Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Бернацький Микита Дмитрович*
(ПІБ)

академічної групи *122-20-ск-1*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

освітньої програми *Комп'ютерні науки*
(назва освітньої програми)

на тему: *Реалізація штучного інтелекту для проходження лабіринту мовою Python*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи				
розділів:				
спеціальний	*доц. Реута О.В.*			
економічний	*проф. Вагонова О.Г.*			
Рецензент	*доц.*			
Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2023

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп'ютерних систем
(повна назва)

М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » 2023 року

ЗАВДАННЯ
на кваліфікаційну роботу

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *122-20-ск-1* *Бернацький М.Д.*
(група) (прізвище та ініціали)

тема кваліфікаційної роботи *Реалізація штучного інтелекту для проходження лабіринту мовою Python*

затверджена наказом ректора НТУ «ДП» від *16.05.2023* № *350-с*

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів проєктно-технологічної практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі. Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***13.05.2023 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки***	***27.05.2023 р.***

Завдання видав _____ *Реута О.В.*
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання _____ *Бернацький М.Д.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: 14*.01.2023 р.*

Термін подання кваліфікаційної роботи до ЕК: *.07.2023 р.*

РЕФЕРАТ

Пояснювальна записка: 65 с., 20 рис., 20 джерел.

Об'єкт розробки: реалізація штучного інтелекту для проходження лабіринту мовою Python.

Мета кваліфікаційної роботи: розробити додаток який буде на основі зваженого графу генерувати лабіринти та вирішувати їх за допомогою різноманітних алгоритмів пошуку шляху.

У вступі розглядається тема та конкретизується мета кваліфікаційної роботи та галузь її застосування, наведено обґрунтування актуальності теми та уточнюється постановка завдання.

У першому розділі проведено аналіз предметної області, розглянуто призначення розробки та галузь застосування, обґрунтовано підставу для розробки, також розглянуто вимоги до характеристик програмного та апаратного забезпечення.

У другому розділі виконано аналіз призначення створюваної системи, описано використання математичних методів та алгоритмі та те як вони працюють, проведено опис використаних технологій та створеного додатку.

В економічному розділі визначено трудомісткість розробленого програмного продукту, проведено підрахунок вартості роботи по створенню застосунку та розраховано час на його створення.

Актуальність даного програмного забезпечення визначається проблемою вирішення різноманітних транспортних задач, на прикладі лабіринту можна подивитись як працюють розроблювані алгоритми та ефективність їх роботи.

Список ключових слів: АЛГОРИТМ ПОШУКУ ШЛЯХУ, A-STAR, BFS, DFS, PYTHON, ЗВАЖЕНИЙ ГРАФ, РЕКУРСИВНИЙ ПОШУК ШЛЯХУ.

ABSTRACT

Explanatory note: 65 pages, 20 pics., 20 sources

The object of development: the implementation of artificial intelligence for passing the labyrinth in the Python language.

The purpose of the qualification work: to develop an application that will generate mazes based on a weighted graph and solve them using various path-finding algorithms.

In the introduction, the topic is considered and the purpose of the qualification work and the field of its application are specified, the justification of the relevance of the topic is given, and the statement of the task is clarified.

In the first section, an analysis of the subject area was carried out, the purpose of the development and the field of application were considered, the basis for the development was substantiated, and the requirements for the characteristics of the software and hardware were also considered.

In the second section, the purpose of the created system is analyzed, the use of mathematical methods and algorithms and how they work are described, the technologies used and the application created are described.

In the economic section, the labor intensity of the developed software product is determined, the cost of work on creating the application is calculated, and the time for its creation is calculated.

The relevance of this software is determined by the problem of solving various transport problems, using the example of a labyrinth, you can see how the developed algorithms work and their efficiency.

Keywords list: PATH SEARCH ALGORITHM, A-STAR, BFS, DFS, PYTHON, WEIGHTED GRAPH, RECURSIVE BACKTRACKING.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

IDE – Integrated Development Environment

ОС – операційна система;

ПЗ – програмне забезпечення

ЕОМ – електронно-обчислювальна машина;

ООП – об’єктно-орієнтоване програмування

RB - Recursive Backtracking

Ру – мова програмування Python

WG – weighted graph (зважений граф)

A* – A-Star algorithm

BFS – Breadth-First Search algorithm

DFS – Depth-First Search algorithm

ВСТУП

Лабіринти - загадкові та захоплюючі структури, які привертають увагу людей з різних сфер інтересів. Вони можуть бути візуальними головоломками для людини або серйозним випробування мдля алгоритмів пошуку шляху, складними моделями реальних проблем, які потребують вирішення.

Метою даної кваліфікаційної робота є вирішенню лабіринтів за допомогою зважених графів та алгоритмів A*, DFS та BFS пошуку шляху. Зважені графи - це графи, у яких кожній дуги або ребру надається вага, що відображає вартість переходу з одного вузла до іншого. Алгоритм A* - це ефективний алгоритм пошуку шляху, який поєднує у собі DFS та BFS, використовуючи евристичну оцінку для пришвидшення пошуку. Алгоритм DFS займається пошуком у глибину, тобто досліджує кожну гілку аж до її кінця, а потім переходить до наступної гілки допоки не знайде вихід. В сою чергу, алгоритм BFS робить пошук шляху у ширину, тобто поступово відвідує кожен вузол , що дозволяє знаходити найкоротший шлях від початкової до цільової точки.

У даній роботі розглянуті основні принципи моделювання лабіринтів за допомогою зважених графів та застосування різноманітних алгоритмів для пошуку оптимальних шляхів рішення у випадково створених або завантажених лабіринтах. Алгоритми A*, DFS та BFS мають свої унікальні особливості та можуть бути використані в залежності від конкретних вимог та характеристик лабіринту.

На прикладі вирішення лабіринтів можна розглядати як діють та ефективність алгоритмів пошуку шляху. Алгоритми пошуку шляху є важливою складовою багатьох задач. Наприклад задачі навігації та планування шляху на мапах та у GPS-навігації. Вони дозволяють знаходити найкоротший шлях від однієї точки до іншої, уникати перешкод та враховувати обмеження маршруту.

Алгоритми пошуку шляху широко використовуються в робототехніці для планування рухів роботів. Вони допомагають роботам знаходити оптимальний

шлях до цілі, уникати перешкод та враховувати різні обмеження, такі як обмеження швидкості або обмеження оборотів.

Також такі алгоритми використовуються у логістиці та управлінні ланцюгом постачання для оптимізації маршрутів доставки. Наприклад вони дуже допомагають у військовій справі знаходити найкоротші маршрути для доставки грузів, мінімізуючи витрати на паливо та час доставки.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Загальні відомості з предметної області

Вирішувач лабіринтів є програмою або алгоритмом, який призначений для пошуку оптимального шляху в лабіринті. Лабіринт може бути представлений у вигляді сітки з комірками або графу, де кожна комірка або вузол відповідає певній позиції в лабіринті, а ребра або рухи з'єднують сусідні комірки або вузли.

Основна задача вирішувача лабіринтів полягає у знаходженні шляху від початкової точки до кінцевої точки у лабіринті. Це може включати пошук найкоротшого шляху, оптимального шляху, шляху з мінімальною кількістю кроків або з іншими обмеженнями.

Самі лабіринти можуть мати різну форму та складність. Вони можуть бути представлені у вигляді двовимірних сіток або графів, де кожна клітина або вузол відповідає певній частині лабіринту. Лабіринти можуть мати стіни, перешкоди або спеціальні властивості.

Вирішувач лабіринтів зазвичай використовує різні алгоритми пошуку шляху, такі як алгоритм Дейкстри, алгоритм A*, алгоритми зворотного пошуку, пошук в ширину (BFS), пошук в глибину (DFS) та інші. Ці алгоритми використовуються для ефективного перебору можливих шляхів в лабіринті з метою знайти найкоротший шлях до цілі або вирішити задачу, якщо лабіринт має специфічні правила чи обмеження.

Для вирішення лабіринтів, алгоритми зазвичай оперують збереженням поточної позиції, стеком або чергою станів, і виконують ітерації до досягнення цілі або до повного обходу лабіринту. Вони можуть використовувати евристичну оцінку (наприклад, відстань Манхеттену або евклідова відстань) для визначення найбільш перспективного напрямку руху.

Деякі алгоритми вирішувачів лабіринтів використовують евристику, щоб оцінити потенційну вартість або відстань до кінцевої точки. Це допомагає алгоритмам зосередитися на більш ймовірних шляхах та прискорює їх роботу. Наприклад, алгоритм A* використовує комбінацію фактичної вартості пройденого шляху та оцінки відстані до кінцевої точки для прийняття рішень щодо найкращого напрямку.

Вирішувачі лабіринтів застосовуються в різних галузях, включаючи робототехніку, комп'ютерні ігри, штучний інтелект та навігацію. Вони можуть використовуватися для планування маршрутів, визначення найшвидшого шляху до цілі, уникнення перешкод, пошуку вихідних шляхів та інших подібних завдань. Такі системи можуть мати важливе практичне застосування, наприклад, для автономних роботів у робототехніці, які повинні навігувати у складних середовищах з перешкодами [13].

1.2. Призначення розробки та галузь застосування

Темою бакалаврської дипломної роботи є: «Реалізація штучного інтелекту для проходження лабіринту мовою Python». Метою роботи є створення програмного застосунку який генерує лабіринт та знаходить оптимальний шлях його вирішення.

Головні критерії розроблювального застосунку:

– Ефективність: застосунок повинен забезпечувати швидку та ефективну генерацію та вирішення лабіринтів. Він повинен оптимально використовувати ресурси системи та мати оптимізований алгоритм для швидкого знаходження шляху в лабіринті.

– Візуалізація: застосунок повинен надавати можливість візуального відображення лабіринтів. Візуалізація повинна бути зрозумілою та чіткою, демонструвати структуру лабіринту, шляхи та перешкоди. Візуалізація може включати кольори, символи або інші графічні елементи для покращення візуального сприйняття.

– Перевірка вирішеності: застосунок повинен мати можливість перевіряти, чи є вирішений лабіринт правильним, тобто чи присутній єдиний шлях від початку до кінця. Він повинен надавати засоби для перевірки правильності вирішення та надавати відповідні повідомлення користувачеві.

– Надійність та стабільність: застосунок повинен бути надійним та стабільним, без збоїв або непередбачуваних помилок. Він повинен відповідати стандартам програмного забезпечення та мати високу якість коду, що сприяє стабільній роботі та запобігає можливим проблемам та помилкам.

1.3. Підстава для розробки

Відповідно до ОКХ та ОПП, згідно навчального плану та графіків навчального процесу, в кінці навчання студент виконує кваліфікаційну роботу (проект). Тема роботи узгоджується з керівником проекту, випускаючою кафедрою, та затверджується наказом ректора.

Отже, підставами для розробки (виконання кваліфікаційної роботи) є:

– ОПП за напрямом підготовки 122 «Комп’ютерні науки»;

– графік навчального процесу та навчальний план;

– наказ ректора Національного технічного університету «Дніпровська політехніка» № 350-с від 16.05.2023 р;

– завдання на дипломний проект на тему «Реалізація штучного інтелекту для проходження лабіринту мовою Python».

1.4. Постановка завдання

Застосунок повинен реалізувати такі дії як:

– Застосунок повинен генерувати лабіринти за заданими параметрами.

– Застосунок повинен знайти шлях від початку до виходу в лабіринті.

– Застосунок повинен надати візуалізацію лабіринту з використанням графічних елементів, які представляють стіни, прохідні шляхи та вихід.

Для виконання проекту необхідно:

– Розробити архітектуру застосунку, визначити компоненти системи, їх взаємозв'язки та функціональність..

– Створити зручний інтерфейс користувача, що дозволяє взаємодіяти з додатком.

– Реалізувати алгоритми генерації та розрішення лабіринту.

– Провести ретельне тестування додатку, включаючи тестування генерації, розрішення лабіринту та інтерфейсу користувача.

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

Розроблене програмне забезпечення, для того, щоб досягнути поставлених цілей, повинно підтримувати виконання таких дій:

– використання відповідного алгоритму генерації для створення унікального лабіринту;

– відображення лабіринту на екрані з використанням графічних елементів;

– використання відповідного алгоритму розрішення для знаходження шляху від початку до виходу;

– візуалізація результату пошуку шялху;

– забезпечення зручного інтерфейсу користувача для легкого використання всіх функцій додатку.

Для підтримки вище перераховані функцій у додатку має бути реалізовано:

– підтримка використовуємої операційної системи та доступ до програми через неї;

– програмна та апаратна сумісності;

– стандартна конфігурація яка дає змогу ввести застосунок в експлуатацію.

1.5.2. Вимоги до інформаційної безпеки

Для коректної роботи програми потрібно реалізувати:

– Можливість редагування даних.

– Можливість тривалої роботи протягом 24 годин.

– Контроль та обробка вхідних даних.

– Збереження цілісності даних у випадку збою системи.

– Локальне збереження даних для більшої захищеності.

1.5.3. Вимоги до складу та параметрів технічних засобів

Для забезпечення надійного функціонування програмного забезпечення необхідно, щоб обчислювальна машина, на якій буде експлуатуватися створений застосунок, мала такі характеристики:

– Процесор AMD Ryzen 5 3600. З тактовою частотою 3,6 ГГц та вище

– Не менше ніж 6 Гб оперативної пам’яті.

– 1 Гб вільного місця на твердотілому накопичувач

– Монітор з діагоналлю 15" та більше.
– Маніпулятор «миша».
– Клавіатура.
– Встановлена ОС Windows 10 або новіша

Вище наведені характеристики являють собою рекомендовані. Це означає, що при наявності характеристик не нижче зазначених, розроблений додаток буде функціонувати відповідно до вимог щодо надійності, безпеки та швидкості обробки даних.

1.5.4. Вимоги інформаційної та програмної сумісності

Вимоги інформаційної та програмної сумісності визначають набір критеріїв і стандартів, які програмне забезпечення повинно відповідати для забезпечення його взаємодії з іншими системами, платформами, компонентами та користувачами. Ці вимоги спрямовані на забезпечення сумісності програми з різними середовищами, апаратними та програмними засобами, що дозволяє їй працювати на різних платформах та з іншими системами без конфліктів або непередбачених проблем.

Для коректного функціонування програми необхідно, щоб програмне забезпечення обчислювальної машини, на якій буде експлуатуватися створений застосунок, відповідало наступним вимогам:.

– ОС Windows 10;
– встановлений пакет Python 3.9.13 та вище.

Програмний додаток має бути реалізований на мові програмування Python з використанням базових бібліотек для знаходження оптимального шляху вирішення лабіринту. Для візуалізації було використано базову бібліотеку Tkinter.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1. Функціональне призначення системи

Метою бакалаврської дипломної роботи було розробити програмний застосунок мовою Python який генерує випадковий лабіринт з заданими параметрами та знаходить оптимальний шлях його вирішення в залежності від обраного алгоритму.

