдену дуже швидко. Результат — "Так", час — 10.4 с.

Результати проходження лабіринтів за алгоритмом Dijkstra можна переглянути в Табляці 3.4.

Час	Лабіринт	Успіх	Алгорит
35,7 с	Демо	Так	Dijkstra
17,2 с	Лабіринт 1	Так	Dijkstra
21,3 с	Без виходу	Ні	Dijkstra
13,1 с	Випадковий	Так	Dijkstra
10,4 с	Випадковий	Так	Dijkstra

Таблиця 3.4.

Усі алгоритми працюють коректно як у простих, так і в складних лабіринтах. BFS та Dijkstra демонструють стабільне розширення області пошуку, при цьому BFS зазвичай швидший. DFS показує хаотичний обхід, але успішно знаходить шлях, якщо він існує. A* виявився найефективнішим на складних картах, хоча іноді спостерігалося уповільнення. У випадках з відсутнім виходом усі алгоритми завершували роботу правильно, що підтверджує надійність реалізації. Таким чином, система виконує поставлені завдання повною мірою, а поведінка алгоритмів відповідає їх теоретичним властивостям.

3.5. Інструкція користувача

Дана інструкція призначена для користувачів програмної системи візуалізації алгоритмів пошуку шляху в лабіринті. Вона описує основні кроки для роботи з програмою, принципи використання елементів інтерфейсу та рекомендації для ефективного застосування.

Вимоги до системи

- Операційна система: Windows 10 або новіша
- Наявність .NET Framework (версія 4.5 або вище)
- Мінімальна роздільна здатність екрану: 1024x768
- Microsoft Visual C++ 2015-2022

Запуск програми

1. Відкрийте файл виконання (EXE), що знаходиться у папці з проєктом.
2. Очікуйте завантаження головного вікна програми.

Після запуску відкривається головне вікно, яке містить наступні елементи:

- Панель візуалізації — сітка, де відображається лабіринт і хід алгоритму.
- Список вибору лабіринту (зліва) — оберіть варіант лабіринту: «Лабіринт 1», «Без виходу», «Демо», або «Випадковий».
- Список вибору алгоритму (справа) — оберіть метод пошуку: BFS, DFS, A*, Dijkstra.
- Кнопка "Створити лабіринт" — генерує або завантажує вибраний лабіринт.
- Кнопка "Створити бігуна" — запускає візуалізацію алгоритму.
- Таблиця результатів — фіксує час, тип алгоритму, лабіринт та успішність проходження.
- Кнопка "Про розробника" — виводить інформацію про автора програми.

Послідовність дій користувача

1. Оберіть потрібний тип лабіринту зі списку зліва.
2. Натисніть кнопку "Створити лабіринт".
3. Дочекайтесь, поки лабіринт буде виведено на панелі.
4. Оберіть алгоритм пошуку зі списку справа.
5. Натисніть кнопку "Створити бігуна".
6. Спостерігайте за візуалізацією роботи алгоритму в реальному часі.
7. Результат (успіх/неуспіх, час, назви) буде записаний до таблиці в нижній частині вікна.

Додаткові поради

- Якщо алгоритм не запускається, переконайтесь, що обрано і лабіринт, і алгоритм.
- При виборі опції «Без виходу» алгоритм не зможе знайти шлях — це очікуваний результат.
- Якщо потрібно очистити історію запусків — закрийте та відкрийте програму знову.
- Повторне натискання на ті ж самі кнопки з однаковими параметрами дозволяє повторити експеримент.

Приклад

1. Виберіть «Випадковий» лабіринт.
2. Натисніть "Створити лабіринт".
3. Після побудови сітки — виберіть алгоритм A*.
4. Натисніть "Створити бігуна".
5. Дивіться, як алгоритм прокладає шлях до виходу. Результат з’явиться в таблиці.