Призначення розроблюваного додатку:

- Створювати лабіринти
- Налаштовувати параметри створення лабіринтів
- Зберігати створені лабіринти
- Вирішувати лабіринти за допомогою різних алгоритмів

2.2. Опис застосованих математичних методів

Для вирішення лабіринту зазвичай використовують зважений граф та алгоритми пошуку шляху від вказаної початкової точки до вказаної кінцевої. У створюваному застосунку використовуються наступні види алгоритмів: A-Star, BFS (Breadth-First Search) та DFS (Depth-First Search) [8].

Зважений граф (weighted graph) — це тип графа, де кожному ребру присвоєно числове значення, відоме як вага або вартість. Це фундаментальна структура даних, яка використовується для представлення зв’язків або зв’язків між об’єктами, де вагові коефіцієнти на краях можуть представляти різні величини, такі як відстані, витрати, місткість або будь-який інший відповідний показник. Граф складається з набору вершин (також званих вузлами), з’єднаних ребрами. У зваженому графі кожне ребро має додаткове значення ваги. Вершини представляють сутності або елементи, а ребра представляють відносини або

зв’язки між ними. Ребра можуть бути спрямованими (від однієї вершини до іншої) або ненаправленими (двонаправленими) (рис. 2.1).

Визначальною характеристикою зваженого графа є призначення ваг на його ребра.. Ваги можуть представляти різні величини залежно від контексту. Наприклад, у транспортній мережі ваги можуть представляти відстані між місцями. Конкретна інтерпретація ваг залежить від проблеми, що моделюється. Ваги у зваженому графіку зазвичай представлені як числові значення. Ваги можуть бути цілими числами, числами з плаваючою комою або будь-яким іншим подібним числовим представленням. У деяких випадках ваги також можуть бути нечисловими, наприклад символьними значеннями або категоріальними мітками.

Зважені графи є інструментами у вирішенні проблем оптимізації, таких як проблема комівояжера (це комбінаторна оптимізаційна задача, в якій необхідно знайти найкоротший можливий маршрут, який проходить через всі задані міста (точки) і повертається в початкове місто). Ваги визначають цільову функцію, яку потрібно оптимізувати, наприклад мінімізацію відстаней, витрат або максимізацію ефективності. Зважені графи зазвичай використовуються для моделювання мереж оптимальних маршрутів будь-якої транспортної системи, де ваги представляють відстані, час у дорозі або витрати. Вони допомагають знайти оптимальні маршрути, розрахувати найкоротші шляхи або мінімізувати витрати.

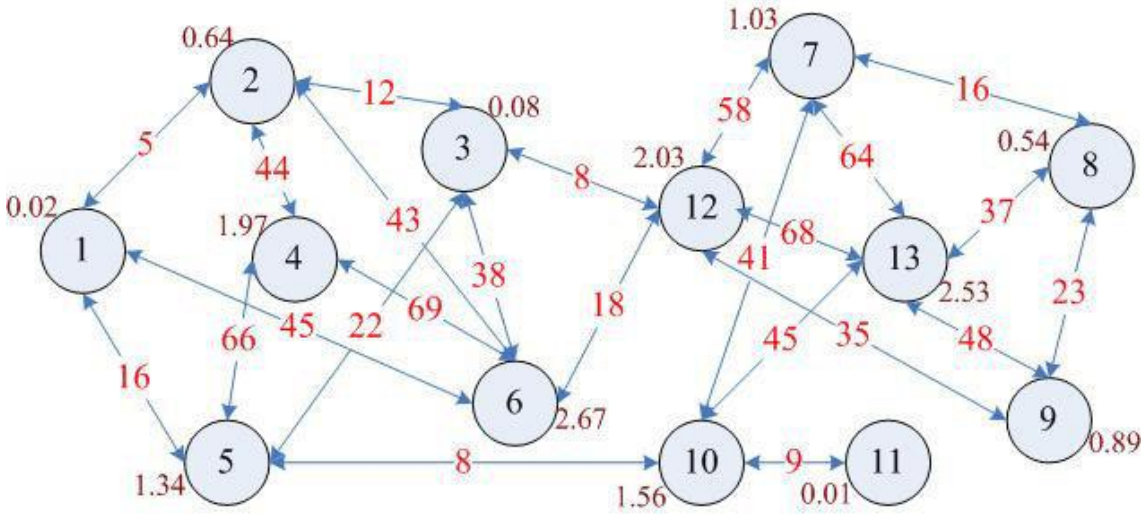


Рис. 2.1 – Приклад зваженого графу

Алгоритм A* (A-Star) — це популярний алгоритм пошуку шляху, який використовується для пошуку найкоротшого шляху між двома вузлами на графі. Основною перевагою алгоритму A-Star є те, що він поєднує ефективність евристичного пошуку з гарантіями оптимальності алгоритму Дейкстри. Він має тенденцію досліджувати найбільш перспективні шляхи на ранній стадії, зменшуючи кількість вузлів, які потрібно відвідати, і покращуючи ефективність виконання.

Цей алгоритм є ефективним, оскільки він використовує як фактичну вартість від початкового вузла до заданого вузла, так і евристичну оцінку відстані, що залишилася від цього вузла до цільового вузла. Евристична функція залежить від конкретної проблеми і повинна надавати оптимістичну оцінку [9].

Алгоритм BFS (Breadth-First Search) — це простий алгоритм обходу графа, який досліджує всі вершини графа в порядку ширини, тобто він відвідує всі вершини на одному рівні перед переходом на наступний рівень. Цей алгоритм гарантує, що всі вершини будуть відвідані в порядку ширини, тобто вершини, ближчі до початку, будуть досліджені перед вершинами, які розташовані далі.

Алгоритм DFS (Depth-First Search) — це алгоритм обходу графа, який досліджує якомога далі кожну гілку запам’ятовуючи розгалуження на нові гілки. Він проходить якомога глибше по графу, перш ніж повернутись досліджувати інші гілки. Перевагою цього алгоритму є відносно низькі вимоги до пам’яті, оскільки потрібно лише зберігати інформацію про поточний досліджуваний шлях. Він не вимагає зберігання інформації про всі вершини в графі.

Основним недоліком цього алгоритму є те, що не гарантується знаходження рішення, якщо воно існує. Він може застрягти в нескінченних циклах або не в змозі дослідити певні частини графа. Алгоритм DFS не гарантує знаходження найкоротшого шляху чи оптимального рішення. Він може знайти рішення, але воно може бути не найефективнішим чи найкоротшим [10].

2.3. Опис використаних технологій та мов програмування

При створенні проекту використовувався об’єктно-орієнтований підхід програмування, сам код був написаний інтерпретованою мовою програмування Python з використанням стандартних бібліотек таких як Enum, Tkinter та інших. Процес розробки відбувався в інтегрованому середовищі розробки IntelliJ IDEA.

2.3.1 Об’єктно-орієнтований підхід

ООП — це парадигма програмування, яка фокусується на організації проектування та розробки програмного забезпечення навколо об’єктів, які є екземплярами класів. Він забезпечує спосіб структурування та інкапсуляції даних і функціональних можливостей у програмі, що робить її більш модульною, придатною для багаторазового використання та простішою в обслуговуванні [2].

Ключові поняття та особливості об’єктно-орієнтованого підходу:

1. Ядром об’єктно-орієнтованого підходу є об’єкти. Об’єкт представляє конкретний екземпляр класу, який інкапсулює як дані (атрибути або властивості), так і поведінку (методи або функції). Об’єкти є «будівельними блоками» об’єктно-орієнтованих систем і можуть представляти сутності реального світу, такі як автомобіль, банківський рахунок або профіль користувача, а також абстрактні поняття, такі як структура даних або алгоритм.

2. Класи - це «креслення» або «шаблони» для створення об’єктів. Вони визначають структуру та поведінку, якими повинні володіти об'єкти певного типу (екземпляри класу). Клас інкапсулює дані у вигляді атрибутів і вказує методи, які працюють з цими даними. Він діє як «фабрика» для створення об’єктів, оскільки об’єкти є екземплярами класів. Класи надають способи визначення та організації загальних властивостей і поведінки одразу для кількох об’єктів.

3. Інкапсуляція — це механізм групування даних і методів, які маніпулюють цими даними в межах класу. Він забезпечує приховування даних і інформації, надаючи так званий публічний інтерфейс (публічні методи) для взаємодії з об’єктом, зберігаючи внутрішні деталі реалізації прихованими (рис. 2.2). Інкапсуляція захищає цілісність даних об’єкта, керуючи доступом за допомогою методів, і запобігає прямим маніпулюванням даними ззовні об’єкта. Це сприяє модульності та зменшує залежності між різними частинами програми.

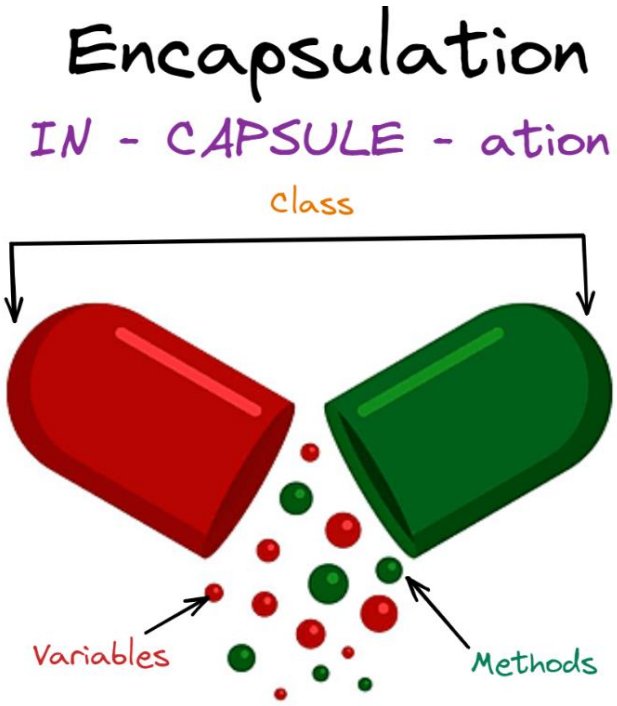


Рис. 2.2 – Абстрактний вигляд інкапсуляції

4. Успадкування дозволяє створювати нові класи на основі існуючих класів. Це означає що новий клас, який називається підкласом або похідним класом, успадковує властивості та поведінку існуючого класу, який називається суперкласом або базовим класом (рис. 2.3). Підклас розширює або спеціалізує функціональність суперкласу, додаючи нові функції або замінюючи існуючі. Спадкування полегшує повторне використання коду, підтримує концепцію зв’язків «is-a» та дозволяє ієрархічну організацію класів.

Успадкування допомагає створити ієрархію класів, де головні класи (суперкласи) знаходяться на вищих рівнях, а похідні класи (підкласи) знаходяться на нижчих рівнях. Підкласи успадковують атрибути та методи своїх суперкласів і можуть далі розширювати або змінювати їх за необхідністю. Цей механізм дозволяє створювати спеціалізовані класи, зберігаючи при цьому повторне використання коду та узгодженість, надані суперкласом.

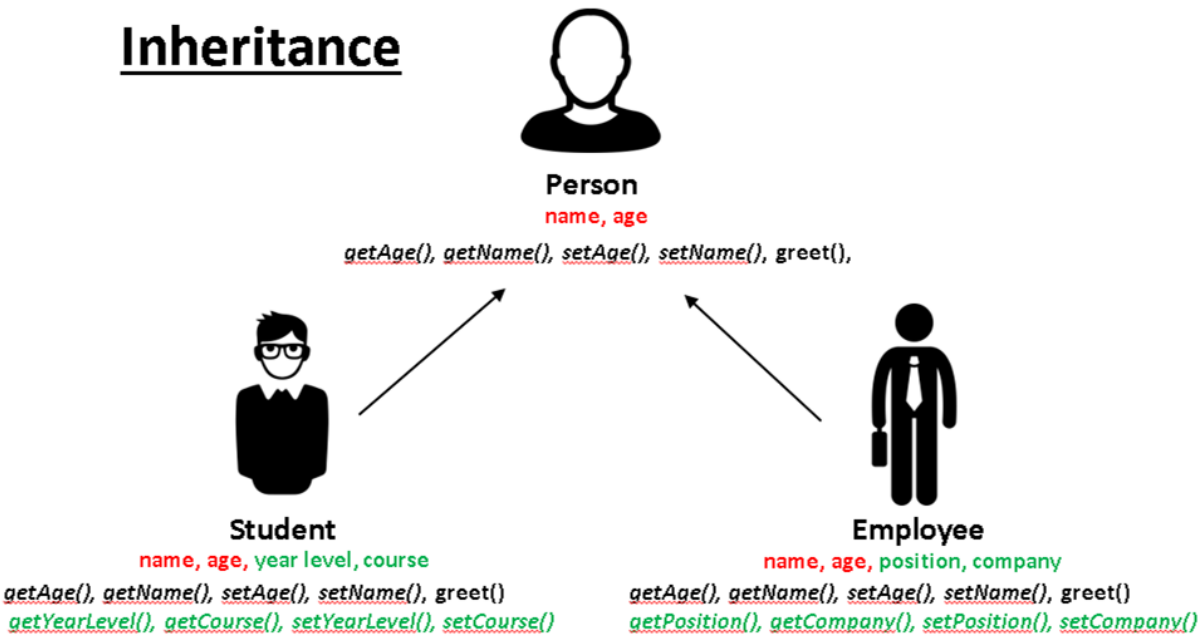


Рис. 2.3 – Абстрактний вигляд успадкування

5. Поліморфізм дозволяє розглядати об’єкти різних класів як об’єкти спільного суперкласу. Це дозволяє використовувати єдиний інтерфейс для представлення різних типів об’єктів (рис. 2.4). Поліморфізм буває двох форм: статичний поліморфізм (досягається за допомогою перевантаження функцій) і динамічний поліморфізм (досягається за допомогою перевизначення методу).

У динамічному поліморфізмі підклас може забезпечувати свою реалізацію методу, який вже визначений у його суперкласі. Це дозволяє різним об’єктам, що належать до різних класів, але мають спільний суперклас, по-різному реагувати на виклик одного методу. Поліморфізм покращує гнучкість, розширюваність і повторне використання коду, оскільки дає змогу писати загальний код, який

може працювати з об’єктами різних типів, якщо вони мають спільний інтерфейс або суперклас.

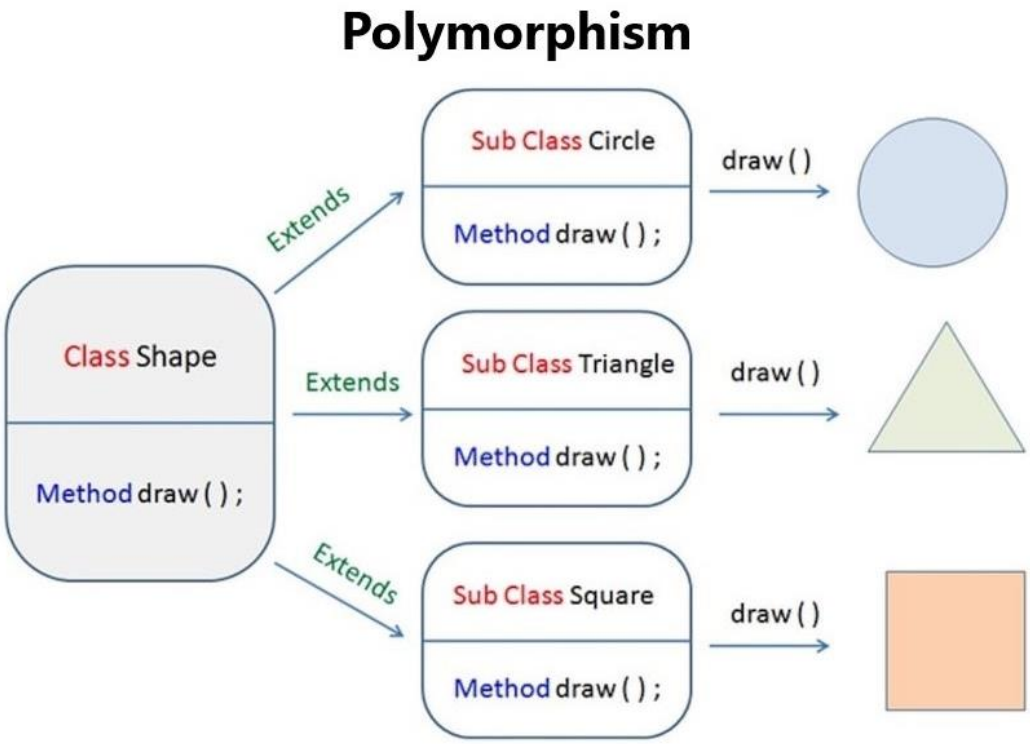


Рис. 2.4 – Абстрактний вигляд поліморфізму

6. Абстракція — це процес спрощення складних систем шляхом виявлення головних функцій і видалення непотрібних елементів. Вона фокусується на тому, що робить об’єкт, а не на тому, як він це робить. Абстракція дозволяє програмістам створювати абстрактні класи або інтерфейси, які визначають набір методів, не вказуючи деталей їх реалізації (рис. 2.5). Ці абстрактні сутності служать «схемами» для конкретних класів, визначаючи очікувану поведінку, яку повинні реалізувати підкласи.

Абстракція забезпечує високорівневе уявлення про об’єкти та їх взаємодію, дозволяючи розробникам проектувати системи та міркувати про них на концептуальному рівні. Це допомагає керувати складністю, розбиваючи систему на модульні компоненти, кожен з яких відповідає за певний набір функцій. Абстракція полегшує підтримку коду, оскільки зміни, внесені до

внутрішньої реалізації об’єкта, не впливають на код, який покладається на його абстрактний інтерфейс.

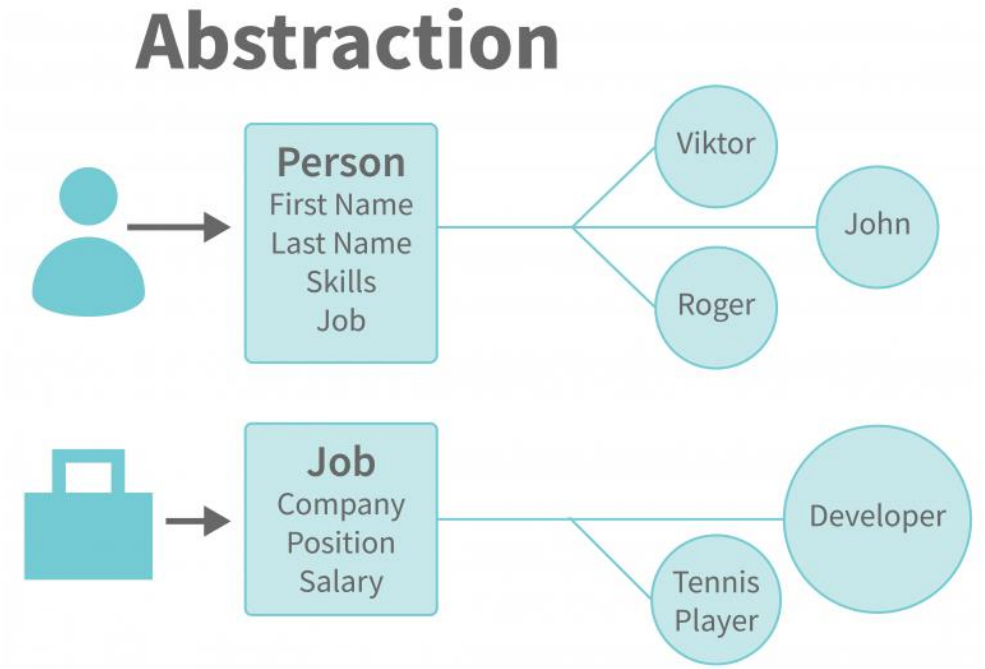


Рис. 2.5 – Абстрактний вигляд поліморфізму

Узагальнюючи, можна сказати, що об’єктно-орієнтований підхід сприяє модульності, яка поділяє системи на автономні модулі (класи чи об’єкти). Кожен модуль інкапсулює певну функціональність і забезпечує чітко визначений інтерфейс для взаємодії з іншими модулями. Ця модульна структура покращує організацію коду, підвищує читабельність коду та дозволяє групам працювати над різними модулями одночасно. Модульність, яку пропонує ООП, також полегшує повторне використання коду. Класи та об’єкти можна легко повторно використовувати в різних частинах програми або навіть в інших проектах. Успадковуючи від існуючих класів або використовуючи попередньо визначені об’єкти, розробник може використовувати вже створений код і не переписувати його багато разів. Повторне використання коду не тільки економить час і зусилля, але й покращує послідовність і зменшує ймовірність появи помилок [3].

2.3.2 Мова програмування Python

Python — це інтерпретована мова програмування високого рівня, яка відома своєю простотою та читабельністю. Вона була створена Гвідо ван Россумом і вперше випущена у 1991 році. Мова програмування Python використовує чистий і зрозумілий синтаксис із значущими відступами. Він забезпечує стиль кодування, який підкреслює читабельність, зменшуючи складність коду.

Python є інтерпретованою мовою, що означає, що код виконується рядок за рядком без необхідності компіляції. Це забезпечує інтерактивний режим, у якому користувачі можуть вводити та виконувати Ру-код без необхідності створення та запуску окремого скрипту. Також Python підтримує кілька парадигм програмування, включаючи процедурне, об'єктно-орієнтоване та функціональне програмування. Що дозволяє розробникам вибирати найбільш відповідний підхід для кожної окремої ситуації. У мові Ру автоматично керується пам'ять за допомогою збирача сміття, звільняючи розробників від ручного керування пам'яттю. Ця функція спрощує виділення та зняття пам'яті, роблячи Python більш зручним для розробника, а код менш громістким.

Одним з плюсів мови програмування Python є те, що він постачається з великою стандартною бібліотекою, яка надає широкий спектр модулів і функцій для звичайних завдань, таких як файловий ввід/вивід, мережа, веб-розробка, доступ до бази даних тощо. Стандартна бібліотека зменшує потребу у зовнішніх залежностях і прискорює розробку. Якщо стандартної бібліотеки не вистачає, то ця мова має величезну екосистему сторонніх бібліотек і пакетів, які розширюють його функціональність. Наприклад такі бібліотеки, як NumPy, Pandas, Matplotlib, TensorFlow, які широко використовуються для наукових обчислень, аналізу даних, візуалізації, машинного навчання тощо [1].

2.3.3 Бібліотека Tkinter

Tkinter — це стандартна Ру бібліотека, яка використовується для створення графічних інтерфейсів користувача (GUI). Він надає набір інструментів і віджетів, які дозволяють розробникам створювати інтерактивні та візуально привабливі програми. Ця бібліотека надає широкий спектр готових віджетів, таких як кнопки, мітки, поля введення, прапорці, перемикачі, спадні меню тощо. Ці віджети служать будівельними блоками для побудови інтерфейсу користувача програми. Tkinter слідує парадигмі програмування, керування подіями. Це означає, що програма чекає дій або подій користувача (таких як натискання кнопок або рух миші) і реагує відповідно. Розробники можуть визначити обробники подій або зворотні виклики для виконання певних дій, коли відбувається подія. Tkinter пропонує кілька менеджерів компонування, які допомагають упорядковувати та розміщувати віджети у вікні чи фреймі. Найпоширенішим менеджером макета є макет сітки, який дозволяє організовувати віджети в рядки та стовпці. Ця бібліотека надає методи керування розміром і розташуванням вікон і віджетів. Розробники можуть вказувати розміри та керувати тим, як віджети розширюються чи звужуються під час зміни розміру вікна. Також надає методи налаштування зовнішнього вигляду віджетів, змінюючи такі атрибути, як кольори, шрифти, межі тощо, підтримує використання зображень і піктограм у програмах графічного інтерфейсу користувача [19].

2.3.4 Середовище розробки IntelliJ IDEA

IntelliJ IDEA — це інтегроване середовище розробки (IDE), розроблене JetBrains. Воно було створене для розробки на мовах програмування Kotlin, Groovy, Scala, Python, JavaScript тощо. IntelliJ IDEA надає розробникам повний набір інструментів і функцій для оптимізації процесу розробки програмного забезпечення та підвищення продуктивності. IDE пропонує високоінтелектуальний редактор коду з розширеним доповненням коду, підсвічуванням синтаксису, навігацією по коду та можливостями рефакторингу. Воно аналізує код і надає пропозиції та швидкі виправлення, які допомогають написати чистий код без помилок (рис. 2.6). IntelliJ IDEA має вбудовану підтримку популярних систем контролю версій, таких як Git, Subversion, Mercurial і Perforce, що дозволяє виконувати операції керування версіями безпосередньо з програми. Також IntelliJ IDEA підтримує різні інструменти побудови проекту, зокрема Gradle, Maven і Ant що дозволяє легко керувати залежностями та запускати збірки без необхідності щось завантажувати та налаштовувати власноруч.

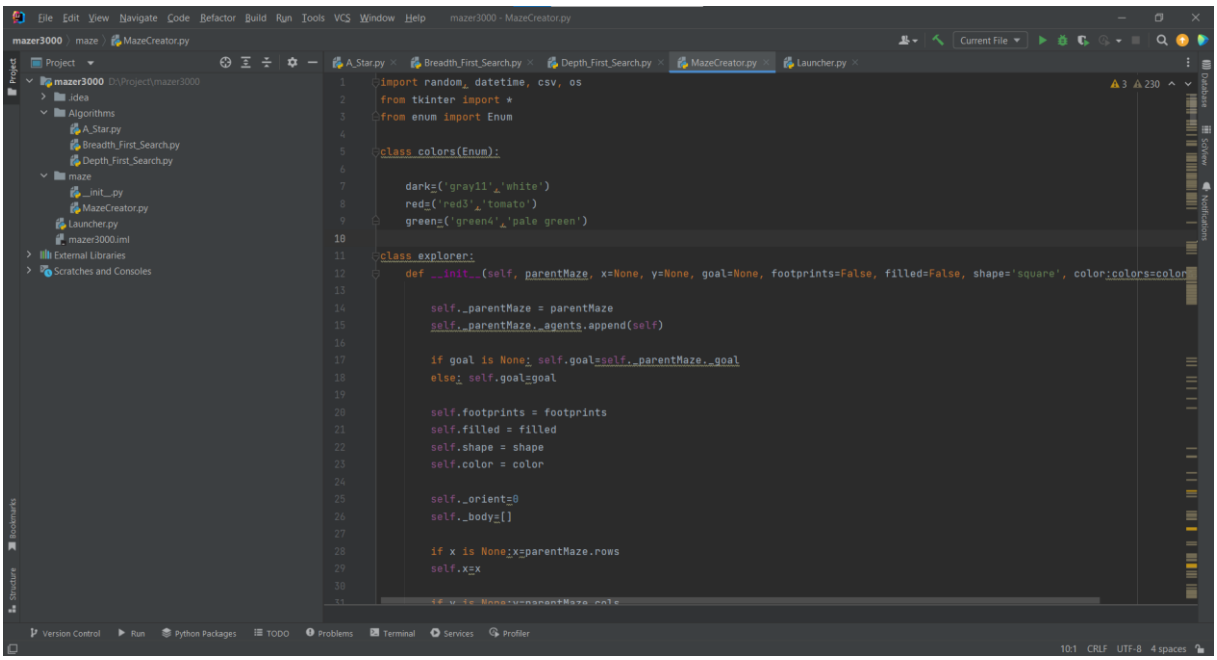


Рис. 2.6 – Інтерфейс IntelliJ IDEA під час розробки

Зокрема для розробки на мові програмування Python саме IntelliJ IDEA підтримує віртуальні середовища для Ру проектів, дозволяючи створювати ізольовані середовища з певними версіями Python і залежностями пакетів. Він також забезпечує інтеграцію з такими менеджерами пакетів, як pip і conda, що полегшує встановлення, оновлення та керування пакетами Python. IntelliJ IDEA пропонує інтелектуальне завершення коду, пропозиції та автоматичний імпорт для коду. Він аналізує кодову базу та надає контекстно-залежні пропозиції для прискорення процесу кодування. IDE дозволяє легко переміщатися по кодовій базі Python за допомогою таких функцій, як «Перейти до визначення», «Знайти використання» та «Перегляд структури», а також надає різні варіанти рефакторингу для покращення структури та зручності обслуговування коду [6].

2.4. Опис структури системи та алгоритмів її функціонування

2.4.1 Алгоритм "Recursive Backtracking"

Для створення генератору лабіринту використовувався алгоритм "Recursive Backtracking".

Алгоритм "Recursive Backtracking" є одним з популярних способів генерації випадкових лабіринтів. Він базується на рекурсивному підході та використовує стек для відстеження шляху. Основна ідея полягає в тому, щоб розпочати зі стартової клітини та рухатися випадково проникаючи вглиб лабіринту, будуючи шляхи через невідвідані клітини. Кожен раз, коли алгоритм доходить до точки без варіантів (немає невідвіданих сусідніх клітин), він повертається назад (backtracks) до попередньої клітини та продовжує процес генерації з іншої невідвіданої клітини (рис. 2.7). Цей процес відбувається до тих пір, поки не будуть відвідані всі клітини лабіринту і побудовані шляхи [11].