Висновок

У результаті виконання кваліфікаційної роботи було розроблено програмну систему візуалізації алгоритмів пошуку шляху в лабіринті, яка дозволяє наочно демонструвати роботу чотирьох популярних методів: пошуку в ширину (BFS), пошуку в глибину (DFS), алгоритму Дейкстри та алгоритму A*. Усі алгоритми реалізовано у вигляді незалежних модулів з можливістю інтерактивної взаємодії з користувачем через графічний інтерфейс.

На етапі теоретичного дослідження було проаналізовано основи візуалізації, класифіковано алгоритми пошуку шляху, розглянуто існуючі рішення та обґрунтовано доцільність створення власної системи. Особливу увагу приділено моделюванню задачі у вигляді графа, що дало змогу формально визначити структуру лабіринту, шляху та критерії ефективності обчислень.

У ході проектування системи було обрано сучасні засоби розробки: середовище Visual Studio, мову C++/CLI та бібліотеку Windows Forms. Побудовано архітектуру за принципом модульності, що включає окремі компоненти для візуалізації, логіки алгоритмів, генерації лабіринтів і взаємодії з користувачем. Такий підхід дозволив досягти високої наочності, стабільності роботи та простоти в обслуговуванні.

Було проведено серію тестувань для кожного алгоритму на кількох варіантах лабіринтів, у тому числі випадкових. Результати перевірки підтвердили правильність реалізації, коректну візуалізацію та ефективність використаних підходів. Крім того, система дозволяє зручно порівнювати час виконання та поведінку алгоритмів у різних умовах.

Розроблену програму можна активно застосовувати у навчальному процесі для демонстрації алгоритмів пошуку, аналізу ефективності рішень та практичного засвоєння графових структур. Також вона може бути використана як основа для подальших розробок або вдосконалень, зокрема — додавання

нових методів, розширення генератора лабіринтів або створення веб-версії застосунку.

Програмна система демонструє важливість поєднання теоретичних знань з практичним досвідом. Завдяки інтуїтивно зрозумілому інтерфейсу та візуальній анімації, користувачі різного рівня підготовки можуть швидко ознайомитися з принципами роботи алгоритмів та їх особливостями. Це дозволяє покращити якість навчання та зробити складні теми більш доступними.

Окрему увагу заслуговує гнучкість реалізованої архітектури, яка дозволяє адаптувати систему до нових вимог. За потреби користувач може додати інші типи алгоритмів або змінити правила побудови лабіринтів, не порушуючи загальну структуру програми. Це робить систему придатною для використання як базової платформи в освітніх та наукових проєктах.

Успішна реалізація цього проєкту підтверджує доцільність використання C++/CLI у навчальних системах, орієнтованих на графіку. Поєднання високої швидкодії мови програмування з простим створенням GUI у середовищі Visual Studio забезпечує ефективність розробки. Такий підхід може бути рекомендований для реалізації аналогічних навчальних систем у майбутньому.

Узагальнюючи, дана кваліфікаційна робота поєднала теоретичні знання в галузі алгоритмів, графів та інтерфейсного програмування з практичною реалізацією, що в повній мірі відповідає поставленій меті та підтверджує досягнення очікуваних результатів.