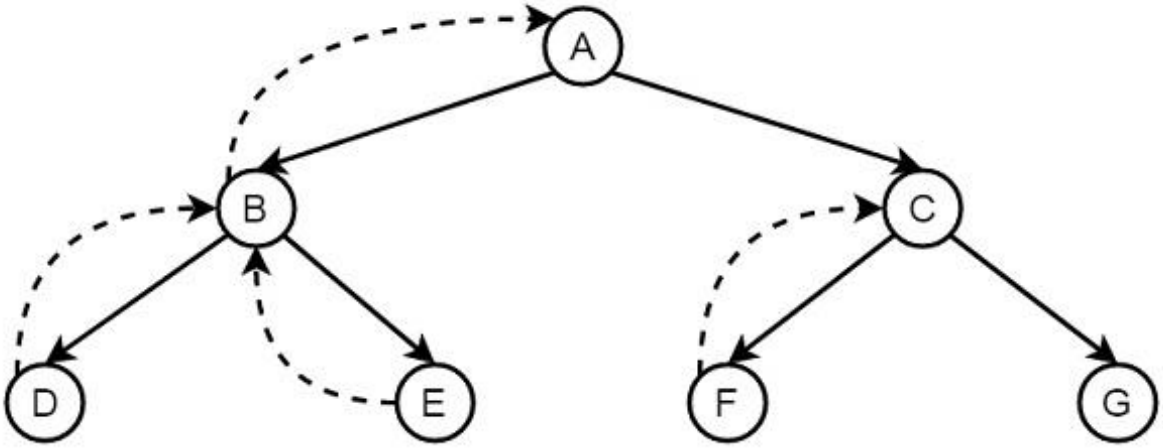


Рис. 2.7 – Приклад алгоритму "Recursive Backtracking"

В результаті виконання алгоритму "RB" утворюється випадковий лабіринт з прохідними шляхами та стінами. Цей алгоритм забезпечує випадковість лабіринту та може створювати різні варіації лабіринтів з різною складністю.

Особливістю алгоритму "Recursive Backtracking" є його здатність до глибокого проникнення в лабіринт та створення довгих коридорів. Однак, це може призвести до виникнення довгих тупикових шляхів. Для покращення результату можна використовувати додаткові правила, наприклад, додаткові обмеження на довжину коридорів або збереження більшої кількості стін.

Застосування алгоритму "RB" дозволяє генерувати різноманітні та цікаві лабіринти, які можуть бути використані для різноманітних завдань, пов'язаних з розробкою алгоритмів пошуку шляхів, навігації або візуалізації [17].

2.4.2 Алгоритм "A*"

Алгоритм A* (A-star) - це ефективний алгоритм пошуку найкоротшого шляху в графі з вагованими ребрами. Він використовує принципи оцінки шляхів та пріоритетного вибору для ефективного пошуку найкоротшого шляху від початкової вершини до цільової вершини.

Основний принцип роботи алгоритму A* полягає в поєднанні оцінки вартості шляху до поточної вершини (g(x)) та оцінки вартості залишкового

шляху до цільової вершини (h(x)). Кожна вершина отримує значення оцінки f(x) = g(x) + h(x), яке вказує на загальну вартість шляху через цю вершину (рис. 2.8).

Для функціонування алгоритм A* спочатку ініціалізує початкову вершину та встановлює її оцінку g(x) = 0 та оцінку f(x) = h(x), де h(x) - евристика від поточної вершини до цільової вершини. Далі ініціалізовує список відкритих вершин та додає до нього початкову вершину. Після чого ініціалізовує порожній список закритих вершин.

Поки список відкритих вершин не порожній: обирається вершину з найменшою оцінкою f(x) зі списку відкритих вершин і призначає її поточною вершиною. Після чого переміщує поточну вершину зі списку відкритих вершин в список закритих вершин. Якщо поточна вершина є цільовою вершиною, алгоритм вибору завершується, оскільки найкоротший шлях до цільової вершини знайдено.

Для кожної сусідньої вершини, що ще не має оцінки спочатку обчислюється оцінка g(x) для сусідньої вершини, яка включає вартість переходу від поточної вершини до сусідньої. Зазвичай це вага ребра між поточною та сусідньою вершиною. Якщо сусідня вершина вже знаходиться у списку відкритих або закритих вершин і має меншу оцінку g(x), цю вершина пропускається. Далі встановлюється посилання на поточну вершину як попередню для сусідньої вершини та обчислюється оцінка f(x) для сусідньої вершини, що включає оцінку g(x) та еврестичну оцінку h(x) для наступної вершини. По закінченню сусідня вершина додається до списку відкритих вершин.

Якщо алгоритм завершився успішно (цільова вершина досягнута), то починається відновлення найкоротшого шляху. Починаючи з цільової вершини, використовуючи посилання на попередню вершину, здійснюється перехід до попередньої вершини та записується кожна вершину, яку проходить алгоритм, для створення найкоротшого шляху.

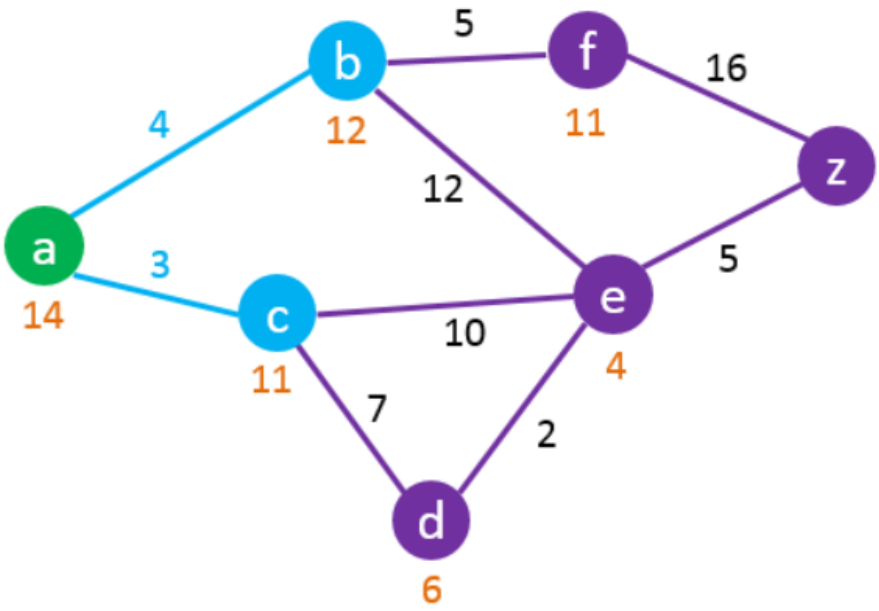


Рис. 2.8 – Приклад алгоритму "A*"

Після завершення алгоритму A*, найкоротший шлях можна отримати, рухаючись від цільової вершини до початкової за допомогою посилань на попередні вершини.

Оцінка залишкового шляху до цільової вершини, яка використовується в алгоритмі A*, називається евристикою (heuristic). Евристика використовується в алгоритмі A* для оцінки вартості залишкового шляху від поточної вершини до цільової вершини. Вона допомагає алгоритму приймати рішення про вибір найкращого шляху, орієнтуючись на очікувану вартість досягнення цілі.

У створеному додатку евристика використовує Манхеттенську відстань (сума модулів різниць по координатах) для обчислення оцінки. Було обрано саме Манхеттенську відстань тому, що вона завжди недооцінює вартість шляху від поточної вершини до цілі. Це означає, що вона ніколи не переоцінює вартість шляху і завжди дає реалістичну оцінку. Ще одна з причин це те, що обчислення Манхеттенської відстані дуже швидке. Вона вимагає лише обчислення модулів різниць по координатах точок. Це робить її практично застосовною для великих графів та складних задач. Також Манхеттенська відстань враховує тільки вертикальні та горизонтальні переміщення між точками, не враховуючи

діагональні переміщення. Що дозволяє алгоритму A* більш ефективно рухатись по сітці або графу, оскільки враховуються тільки найпростіші і допустимі шляхи.

Алгоритм A* гарантує знаходження найкоротшого шляху, якщо виконуються наступні умови:

- Ваги ребер (або вартість переходу між вершинами) не мають від'ємних значень.
- Функція оцінки h(x) (евристика) є допустимою, тобто ніколи не переоцінює вартість шляху до цілі.

Алгоритм A* є потужним інструментом для розв'язання задач навігації, пошуку шляху, планування траєкторій та багатьох інших варіантів, де необхідно знайти оптимальний шлях у графі з вагованими ребрами [15].

2.4.3 Алгоритм "BFS"

Алгоритм BFS працює на принципі пошуку в ширину, спочатку оброблюючи всі сусідні вершини поточної вершини перед переходом до наступного рівня. Це гарантує, що найдовший можливий шлях буде знайдено, оскільки алгоритм просувається поетапно по рівнях графу (рис. 2.9).

Алгоритм BFS ініціалізує початкову вершину та додає її до черги (queue) або списку відвіданих вершин. Далі ініціалізується список відвіданих вершин та додається початкова вершину до списку. Поки черга не порожня обирається перша вершина з черги і призначається поточною вершиною. Для кожної сусідньої вершини, яка ще не була відвідана додається сусідня вершина до черги або списку відвіданих вершин. Далі додається сусідня вершина до списку відвіданих вершин та встановлюється посилання на поточну вершину як попередню для сусідньої вершини.

Якщо алгоритм завершився успішно (досягнута цільова вершина), починається відновлення шляху, переходячи від цільової вершини до початкової за допомогою посилань на попередні вершини.

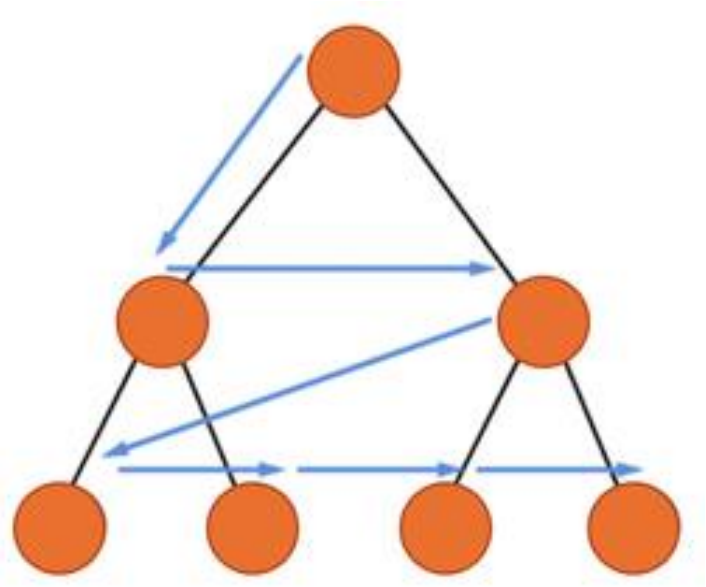


Рис. 2.9 – Приклад алгоритму "BFS"

Алгоритм BFS є ефективним для знаходження найкоротшого шляху в невагованих графах або графах з однаковими вагами ребер. Він також може бути використаний для виявлення циклів у графах, перевірки зв'язності та пошуку всіх досяжних вершин з початкової вершини.

Однак, варто відзначити, що алгоритм BFS не є оптимальним для графів з великою кількістю вершин або з великою кількістю ребер, оскільки вимагає збереження всіх відвіданих вершин у пам'яті.

Загалом, алгоритм BFS є потужним інструментом для розв'язання різних задач, пов'язаних з графами. Він простий у реалізації і часто використовується як основний алгоритм для пошуку шляхів, перебору вершин та визначення зв'язності у графах [16].

2.4.4 Алгоритм "DFS"

Алгоритм DFS працює на основі принципу "останній прийшов, перший пішов" (LIFO - Last-In-First-Out). Він досліджує граф в глибину, переходячи якомога глибше перед тим, як переходити до наступної вершини. Алгоритм розпочинає з початкової вершини і відвідує всі сусідні вершини по можливості, проникаючи вглиб графа, до тих пір, поки вже немає сусідніх невідвіданих вершин. Якщо знайдений шлях має цикл, алгоритм повертається назад до останньої розглянутої вершини з невідвіданими сусідніми вершинами і продовжує виконуватись (рис. 2.10).

Для виконання алгоритму DFS спочатку ініціалізується стек (stack) далі обирається початкова вершина і додається до цього стеку. Поки стек не порожній обирається остання вершину зі стеку і призначається поточною вершиною. Далі здійснюється перевірка, чи була поточна вершина відвідана. Якщо так, здійснюється переход до наступної ітерації, поточна вершину відмічається як відвідана та додаються сусідні вершини поточної вершини, які ще не були відвідані, до стеку. Алгоритм завершується, коли стек стає порожнім і всі вершини були відвідані або дочасно було знайдено рішення.

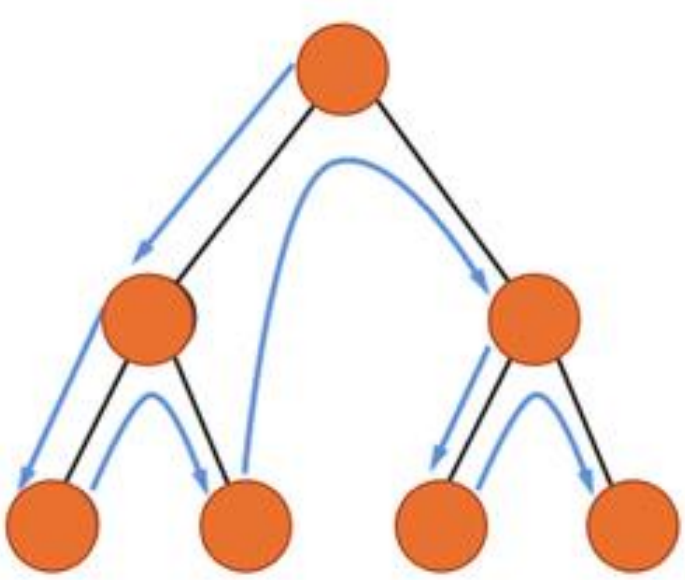


Рис. 2.10 – Приклад алгоритму "DFS"

Алгоритм DFS використовується для знаходження шляхів, перевірки зв'язності, виявлення циклів та інших операцій на графах. Він також може бути використаний для генерації дерев та перебору комбінацій. Однак, значним мінусом є те, що алгоритм DFS не гарантує знаходження найкоротшого шляху та він може потрапити в нескінченний цикл, якщо граф має цикли [14].

2.5. Обгрунтування та організація вхідних та вихідних даних програми

Вхідні дані потрапляють в систему обробки у вигляді задання користувачем налаштувань які записуються до змінних та використовуються для генерування масиву даних або у вигляді CSV файлу з масивом даних зібраних у таблицю. Цей масив даних являє собою лабіринт який використовується алгоритмами пошуку шляху.

2.6. Опис роботи розробленої системи

Розроблювана система являє собою вирішувач лабіринтів, який генерує або використовує вже згенерований лабіринт та знаходить шляхи його вирішення. Ця система складається з двох компонентів: невеликого простого консольного користувацького інтерфейсу для задання початкових даних та інтерфейсу який відрисовує згенерований лабіринт, шляхи виконання алгоритму пошуку шляху та знайдений шлях вирішення лабіринту.

2.6.1. Використані технічні засоби

Так як додаток немає серверної частини та вся інформація оброблюється локально, тобто характеристиками ЕОМ на якій розроблювався та тестувався додаток є наступна конфігурація:

- Маніпулятор “миша”.
- Клавіатура.
- 50 Гб вільного місця на твердотілому накопичувачі
- Процесор Intel Core i5-10400 з тактовою частотою 2.9 ГГц.
- Інтегрований графічний процесор Intel UHD Graphics 630.
- 8 ГБ оперативної пам’яті.
- Монітор з діагоналлю 15".

2.6.2. Використані програмні засоби

Для розробки ПЗ використовувалась IDE IntelliJ IDEA 2022.3.1 з встановленим додатком для розробки на Ру та власне мова програмування Python з базовими математичними та графічними бібліотеками.

Для правильного функціонування додатку необхідні такі програмні засоби, як ОС Windows 10 та встановлений пакет Python 3.11.4.

2.6.3. Виклик та завантаження програми

Виклик програми відбувається за допомогою BAT файлу який запускає виконання програми. В свою чергу програма генерує масив даних та тимчасово зберігає його або виконує заданий CSV файл з раніше підготовленим масивом даних.

2.6.4. Опис інтерфейсу користувача

Після запуску програми відкривається консольне вікно в якому треба налаштувати хід виконання програми.

Спочатку необхідно обрати один з алгоритмів рішення лабіринту на вибір: A-Star, BFS або DFS (рис. 2.11).

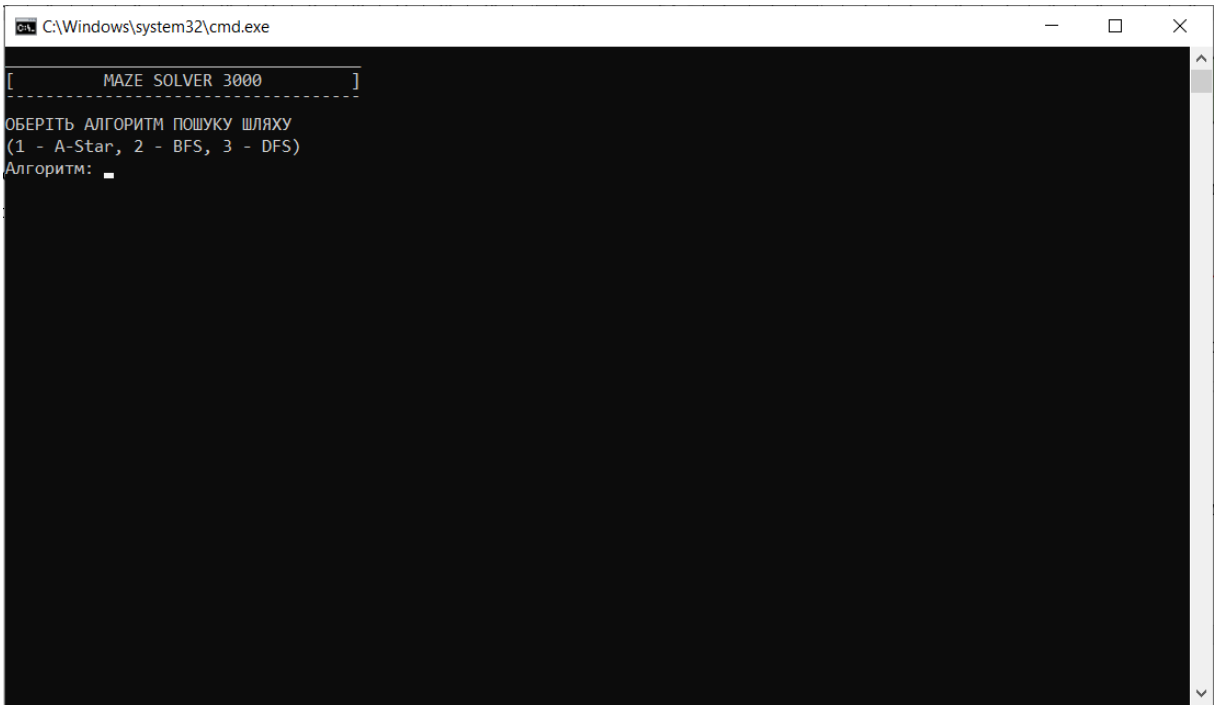


Рис. 2.11 – Вибір алгоритму при первинному налаштуванні

Після того як обрано алгоритм пошуку необхідно обрати який використовувати лабіринт (рис. 2.12).

На вибір є згенерувати новий за заданими далі параметрами або завантажити раніше створений у форматі CSV (рис. 2.13).

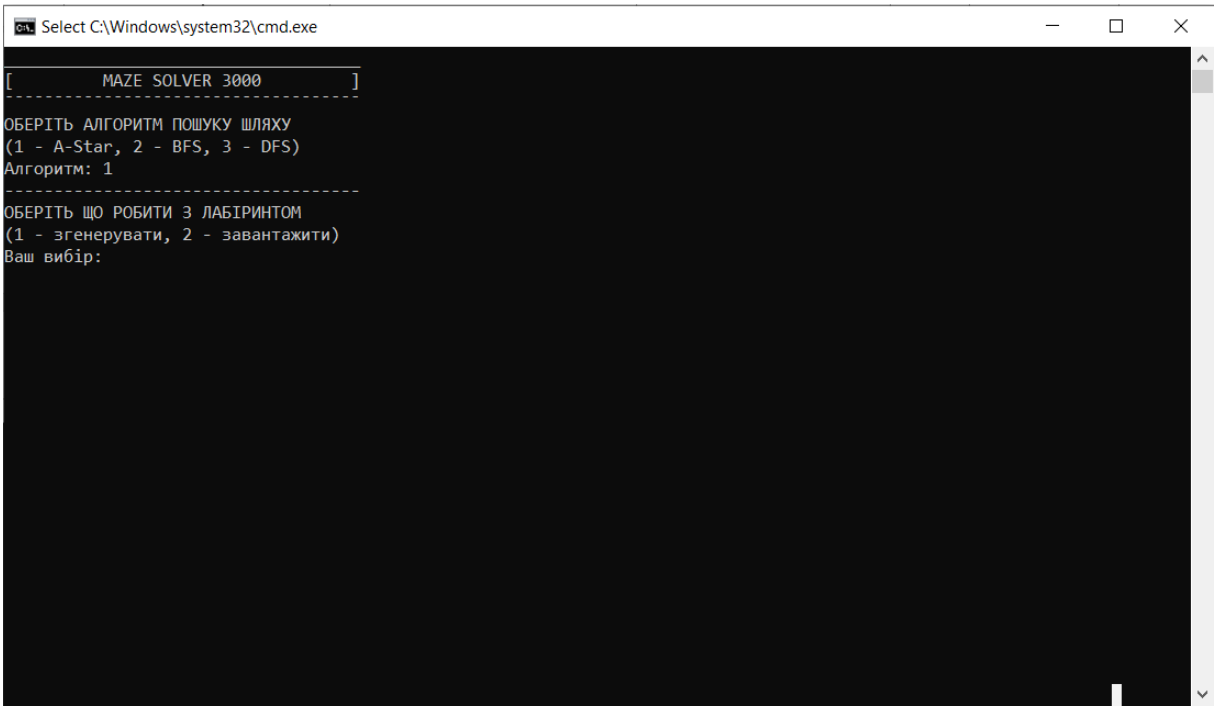
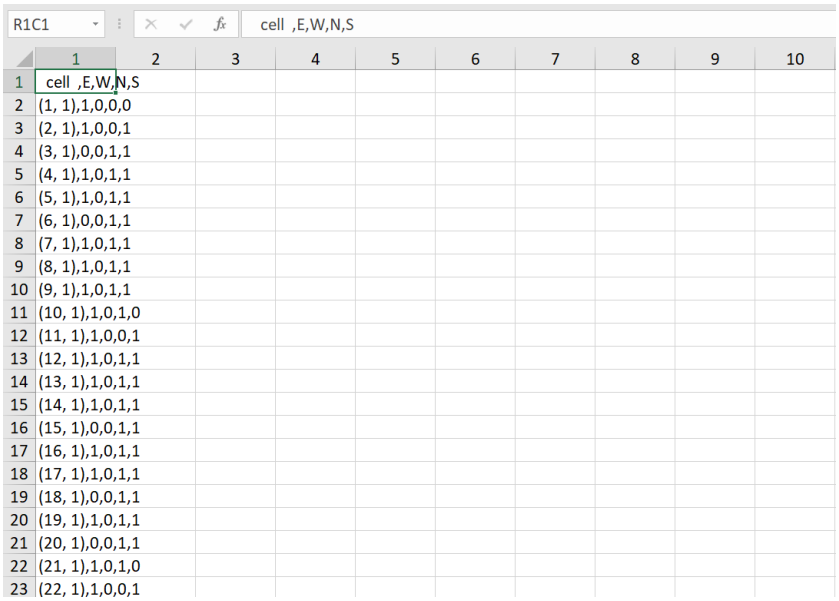


Рис. 2.12 – Вибір лабіринту при первинному налаштуванні



R1C1 | cell ,E,W,N,S

	1	2	3	4	5	6	7	8	9	10
1	cell ,E,W,N,S									
2	(1, 1),1,0,0,0									
3	(2, 1),1,0,0,1									
4	(3, 1),0,0,1,1									
5	(4, 1),1,0,1,1									
6	(5, 1),1,0,1,1									
7	(6, 1),0,0,1,1									
8	(7, 1),1,0,1,1									
9	(8, 1),1,0,1,1									
10	(9, 1),1,0,1,1									
11	(10, 1),1,0,1,0									
12	(11, 1),1,0,0,1									
13	(12, 1),1,0,1,1									
14	(13, 1),1,0,1,1									
15	(14, 1),1,0,1,1									
16	(15, 1),0,0,1,1									
17	(16, 1),1,0,1,1									
18	(17, 1),1,0,1,1									
19	(18, 1),0,0,1,1									
20	(19, 1),1,0,1,1									
21	(20, 1),0,0,1,1									
22	(21, 1),1,0,1,0									
23	(22, 1),1,0,0,1									

Рис. 2.13 – Файл з масивом даних у форматі CSV

Якщо обрано згенерувати новий лабіринт, то з’явиться три додаткових налаштування.

Одне з додаткових налаштувань це можливість обрати розмір лабіринту. Лабіринт може бути будь-якого розміру, це впливає на час його генерування та подальшої обробки (рис. 2.14).

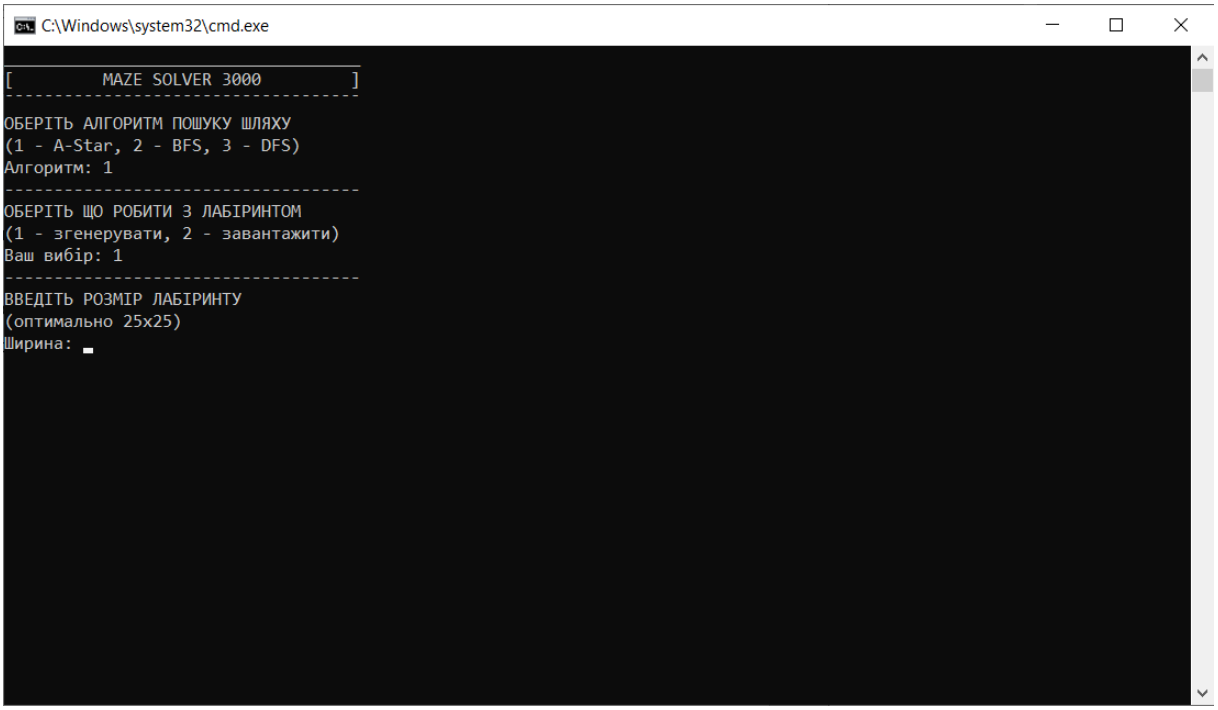


Рис. 2.14 – Вибір розміру лабіринту при первинному налаштуванні

Наступне додаткове налаштування це можливість обрати відсоток зациклювань у лабіринті (рис. 2.15). Зациклювання означає скільки кругових маршрутів, тобто маршрутів де можна не повертаючись назад повернутись на початкову точку, буде в згенерованому лабіринті.

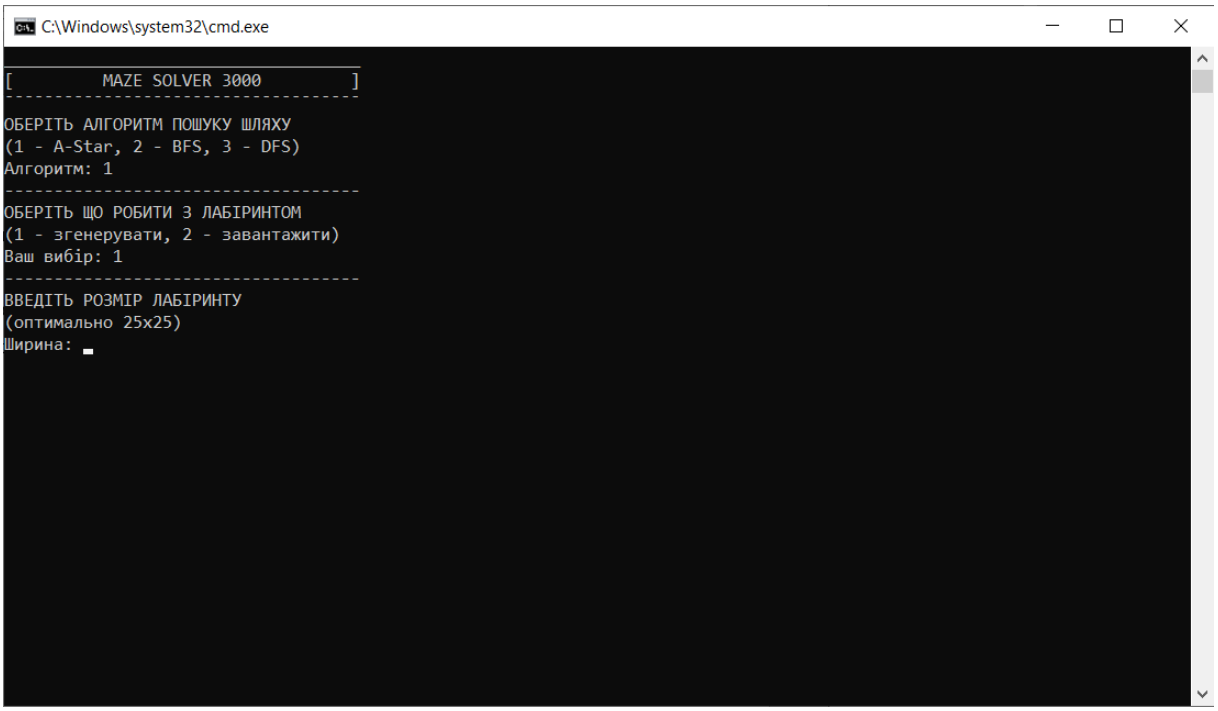


Рис. 2.15 – Вибір зациклювання лабіринту при первинному налаштуванні

Останнє додаткове налаштування дає можливість обрати чи потрібно зберегти у форматі CSV новий згенерований лабіринт (рис. 2.16).