Список використаних джерел

1. Кормен Т. Х., Лейзерсон Ч. Е., Рівест Р. Л., Штайн К. Алгоритми: побудова та аналіз. — Київ: Вільямс, 2020.
2. Алгоритми українською — Algoua. URL: https://algoua.com/ — Дата звернення: 03.06.2025.
3. Алгоритм Дейкстри. Теорія. URL: http://choippo.cn.sch.in.ua/Files/downloadcenter/Алгоритм%20Дейкстри.%20Теорія.pdf — Дата звернення: 03.06.2025.
4. Clement Mihailescu. Pathfinding Visualizer. URL: https://clementmihailescu.github.io/Pathfinding-Visualizer/ — Дата звернення: 03.06.2025.
5. VisuAlgo.net. Visualisation of graph algorithms. URL: https://visualgo.net/en/ — Дата звернення: 03.06.2025.
6. MazeSolver — візуалізація пошуку шляху. URL: https://esstudio.site/maze-solver/ — Дата звернення: 03.06.2025.
7. AlgoVis.io — алгоритми пошуку шляху. URL: https://tobinatore.github.io/algovis/index.html — Дата звернення: 03.06.2025.
8. Algorithm Visualizer. URL: https://algorithm-visualizer.org — Дата звернення: 03.06.2025.
9. Universal implementation of BFS, DFS, Dijkstra and A*. CodeProject. URL: https://www.codeproject.com/Articles/5368418/Universal-implementation-of-BFS-DFS-Dijkstra-and-2 — Дата звернення: 03.06.2025.
10. Microsoft Docs. Windows Forms documentation. URL: https://learn.microsoft.com/en-us/dotnet/desktop/winforms — Дата звернення: 03.06.2025.
11. Qt Documentation. Офіційна документація середовища Qt. URL: https://doc.qt.io — Дата звернення: 03.06.2025.
12. Microsoft Docs. Developing Desktop Apps with .NET. URL: https://learn.microsoft.com/en-us/dotnet/desktop/ — Дата звернення: 03.06.2025.

13. MSDN Library. C++/CLI Language Reference. URL: https://learn.microsoft.com/en-us/cpp/dotnet/ — Дата звернення: 03.06.2025.

14. Wirth N. Algorithms and Data Structures. — Prentice Hall, 1985.

15. Stack Overflow. Приклади реалізації алгоритмів пошуку шляху на C++. URL: https://stackoverflow.com/questions/18570585/ — Дата звернення: 03.06.2025.

16. Шилдт Г. C++: Повне керівництво. — Київ: Діалектика, 2021.

17. Stroustrup B. The C++ Programming Language. 4th ed. — Addison-Wesley, 2013. — 1376 p.

18. Microsoft Docs. Visual Studio IDE Documentation. URL: https://learn.microsoft.com/en-us/visualstudio/ide/ — Дата звернення: 03.06.2025.

19. Windows Forms .NET Overview. URL: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/ — Дата звернення: 03.06.2025.