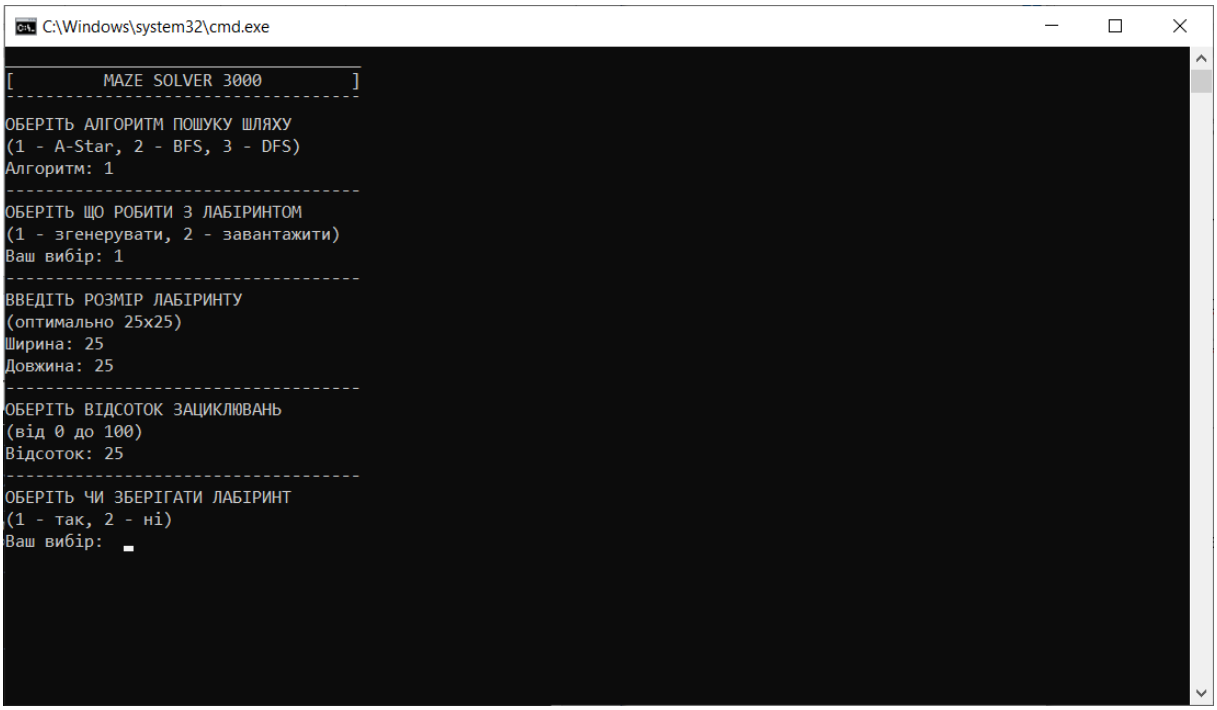


Рис. 2.16 – Можливість зберігати лабіринт при первинному налаштуванні

Наступним налаштуванням є можливість обрати координати початкової точки з якої алгоритм пошуку шляху почне вирішення задачі (рис. 2.17).

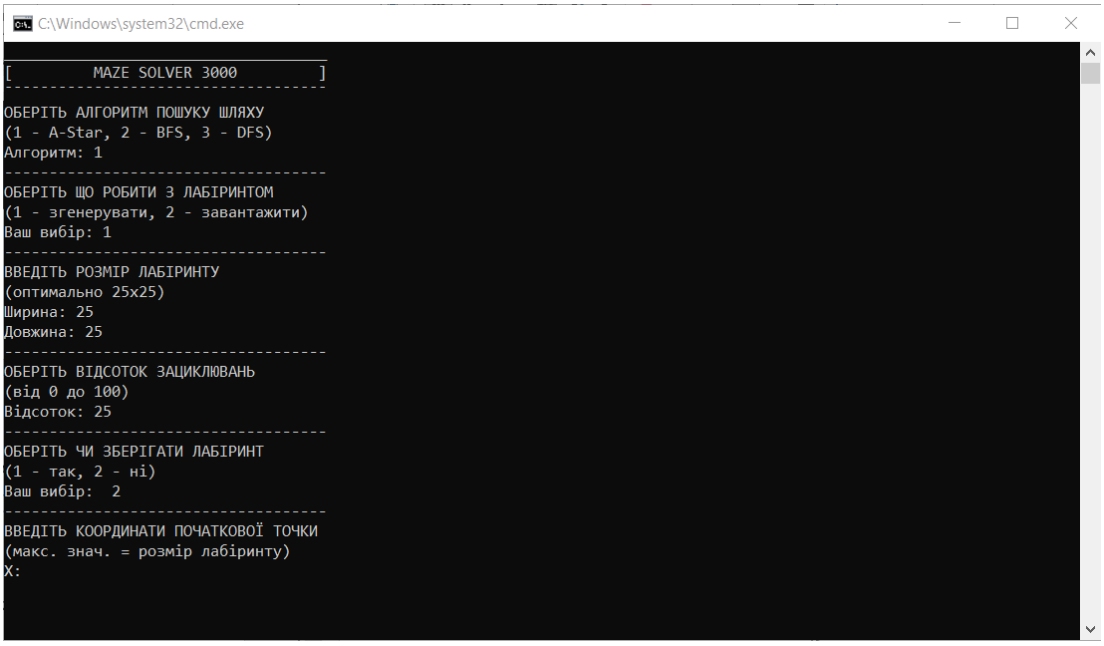


Рис. 2.17 – Вибір початкової позиції при первинному налаштуванні

Останнє налаштування це можливість обрати координати кінцевої точки на якій алгоритм пошуку шляху закінчить вирішувати задачу та відобразить знайдений шлях (рис. 2.18).

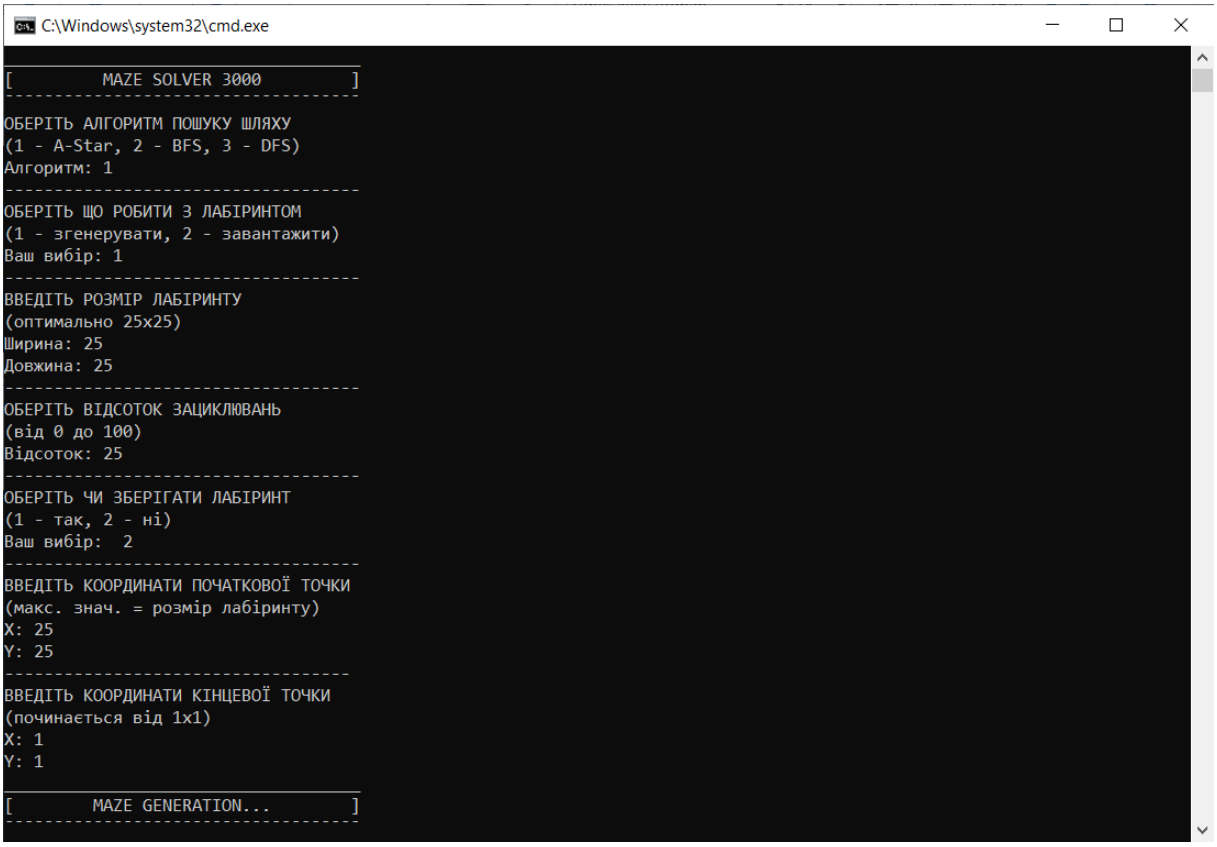


Рис. 2.18 – Вибір кінцевої позиції при первинному налаштуванні

Після завершення налаштувань та натискання кнопки ENTER розпочнеться генерування лабіринту. Після того як лабіринт буде згенеровано його почне вирішувати раніше обраний алгоритм пошуку шляху. На згенерованому лабіринті почнуть з’являтись невеликі червоні квадратики які відображають хід виконання алгоритму рішення (рис. 2.19). Після вирішення задачі, а саме знаходження оптимального шляху проходження лабіринту на рішення алгоритму, почнуть поступово з’являтись зелені квадратики які будуть відображати знайдений шлях (рис. 2.20).

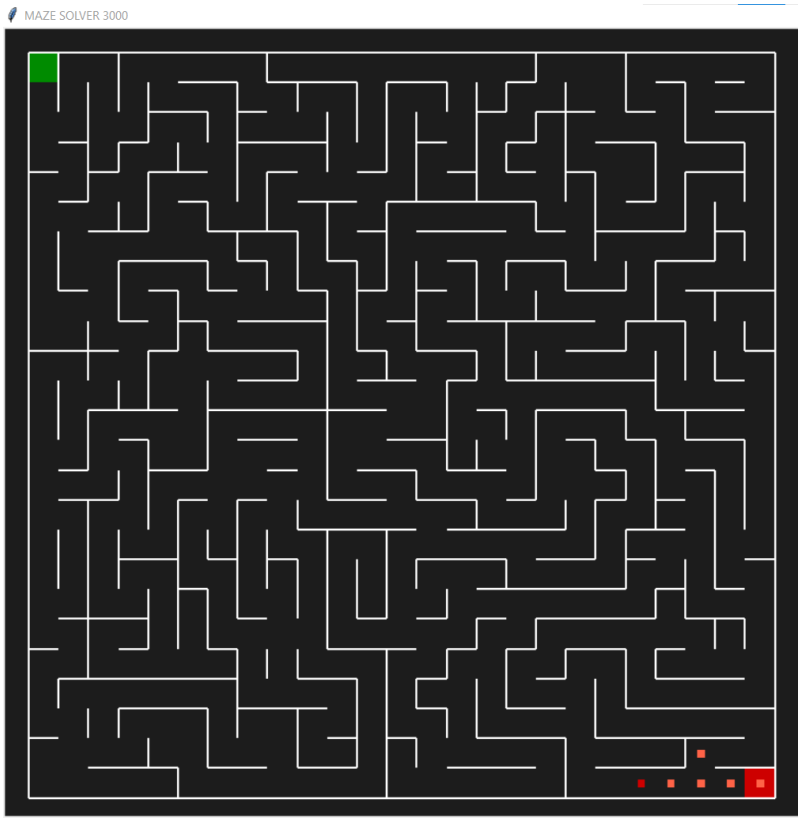


Рис. 2.19 – Початок вирішення згенерованого лабіринту

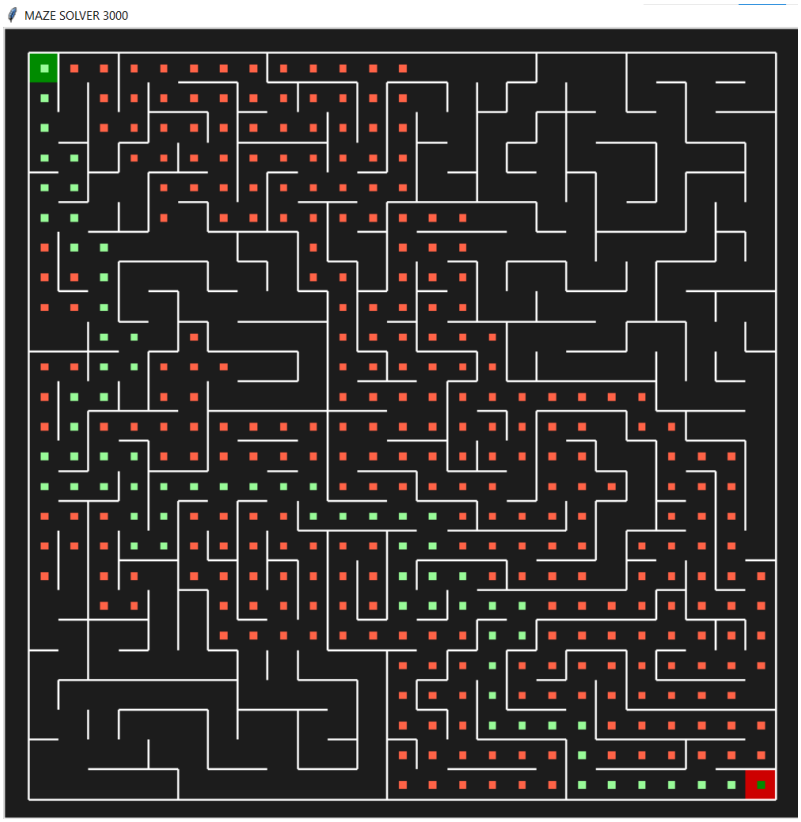


Рис. 2.20 – Кінець вирішення згенерованого лабіринту з відображенням результату

РОЗДІЛ 3. ЕКОНОМІЧНА ЧАСТИНА

3.1. Визначення трудомісткості розробки програмного забезпечення

Задані дані:

1. передбачуване число операторів – 500;
2. коефіцієнт складності програми – 1,65;
3. коефіцієнт корекції програми в ході її розробки – 0,3;
4. годинна заробітна плата програміста, грн/год – 105;

Годинна заробітна плата python-розробника початківця була вирахувана виходячи з даних «Української спільноти програмістів (DOU)». Згідно цього ресурсу заробітна плата розробника такого рівня в середньому дорівнює приблизно 500 доларів США у місяць. При курсі валют НБУ на середину червня 2023 року один американський долар дорівнює 36,94 грн, виходячи з цього можна підрахувати, що середня заробітна плата в гривнях дорівнює 18 470 грн. При восьмигодинному робочому дні (в середньому 176 робочих годин на місяць) середня заробітна плата за годину буде становити 105 грн.

5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,25;

6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,05;

7. вартість машино-години ЕОМ, грн/год – 14,32.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки програмного забезпечення може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д, \text{ людино-годин,} \qquad (3.1)$$

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50 людино-годин);

$t_и$ - витрати праці на дослідження алгоритму рішення задачі;

$t_а$ - витрати праці на розробку блок-схеми алгоритму;

$t_п$ - витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ - витрати праці на налагодження програми на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1 + p), \tag{3.2}$$

де q – передбачуване число операторів;

C – коефіцієнт складності програми;

p – коефіцієнт корекції програми в ході її розробки.

$$Q = 500 \cdot 1.65 \cdot (1 + 0,3) = 1072,5$$

Витрати праці на вивчення опису задачі $t_и$ визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_и = \frac{Q \cdot B}{(75..85) \cdot k}, \text{ людино-годин}, \tag{3.3}$$

де B – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k – коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

$$t_u = \frac{1072,5 \cdot 1,25}{75 \cdot 1,05} = 17,02 \text{ людино-годин}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20...25) \cdot k}, \text{ людино-годин} \quad (3.4)$$

$$t_a = \frac{1072,5}{20 \cdot 1,05} = 51,07 \text{ людино-годин}$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20...25) \cdot k} \text{ людино-годин}, \quad (3.5)$$

$$t_n = \frac{1072.5}{25 \cdot 1,05} = 40,85 \text{ людино-годин}$$

Витрати праці на налагодження програми на ЕОМ:

–за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4..5) \cdot k}, \text{ людино-годин} \quad (3.6)$$

$$t_{отл} = \frac{1072.5}{5 \cdot 1,05} = 204,28 \text{ людино-годин}$$

–за умови комплексного налагодження завдання:

$$t^k_{отл} = 1,5 \cdot t_{отл}, \text{ людино-годин}, \quad (3.7)$$

$$t^{k}_{отл} = 1{,}5 \cdot 204{,}28 = 306{,}42 \text{ людино-годин.}$$

Витрати праці на підготовку документації:

$$t_{д} = t_{др} + t_{до}\text{, людино-годин,} \tag{3.8}$$

де $t_{др}$ – трудомісткість підготовки матеріалів і рукопису;

$$t_{др} = \frac{Q}{15..20 \cdot k}\text{, людино-годин,} \tag{3.9}$$

$$\text{t}_{\text{др}} = \frac{1072{,}5}{20 \cdot 1{,}05} = 51{,}07 \text{ людино-годин.}$$

$t_{до}$ – трудомісткість редагування, печатки й оформлення документації:

$$t_{до} = 0{,}75 \cdot t_{др}\text{, людино-годин,} \tag{3.10}$$

$$\text{t}_{\text{до}} = 0{,}75 \cdot 51{,}07 = 38{,}30 \text{ людино-годин,}$$

$$\text{t}_{\text{д}} = 51{,}07 + 38{,}30 = 89{,}37 \text{ людино-годин.}$$

Розрахуємо трудомісткості розробки ПЗ:

$$\text{t} = 50 + 17{,}02 + 51{,}07 + 40{,}85 + 204{,}28 + 89{,}37 = 452{,}59$$

людино-годин.

3.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ *Кпо* включають витрати на заробітну плату виконавця програми *Зз/п* і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$К_{по} = З_{зп} + З_{мв}, \text{ грн.} \quad (3.11)$$

Заробітна плата виконавців визначається за формулою:

$$З_{зп} = t \cdot C_{пр}, \text{ грн,} \quad (3.12)$$

де t – загальна трудомісткість, людино-годин;

$C_{пр}$ – середня годинна заробітна плата програміста, грн/година.

$$З_{зп} = 452{,}59 \cdot 105 = 47\,521{,}95 \text{ грн.}$$

Вартість машинного часу, необхідного для налагодження програми:

$$З_{мв} = t_{отл} \cdot C_{мч}, \text{ грн,} \quad (3.13)$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год,

$C_{мч}$ – вартість машино-години ЕОМ, грн/год,

$C_{мч}$ = 14,32 грн/год.

$$З_{мв} = 204{,}28 \cdot 14{,}32 = 2\,925{,}28 \text{ грн.}$$

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення ПЗ:

$$К_{по} = 47\ 521{,}95 + 2\ 925{,}28 = 50\ 447{,}23 \text{ грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k \cdot F_p}, \text{ міс.,} \qquad (3.14)$$

де B_k – число виконавців (приймається 1),

F_p – місячний фонд робочого часу (40 годин на тиждень F_p = 176 годин).

$$Т = \frac{452{,}59}{1 \cdot 176} = 2{,}5 \text{ міс.}$$

Висновки: Вартість розробка даного програмного забезпечення, при вартості долару США - 36,94 грн. за долар та середній заробітній платі у 500 доларів США, коштує 50 447,23 грн. Очікувана ймовірна тривалість часу розробки – 2,5 місяця при стандартному 176-годинному робочому місяці (40 робочих годин на тиждень). Цей термін включає в себе час на дослідження та розробку алгоритму розв’язання задачі та розробку дизайну. На розробку додатка буде витрачено 452,59 людино-годин.

ВИСНОВКИ

В даній кваліфікаційній роботі було створено додаток для генерування та вирішення лабіринту за допомогою трьох алгоритмів пошуку шляху A-Star, BFS (Breadth-First Search) та DFS (Depth-First Search) – «MAZE SOLVER 3000».

В першому розділі було розглянуто область застосування додатку, проаналізовано існуючі перспективні рішення об’єкту розробки та обґрунтовано вибір теми розроблюваного додатку.

В другому розділі розглянуто процес розробки додатку з користувацьким інтерфейсом у консолі для налаштування та відрисованому вікні для виводу результатів за допомогою інтегрованого середовища розробки програмного забезпечення IntelliJ IDEA, а також технології та математичні методи використовувані для розробки

У процесі роботи над кваліфікаційною роботою було освоєно:

- Взаємодію із математичними та графічними бібліотеками мови програмування Python.
- Використання математичних методів для розв’язання графів.

Були закріплені знання щодо:

- Створення додатків на інтегрованій мові програмування Python.
- Створення додатків за допомогою інтегрованого середовища програмування IntelliJ IDEA.

У разі необхідності розроблений додаток має можливість проводити подальше вдосконалення та доробку, тобто є відкритим програмним продуктом.

В третьому, економічному, розділі було проведено розрахунки вартості створення розробленого програмного продукту з урахуванням трудомісткості проекту та витрат на його розробку,

Згідно цих розрахунків було виведено, що собівартість розробки подібного додатку становитиме 50 447,23 грн, а на його розробку піде 452,59 годин.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Python for Data Analysis. Data Wrangling with Pandas. NumPy and IPython, 466 pages
2. Fluent Python: Clear, Concise, and Effective Programming 1st Edition, Luciano Ramalho, 792 pages
3. Mastering OOP Python: Build powerful applications with reusable code using OOP design patterns and Python 3.7, by Steven F. Lott, 770 pages
4. Документація Python - http://surl.li/iozph
5. Документація по Python у IntelliJ IDEA - http://surl.li/ivohm
6. Документація по IDE IntelliJ IDEA - http://surl.li/ivoho
7. Побудова лабіринтів за допомогою графів - http://surl.li/ivohr
8. Graph Algorithms: Practical Examples in Apache Spark and Neo4j 1st Edition by Mark Needham and Amy Hodler, 265 pages
9. Що таке алгоритм A* - http://surl.li/ivoht
10. Різниця між алгоритмами BFS та DFS - http://surl.li/ivohv
11. Алгоритм Бектрекінгу - http://surl.li/ivohx
12. Introduction to Graph Theory by Richard J. Trudeau, 224 pages
13. Solving Transport Problems: Towards Green Logistics (Computer Engineering) 1st Edition by Walid Besbes, 272 pages
14. A depth-first search routing algorithm for star graphs and its performance evaluation - http://surl.li/ivohy
15. An Improved multi-objective a-star algorithm for path planning in a large workspace - http://surl.li/ivohz
16. A parallel path-following phase unwrapping algorithm based on a top-down breadth-first search approach - http://surl.li/ivoid
17. Backtracking search optimization algorithm - http://surl.li/ivoid
18. Graph Databases for Beginners - http://surl.li/ivoih
19. Довідник з бібліотеки Python Tkinter - http://surl.li/ivogb
20. Python GUI Programming With Tkinter - http://surl.li/ivoin

ДОДАТОК А

КОД ПРОГРАМИ

Лістинг Launcher.py:

```
from maze import mazeGen, colors, explorer
from Algorithms.A_Star import A_Star
from Algorithms.Breadth_First_Search import BFS
from Algorithms.Depth_First_Search import DFS
import os

print("_____________________________________")
print("[         MAZE SOLVER 3000          ]")
print("_____________________________________")

print("ОБЕРІТЬ АЛГОРИТМ ПОШУКУ ШЛЯХУ")
print("(1 - A-Star, 2 - BFS, 3 - DFS)")
algorithm = int(input("Алгоритм: "))
print("-----------------------------------")

print("ОБЕРІТЬ ЩО РОБИТИ З ЛАБІРИНТОМ")
print("(1 - згенерувати, 2 - завантажити)")
choise = int(input("Ваш вибір: "))
print("-----------------------------------")

maze_width = 0
maze_length = 0
loop = 0
filename = None
save = False


if choise == 1:
    print("ВВЕДІТЬ РОЗМІР ЛАБІРИНТУ")
    print("(оптимально 25x25)")
    maze_width = int(input("Ширина: "))
    maze_length = int(input("Довжина: "))
    print("-----------------------------------")

    print("ОБЕРІТЬ ВІДСОТОК ЗАЦИКЛЮВАНЬ")
    print("(від 0 до 100)")
    loop = int(input("Відсоток: "))
    print("-----------------------------------")

    print("ОБЕРІТЬ ЧИ ЗБЕРІГАТИ ЛАБІРИНТ")
    print("(1 - так, 2 - ні)")
    chek = int(input("Ваш вибір:  "))
```

```
    if chek == 1: save = True
    elif chek == 2: save = False
    print("-----------------------------------")

elif choise == 2:
    print("ОБЕРІТЬ ФАЙЛ ЛАБІРИНТУ")
    print("(filename.csv)")
    filename = input("Ім'я файлу: ")
    filename = os.path.basename(filename)
    print("-----------------------------------")

print("ВВЕДІТЬ КООРДИНАТИ ПОЧАТКОВОЇ ТОЧКИ")
print("(макс. знач. = розмір лабіринту)")
maze_start_x = int(input("X: "))
maze_start_y = int(input("Y: "))
print("-----------------------------------")

print("ВВЕДІТЬ КООРДИНАТИ КІНЦЕВОЇ ТОЧКИ")
print("(починається від 1x1)")
maze_end_x = int(input("X: "))
maze_end_y = int(input("Y: "))

print("_____________________________________")
print("[        MAZE GENERATION...         ]")
print("‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾")

maze = mazeGen(maze_length, maze_width)
maze.CreateMaze(maze_end_y, maze_end_x, loopPercent=loop,
saveMaze=save, loadMaze=filename)

explorer(maze, maze_start_y, maze_start_x, color=colors.red,
shape='square', filled=True)
start = explorer(maze, maze_start_y, maze_start_x,
footprints=True, color=colors.red, filled=False)
end = explorer(maze, maze_end_y, maze_end_x, footprints=True,
color=colors.green, filled=False, goal=(maze_start_y,
maze_start_x))

if algorithm == 1:
    asFinder, asWay, fwdWay = A_Star(maze, (maze_start_y,
maze_start_x))
    maze.tracePath({start: asFinder}, delay=200)
    maze.tracePath({end: asWay}, delay=200)

if algorithm == 2:
    bfsFinder, bfsWay, fwdWay = BFS(maze, (maze_start_y,
maze_start_x))
    maze.tracePath({start: bfsFinder}, delay=200)
    maze.tracePath({end: bfsWay}, delay=200)

if algorithm == 3:
    dfsFinder, dfsWay, fwdWay = DFS(maze, (maze_start_y,
maze_start_x))
```

```
    maze.tracePath({start: dfsFinder}, delay=200)
    maze.tracePath({end: dfsWay}, delay=200)

maze.run()
```

Лістинг A_Star.py:

```
from queue import PriorityQueue

def herstc(point_1, point_2):
    x1, y1 = point_1
    x2, y2 = point_2

    return (abs(x1 - x2) + abs(y1 - y2))

def A_Star(maze, maze_start=None):

    PrQu = PriorityQueue()
    PrQu.put((herstc(maze_start, maze._goal),
              herstc(maze_start, maze._goal), maze_start))

    asWay = {}

    dst_sp = {row: float("inf") for row in maze.grid}
    dst_sp[maze_start] = 0


    dst_sum = {row: float("inf") for row in maze.grid}
    dst_sum[maze_start] = herstc(maze_start, maze._goal)

    asFinder = [maze_start]

    while not PrQu.empty():
        currCell = PrQu.get()[2]
        asFinder.append(currCell)

        if currCell == maze._goal:
            break
        for d in 'ESNW':
            if maze.maze_map[currCell][d]:
                if d == 'E': d_x, d_y = 0, 1
                elif d == 'W': d_x, d_y = 0, -1
                elif d == 'N': d_x, d_y = -1, 0
                elif d == 'S': d_x, d_y = 1, 0

                childCell = (currCell[0] + d_x, currCell[1] + d_y)

                temp_dst_sp = dst_sp[currCell] + 1
                temp_dst_sum = temp_dst_sp
                   + herstc(childCell, maze._goal)
```

```
                if temp_dst_sum < dst_sum[childCell]:
                    asWay[childCell] = currCell
                    dst_sp[childCell] = temp_dst_sp
                    dst_sum[childCell] = temp_dst_sum
                    PrQu.put((dst_sum[childCell],
                    herstc(childCell, maze._goal), childCell))

    fwdWay = {}
    cell = maze._goal
    while cell != maze_start:
        fwdWay[asWay[cell]] = cell
        cell = asWay[cell]
    return asFinder, asWay, fwdWay
```