Додаток А

Програмний код графічного інтерфейсу

```
#pragma once

#include <memory>
#include <vector>
#include <random>
#include "CubeMovement.h"
#include "MazePresets.h"
#include "MazeRandomizer.h"

namespace Project3 {

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;

public ref class MyForm : public System::Windows::Forms::Form
{
public:
MyForm(void)
{
InitializeComponent();
presetMazes = LoadPresetMazes();

comboBox1->Items->Add("Лабіринт 1");
comboBox1->Items->Add("Без виходу");
comboBox1->Items->Add("Демо");
comboBox1->Items->Add("Випадковий");

comboBox2->Items->Add("BFS");
comboBox2->Items->Add("DFS");
```

```
comboBox2->Items->Add("A*");
comboBox2->Items->Add("Dijkstra");

this->MaximizeBox = false;
this->MinimizeBox = false;
this->StartPosition = FormStartPosition::CenterScreen;
}

protected:
~MyForm()
{
if (components)
delete components;
}
private:
    System::ComponentModel::Container^ components;
    System::Windows::Forms::Panel^ panel1;
    System::Windows::Forms::ComboBox^ comboBox1;
    System::Windows::Forms::ComboBox^ comboBox2;
    System::Windows::Forms::Button^ button1;
    System::Windows::Forms::Button^ button3;

    System::Windows::Forms::Button^ buttonInfo;
    System::Windows::Forms::DataGridView^ dataGridView1;
    System::Windows::Forms::DataGridViewTextBoxColumn^ Col_Time;
    System::Windows::Forms::DataGridViewTextBoxColumn^ Col_Lab;
    System::Windows::Forms::DataGridViewTextBoxColumn^ Col_exit;
    System::Windows::Forms::DataGridViewTextBoxColumn^ Col_Alg;

    cli::array<cli::array<wchar_t>^>^ maze = gcnew
cli::array<cli::array<wchar_t>^>(21);
    List<cli::array<cli::array<wchar_t>^>^>^ presetMazes;

#pragma region Windows Form Designer generated code
void InitializeComponent(void)
{
this->panel1 = (gcnew System::Windows::Forms::Panel());
this->comboBox1 = (gcnew System::Windows::Forms::ComboBox());
this->comboBox2 = (gcnew System::Windows::Forms::ComboBox());
this->button1 = (gcnew System::Windows::Forms::Button());
this->button3 = (gcnew System::Windows::Forms::Button());
this->buttonInfo = (gcnew System::Windows::Forms::Button());
this->dataGridView1 = (gcnew System::Windows::Forms::DataGridView());
```

```
this->Col_Time = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());

this->Col_Lab = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());

this->Col_exit = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());

this->Col_Alg = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());

(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>dataGridView1))->BeginInit();

this->SuspendLayout();

//

// panel1

//

this->panel1->BackColor = System::Drawing::SystemColors::ControlDark;

this->panel1->Location = System::Drawing::Point(296, 12);

this->panel1->Name = L"panel1";

this->panel1->Size = System::Drawing::Size(419, 419);

this->panel1->TabIndex = 0;

//

// comboBox1

//

this->comboBox1->Location = System::Drawing::Point(12, 168);

this->comboBox1->Name = L"comboBox1";

this->comboBox1->Size = System::Drawing::Size(121, 21);

this->comboBox1->TabIndex = 3;

this->comboBox1->Text = L"Лабіринт";

this->comboBox1->SelectedIndexChanged += gcnew System::EventHandler(this,
&MyForm::comboBox1_SelectedIndexChanged);

//

// comboBox2

//

this->comboBox2->Location = System::Drawing::Point(169, 168);

this->comboBox2->Name = L"comboBox2";

this->comboBox2->Size = System::Drawing::Size(121, 21);

this->comboBox2->TabIndex = 2;

this->comboBox2->Text = L"Алгоритм";

//
```

```
// button1
//
this->button1->BackColor = System::Drawing::SystemColors::ButtonShadow;
this->button1->Location = System::Drawing::Point(169, 195);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(120, 30);
this->button1->TabIndex = 6;
this->button1->Text = L"Створити бігуна";
this->button1->UseVisualStyleBackColor = false;
this->button1->Click += gcnew System::EventHandler(this,
&MyForm::button1_Click);
//
// button3
//
this->button3->BackColor = System::Drawing::SystemColors::ButtonShadow;
this->button3->Location = System::Drawing::Point(13, 195);
this->button3->Name = L"button3";
this->button3->Size = System::Drawing::Size(120, 30);
this->button3->TabIndex = 5;
this->button3->Text = L"Створити лабіринт";
this->button3->UseVisualStyleBackColor = false;
this->button3->Click += gcnew System::EventHandler(this,
&MyForm::button3_Click);
//
// buttonInfo
//
this->buttonInfo->BackColor =
System::Drawing::SystemColors::ButtonShadow;
this->buttonInfo->Location = System::Drawing::Point(12, 231);
this->buttonInfo->Name = L"buttonInfo";
this->buttonInfo->Size = System::Drawing::Size(120, 32);
this->buttonInfo->TabIndex = 1;
this->buttonInfo->Text = L"Про розробинка";
this->buttonInfo->UseVisualStyleBackColor = false;
this->buttonInfo->Click += gcnew System::EventHandler(this,
&MyForm::buttonInfo_Click);
      //
        // dataGridView1
```

```
        //
        this->dataGridView1->Columns->AddRange(gcnew cli::array<
System::Windows::Forms::DataGridViewColumn^  >(4) {
            this->Col_Time,
                this->Col_Lab, this->Col_exit, this->Col_Alg
        });
        this->dataGridView1->Location = System::Drawing::Point(12, 12);
        this->dataGridView1->Name = L"dataGridView1";
        this->dataGridView1->ShowRowErrors = false;
        this->dataGridView1->Size = System::Drawing::Size(278, 150);
        this->dataGridView1->TabIndex = 4;
        this->dataGridView1->CellContentClick += gcnew
System::Windows::Forms::DataGridViewCellEventHandler(this,
&MyForm::dataGridView1_CellContentClick);
        //
        // Col_Time
        //
        this->Col_Time->HeaderText = L"Час";
        this->Col_Time->Name = L"Col_Time";
        this->Col_Time->Width = 50;
        //
        // Col_Lab
        //
        this->Col_Lab->HeaderText = L"Лабіринт";
        this->Col_Lab->Name = L"Col_Lab";
        this->Col_Lab->Width = 80;
        //
        // Col_exit
        //
        this->Col_exit->HeaderText = L"Успіх";
        this->Col_exit->Name = L"Col_exit";
        this->Col_exit->Width = 40;
        //
        // Col_Alg
        //
        this->Col_Alg->HeaderText = L"Алгоритм";
        this->Col_Alg->Name = L"Col_Alg";
        this->Col_Alg->Width = 65;
                //

// MyForm

//

this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Inherit;

this->BackColor = System::Drawing::SystemColors::ControlDarkDark;

this->ClientSize = System::Drawing::Size(744, 454);

this->Controls->Add(this->buttonInfo);

this->Controls->Add(this->comboBox2);

this->Controls->Add(this->comboBox1);

this->Controls->Add(this->dataGridView1);

this->Controls->Add(this->button3);

this->Controls->Add(this->button1);

this->Controls->Add(this->panel1);
```

```
this->FormBorderStyle = System::Windows::Forms::FormBorderStyle::Fixed3D;

this->Name = L"MyForm";

this->Text = L"Algoritm";

this->Load += gcnew System::EventHandler(this, &MyForm::MyForm_Load);

(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>dataGridView1))->EndInit();

this->ResumeLayout(false);


}
```

Додаток Б

Програмний код алгоритмів генерації лабіринту

```
using namespace System; using namespace System::Collections::Generic;

inline List<cli::array<cli::array^>^>^ LoadPresetMazes() { auto presetMazes =
gcnew List<cli::array<cli::array^>^>();

// ===== Лабіринт 1 =====
cli::array<cli::array<wchar_t>^>^ maze1 = gcnew cli::array<cli::array<wchar_t>^>(21);
maze1[0] = gcnew cli::array<wchar_t>{L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x',
L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x'};
maze1[1] = gcnew cli::array<wchar_t>{L'x', L'#', L'y', L'y', L'x', L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'y',
L'y', L'y', L'y', L'y', L'y', L'y', L'y', L'x'};
maze1[2] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'x', L'x', L'y', L'x', L'y', L'x',
L'x', L'x', L'x', L'x', L'x', L'x', L'y', L'x'};
maze1[3] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x',
L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'x'};
maze1[4] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x',
L'x', L'y', L'x', L'x', L'y', L'x', L'y', L'x'};
maze1[5] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'y',
L'y', L'y', L'y', L'x', L'y', L'x', L'y', L'x'};
maze1[6] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'y', L'x', L'y', L'y', L'y', L'x', L'x', L'x', L'x', L'x',
L'x', L'x', L'y', L'x', L'y', L'x', L'y', L'x'};
maze1[7] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'y', L'y', L'y', L'y', L'y',
L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x'};
maze1[8] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'x', L'x', L'y', L'x', L'x', L'x', L'x', L'x', L'x', L'x',
L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x'};
maze1[9] = gcnew cli::array<wchar_t>{L'x', L'y', L'y', L'y', L'y', L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'y',
L'y', L'x', L'y', L'x', L'y', L'x', L'y', L'x'};
maze1[10] = gcnew cli::array<wchar_t>{L'x', L'x', L'x', L'y', L'x', L'x', L'x', L'x', L'x', L'y', L'x', L'x', L'x',
L'x', L'x', L'y', L'x', L'y', L'x', L'y', L'x'};
maze1[11] = gcnew cli::array<wchar_t>{L'x', L'y', L'y', L'y', L'x', L'y', L'y', L'y', L'x', L'y', L'y', L'y', L'y',
L'y', L'y', L'y', L'x', L'y', L'x', L'y', L'x'};
maze1[12] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'x', L'x', L'x', L'x', L'y', L'x', L'x', L'x', L'x', L'x',
L'x', L'x', L'x', L'x', L'y', L'x', L'y', L'x'};
maze1[13] = gcnew cli::array<wchar_t>{L'x', L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'y', L'y', L'y', L'y', L'y',
L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'x'};
maze1[14] = gcnew cli::array<wchar_t>{L'x', L'x', L'x', L'y', L'x', L'y', L'x', L'y', L'x', L'x', L'x', L'x', L'x',
L'x', L'x', L'x', L'x', L'x', L'x', L'y', L'x'};
maze1[15] = gcnew cli::array<wchar_t>{L'x', L'y', L'y', L'y', L'x', L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'y',
L'y', L'y', L'y', L'y', L'y', L'y', L'y', L'x'};
maze1[16] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'y', L'x', L'x', L'x', L'x', L'x', L'y', L'x', L'y', L'x',
L'x', L'x', L'x', L'x', L'x', L'x', L'y', L'x'};
maze1[17] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'x', L'y', L'y',
L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'x'};
maze1[18] = gcnew cli::array<wchar_t>{L'x', L'y', L'x', L'x', L'x', L'x', L'x', L'y', L'x', L'y', L'x', L'x', L'x',
L'x', L'x', L'x', L'x', L'y', L'x', L'y', L'x'};
```

```
maze1[19] = gcnew cli::array<wchar_t>{L'x', L'y', L'y', L'y', L'y', L'y', L'y', L'y', L'x', L'y', L'y', L'y', L'y',
L'y', L'y', L'y', L'y', L'y', L'y', L'z', L'x'};
maze1[20] = gcnew cli::array<wchar_t>{L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x',
L'x', L'x', L'x', L'x', L'x', L'x', L'x', L'x'};
presetMazes->Add(maze1);


// ===== Лабіринт 2 (тупик) =====
cli::array<cli::array<wchar_t>^>^ maze2 = gcnew cli::array<cli::array<wchar_t>^>(21);
maze2[0] = gcnew cli::array<wchar_t>{ L'x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x' };
maze2[1] = gcnew cli::array<wchar_t>{ L'x','#','y','y','x','y','x','y','y','y','x','y','x','y','y','y','x','y','y','y','x' };
maze2[2] = gcnew cli::array<wchar_t>{ L'x','x','x','y','x','y','x','y','x','y','y','y','y','y','x','y','y','y','x','y','x' };
maze2[3] = gcnew cli::array<wchar_t>{ L'x','y','y','y','x','y','x','y','x','x','x','x','x','y','x','x','x','y','x','y','x' };
maze2[4] = gcnew cli::array<wchar_t>{ L'x','y','x','x','x','y','x','y','x','y','y','y','x','y','y','y','x','y','x','y','x' };
maze2[5] = gcnew cli::array<wchar_t>{ L'x','y','x','y','y','y','x','y','x','y','x','y','x','x','x','y','x','y','x','y','x' };
maze2[6] = gcnew cli::array<wchar_t>{ L'x','y','x','y','x','x','x','y','x','y','x','y','y','y','x','y','x','y','x','y','x' };
maze2[7] = gcnew cli::array<wchar_t>{ L'x','y','x','y','y','y','y','y','x','y','x','x','x','y','x','y','x','y','x','y','x' };
maze2[8] = gcnew cli::array<wchar_t>{ L'x','y','x','x','x','x','x','x','x','y','x','y','y','y','x','y','x','y','x','y','x' };
maze2[9] = gcnew cli::array<wchar_t>{ L'x','y','y','y','y','y','y','y','y','y','x','y','x','x','x','y','x','y','x','y','x' };
maze2[10] = gcnew cli::array<wchar_t>{ L'x','x','x','x','x','x','x','x','x','y','x','y','x','y','y','y','x','y','x','y','x' };
maze2[11] = gcnew cli::array<wchar_t>{ L'x','y','y','y','y','y','y','y','y','x','y','x','y','x','x','y','x','x','y','x','y','x' };
maze2[12] = gcnew cli::array<wchar_t>{ L'x','y','x','x','x','x','x','y','x','y','x','y','y','y','y','y','x','y','x','y','x' };
maze2[13] = gcnew cli::array<wchar_t>{ L'x','y','x','y','y','y','x','y','x','y','x','x','x','x','x','y','x','y','x','y','x' };
maze2[14] = gcnew cli::array<wchar_t>{ L'x','y','x','y','x','y','x','y','x','y','y','y','y','y','x','y','x','y','x','y','x' };
maze2[15] = gcnew cli::array<wchar_t>{ L'x','y','x','y','x','y','x','y','x','x','x','x','x','y','x','y','x','y','x','y','x' };
maze2[16] = gcnew cli::array<wchar_t>{ L'x','y','x','y','x','y','x','y','y','y','y','y','x','y','x','y','x','y','x','y','x' };
maze2[17] = gcnew cli::array<wchar_t>{ L'x','y','x','y','x','y','x','x','x','x','x','y','x','y','x','y','x','y','x','y','x' };
maze2[18] = gcnew cli::array<wchar_t>{ L'x','y','x','y','x','y','x','y','y','y','x','y','x','y','x','y','x','y','x','x','x' };
maze2[19] = gcnew cli::array<wchar_t>{ L'x','y','x','y','x','y','y','y','x','y','y','y','y','y','x','y','x','y','x','z','x' };
maze2[20] = gcnew cli::array<wchar_t>{ L'x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x' };
presetMazes->Add(maze2);


// ===== Лабіринт 3(демо) =====
cli::array<cli::array<wchar_t>^>^ maze5 = gcnew cli::array<cli::array<wchar_t>^>(21);

for (int i = 0; i < 21; ++i) {
    maze5[i] = gcnew cli::array<wchar_t>(21);
    for (int j = 0; j < 21; ++j) {
        if (i == 0 || i == 20 || j == 0 || j == 20)
            maze5[i][j] = L'x'; // границі
        else
            maze5[i][j] = L'y'; // шлях
    }
}

// Старт і фініш
maze5[1][1] = L'#';  // початок
maze5[19][19] = L'z'; // кінець
```

```
presetMazes->Add(maze5);


return presetMazes;


//Довільний лабіринт

for each (int dir in dirs)
{
   int nx = x + dx[dir];
   int ny = y + dy[dir];
   if (nx > 0 && ny > 0 && nx < maze->Length - 1 && ny < maze[0]->Length - 1 && maze[nx][ny] ==
L'x') {
       maze[nx][ny] = L'y';
       maze[x + dx[dir] / 2][y + dy[dir] / 2] = L'y';
       CarveMaze(maze, nx, ny, rng);
   }
}


}

cli::array<cli::array^>^ GenerateRandomMaze(int size) { if (size < 5) size = 5; if (size % 2 == 0) size
+= 1;

cli::array<cli::array<wchar_t>^>^ maze = gcnew cli::array<cli::array<wchar_t>^>(size);
for (int i = 0; i < size; i++) {
   maze[i] = gcnew cli::array<wchar_t>(size);
   for (int j = 0; j < size; j++)
       maze[i][j] = L'x';
}

std::mt19937 rng(static_cast<unsigned int>(time(nullptr)));

maze[1][1] = L'y';
CarveMaze(maze, 1, 1, rng);

maze[1][1] = L'#';
maze[size - 2][size - 2] = L'z';

return maze;
}
```