Лістинг Breadth_First_Search.py:

```
from collections import deque

def BFS(maze, maze_start=None):
    if maze_start is None: maze_start = (maze.rows, maze.cols)

    explored = set()

    frontier = deque()
    frontier.append(maze_start)


    bfsWay = {}
    bfsFinder=[]

    while len(frontier) > 0:
        currCell=frontier.popleft()

        if currCell==maze._goal:
            break
        for d in 'ESNW':
            if maze.maze_map[currCell][d] == True:
                if d == 'E': d_x, d_y = 0, 1
                elif d == 'W': d_x, d_y = 0, -1
                elif d == 'N': d_x, d_y = -1, 0
                elif d == 'S': d_x, d_y = 1, 0

                childCell = (currCell[0] + d_x, currCell[1] + d_y)

                if childCell in explored: continue
                frontier.append(childCell)
                explored.add(childCell)
                bfsWay[childCell] = currCell
                bfsFinder.append(childCell)

    fwdWay = {}
```

```
    cell = maze._goal
    while cell != maze_start:
        fwdWay[bfsWay[cell]] = cell
        cell = bfsWay[cell]
    return bfsFinder, bfsWay, fwdWay
```

Лістинг Depth_First_Search.py:

```
def DFS(maze, maze_start=None):
    if maze_start is None: maze_start=(maze.rows, maze.cols)

    explored=[maze_start]
    frontier=[maze_start]

    dfsWay={}
    dfsFinder=[]

    while len(frontier) > 0:
        currCell = frontier.pop()
        dfsFinder.append(currCell)

        if currCell == maze._goal:
            break
        for d in 'ESNW':
            if maze.maze_map[currCell][d] == True:
                if d == 'E': d_x, d_y = 0, 1
                elif d == 'W': d_x, d_y = 0, -1
                elif d == 'N': d_x, d_y = -1, 0
                elif d == 'S': d_x, d_y = 1, 0

                childCell = (currCell[0] + d_x, currCell[1] + d_y)

                if childCell in explored: continue
                explored.append(childCell)
                frontier.append(childCell)
                dfsWay[childCell]=currCell

    fwdWay = {}
    cell = maze._goal
    while cell != maze_start:
        fwdWay[dfsWay[cell]] = cell
        cell = dfsWay[cell]
    return dfsFinder, dfsWay, fwdWay
```

Лістинг MazeCreator.py:

```
import random, datetime, csv, os
from tkinter import *
from enum import Enum

class colors(Enum):

    dark=('gray11','white')
    red=('red3','tomato')
    green=('green4','pale green')

class explorer:

    def __init__(self, parentMaze, x=None, y=None, goal=None,
                 footprints=False, filled=False, shape='square',
                 color:colors=colors.dark):

        self._parentMaze = parentMaze
        self._parentMaze._agents.append(self)

        if goal is None: self.goal=self._parentMaze._goal
        else: self.goal=goal

        self.footprints = footprints
        self.filled = filled
        self.shape = shape
        self.color = color

        self._orient=0
        self._body=[]


        if x is None:x=parentMaze.rows
        self.x=x

        if y is None:y=parentMaze.cols
        self.y=y

        self.position=(self.x,self.y)

    def x(self):
        return self._x

    def set_x(self, newX):
        self._x = newX

    x = property(x, set_x)

    def y(self):
        return self._y
```

```
    def set_y(self, newY):
        self._y = newY
        w = self._parentMaze._cell_width
        x = self.x * w - w + self._parentMaze._LabWidth
        y = self.y * w - w + self._parentMaze._LabWidth

        if self.filled:
            self._coord = (y, x, y + w, x + w)
        else:
            self._coord = (y + w / 2.5, x + w / 2.5,
                           y + w / 2.5 + w / 4,
                           x + w / 2.5 + w / 4)

        if hasattr(self, '_head'):

            self._parentMaze._canvas.itemconfig(self._head,
                        fill=self.color.value[1], outline="")
            self._parentMaze._canvas.tag_raise(self._head)

       try:
            self._parentMaze._canvas.tag_lower(self._head, 'ov')
            except: pass

            self._body.append(self._head)

             if not self.filled:
                self._head = self._parentMaze.
                     _canvas.create_rectangle(*self._coord,
                          fill=self.color.value[0], outline='')




                try:
                    self._parentMaze._canvas.tag_lower
                          (self._head, 'ov')

                except:
                    pass

            else:
                self._head = self._parentMaze.
                    _canvas.create_rectangle(*self._coord,
                          fill=self.color.value[0], outline='')

                try:
                    self._parentMaze._canvas.tag_lower
                          (self._head, 'ov')

                except:
                    pass
```

```
                self._parentMaze._redrawCell(self.x, self.y,
                                             theme=self._parentMaze.theme)
        else:
            self._head = self._parentMaze.
                 _canvas.create_rectangle(*self._coord,
                          fill=self.color.value[0], outline='')

             try:
                self._parentMaze._canvas.tag_lower
                                             (self._head, 'ov')
            except:
                pass

            self._parentMaze._redrawCell(self.x, self.y,
                                         theme=self._parentMaze.theme)

    y = property(y, set_y)

class mazeGen:

    def __init__(self,rows=None,cols=None):

        self.maze_map={}
        self.rows=rows
        self.cols=cols

        self.path={}
        self._agents=[]

        self.grid=[]



    def grid(self):
        return self._grid

    def set_grid(self, n):
        self._grid = []
        y = 0
        for _ in range(self.cols):
            x = 1
            y += 1
            for _ in range(self.rows):
                self._grid.append((x, y))
                self.maze_map[x, y] = {'E': 0, 'W': 0,
                                       'N': 0, 'S': 0}
                x += 1

    grid = property(grid, set_grid)

    def update_direction(self, x, y, direction):
        if direction == 'E':
```

```
            self.maze_map[x, y]['E'] = 1
            if y + 1 <= self.cols:
                self.maze_map[x, y + 1]['W'] = 1
        elif direction == 'W':
            self.maze_map[x, y]['W'] = 1
            if y - 1 > 0:
                self.maze_map[x, y - 1]['E'] = 1
        elif direction == 'N':
            self.maze_map[x, y]['N'] = 1
            if x - 1 > 0:
                self.maze_map[x - 1, y]['S'] = 1
        elif direction == 'S':
            self.maze_map[x, y]['S'] = 1
            if x + 1 <= self.rows:
                self.maze_map[x + 1, y]['N'] = 1

    def CreateMaze(self, x=1, y=1, loopPercent=0, saveMaze=False,
                   loadMaze=None, theme:colors=colors.dark):

        _stack=[]
        _closed=[]

        self.theme=theme

        self._goal=(x,y)

        def blockedPassage(cell):

            n = []
            x, y = cell

             if self.maze_map[cell]['E'] == 0 and (x, y + 1)
                         in self.grid: n.append((x, y + 1))

            if self.maze_map[cell]['W'] == 0 and (x, y - 1)
                         in self.grid: n.append((x, y - 1))

            if self.maze_map[cell]['N'] == 0 and (x - 1, y)
                         in self.grid: n.append((x - 1, y))

            if self.maze_map[cell]['S'] == 0 and (x + 1, y)
                         in self.grid: n.append((x + 1, y))

            return n

        def wallRemover(cell1, cell2):

            if cell1[0]==cell2[0]:
                if cell1[1]==cell2[1]+1:
                    self.maze_map[cell1]['W']=1
                    self.maze_map[cell2]['E']=1
                else:
```

```
                self.maze_map[cell1]['E']=1
                self.maze_map[cell2]['W']=1

        else:
            if cell1[0]==cell2[0]+1:
                self.maze_map[cell1]['N']=1
                self.maze_map[cell2]['S']=1
            else:
                self.maze_map[cell1]['S']=1
                self.maze_map[cell2]['N']=1

    def voidCleaner(cell1, cell2):

        ans=False

        if cell1[0]==cell2[0]:
            if cell1[1]>cell2[1]: cell1,cell2=cell2,cell1
            if self.maze_map[cell1]['S']==1
               and self.maze_map[cell2]['S']==1:

                if (cell1[0]+1,cell1[1]) in self.grid
and self.maze_map[(cell1[0]+1,cell1[1])]['E']==1: ans= True

            if self.maze_map[cell1]['N']==1
               and self.maze_map[cell2]['N']==1:

                if (cell1[0]-1,cell1[1]) in self.grid
and self.maze_map[(cell1[0]-1,cell1[1])]['E']==1: ans= True



        else:
            if cell1[0]>cell2[0]: cell1,cell2=cell2,cell1
            if self.maze_map[cell1]['E']==1
               and self.maze_map[cell2]['E']==1:

                if (cell1[0],cell1[1]+1) in self.grid
and self.maze_map[(cell1[0],cell1[1]+1)]['S']==1: ans= True

            if self.maze_map[cell1]['W']==1
               and self.maze_map[cell2]['W']==1:

                if (cell1[0],cell1[1]-1) in self.grid
and self.maze_map[(cell1[0],cell1[1]-1)]['S']==1: ans= True

        return ans

    if not loadMaze:

        _stack = [(x, y)]
        _closed = [(x, y)]
        bias = 0
```

```
        def add_cell_to_stack(dx, dy):
            nonlocal x, y
            self.path[x + dx, y + dy] = x, y
            x += dx
            y += dy
            _closed.append((x, y))
            _stack.append((x, y))

        while len(_stack) > 0:
            cell = []
            bias += 1

            if (x, y + 1) not in _closed and (x, y + 1)
               in self.grid: cell.append("E")

            if (x, y - 1) not in _closed and (x, y - 1)
               in self.grid: cell.append("W")

            if (x + 1, y) not in _closed and (x + 1, y)
               in self.grid: cell.append("S")

            if (x - 1, y) not in _closed and (x - 1, y)
               in self.grid: cell.append("N")

            if len(cell) > 0:
                current_cell = random.choice(cell)
                if current_cell == "E":
                    self.update_direction(x, y, 'E')
                    add_cell_to_stack(0, 1)

                elif current_cell == "W":
                    self.update_direction(x, y, 'W')
                    add_cell_to_stack(0, -1)

                elif current_cell == "N":
                    self.update_direction(x, y, 'N')
                    add_cell_to_stack(-1, 0)

                elif current_cell == "S":
                    self.update_direction(x, y, 'S')
                    add_cell_to_stack(1, 0)

            else:
                x, y = _stack.pop()

        def process_cells(cells, count_percent):
            random.shuffle(cells)
            cell_count = len(cells)
            count = cell_count // 3 * count_percent // 100
            i = 0
            while count < count_percent:
                if len(blockedPassage(cells[i])) > 0:
                    cell = random.choice
```

```
                        (blockedPassage(cells[i]))

                    if not voidCleaner(cell, cells[i]):
                        wallRemover(cell, cells[i])
                        count += 1

                    i += 1

                else: i += 1

                if i == cell_count: break

        if loopPercent != 0:

            pathCells = [(self.rows, self.cols)]

            while pathCells[-1] != (self.rows, self.cols):
                pathCells.append(self.path[pathCells[-1]])

            notPathCells = [i for i in self.grid
                            if i not in pathCells]

            random.shuffle(pathCells)
            random.shuffle(notPathCells)

            count1 = len(pathCells) // 3 * loopPercent // 100
            count2 = len(notPathCells) // 3 *
                                          loopPercent // 100



            i = 0
            count = 0

            for cell in pathCells:
                if count >= count1:
                    break
                if len(blockedPassage(cell)) > 0
                      and not voidCleaner(cell, pathCells[i]):

                      wallRemover(random.choice(
                            blockedPassage(cell)), cell)

                      count += 1

            if len(notPathCells) > 0:

                count = 0

                for cell in notPathCells:
                    if count >= count2: break
                    if len(blockedPassage(cell)) > 0
```

```
                    and not voidCleaner(cell, notPathCells[i]):
                      wallRemover(random.choice
                                  (blockedPassage(cell)), cell)

                        count += 1

        else:

            with open(loadMaze, 'r') as f:

                lines = f.readlines()
                last = lines[-1]
                c = last.strip().strip('"()').split(',')

                self.rows = int(c[0])
                self.cols = int(c[1])
                self.grid = []

                csv_lines = lines[1:]

                for line in csv_lines:
                    c = line.strip().strip('()').split(',')
                    coordinates = (int(c[0]), int(c[1]))
                    self.maze_map[coordinates] = {
                        'E': int(c[2]), 'W': int(c[3]),
                        'N': int(c[4]), 'S': int(c[5])
                    }

        self._drawMaze(self.theme)
        explorer(self, *self._goal, shape='square', filled=True,
                                     color=colors.green)

        if saveMaze:
            dt_string = datetime.datetime.now().strftime
                                            ("%Y-%m-%d (%H-%M-%S)")

            filename = f'mazeGen--{dt_string}.csv'

            with open(filename, 'w', newline='') as f:

                writer = csv.writer(f)
                writer.writerow(['  cell  ', 'E', 'W', 'N', 'S'])

                for k, v in self.maze_map.items(): entry = [k]

                    for i in v.values(): entry.append(i)

                    writer.writerow(entry)

                f.seek(0, os.SEEK_END)
                f.seek(f.tell() - 2, os.SEEK_SET)
                f.truncate()
```

```
    def _drawMaze(self,theme):

        self._LabWidth = 26
        self._win = Tk()
        self._win.state('zoomed')
        self._win.title('MAZE SOLVER 3000')

        scr_width=self._win.winfo_screenwidth()
        scr_height=self._win.winfo_screenheight()

        self._win.geometry(f"{scr_width}x{scr_height}+0+0")
        self._canvas = Canvas(width=scr_width, height=scr_height,
                                                bg=theme.value[0])
        self._canvas.pack(expand=YES, fill=BOTH)

        k_values = {
            (95, 95): 0, (80, 80): 1,
            (70, 70): 1.5, (50, 50): 2,
            (35, 35): 2.5, (22, 22): 3
        }

        k = k_values.get((self.rows, self.cols), 3.25)

        self._cell_width = round(min((
        (scr_height - self.rows - k * self._LabWidth)/(self.rows)),
        ((scr_width - self.cols - k * self._LabWidth)/(self.cols)),
         90), 3)

        if self._win is not None:

            if self.grid is not None:

                for cell in self.grid:
                    x, y = cell
                    w = self._cell_width
                    x = x * w - w + self._LabWidth
                    y = y * w - w + self._LabWidth

                    directions = {'E': (y + w, x, y + w, x + w),
                                  'W': (y, x, y, x + w),
                                  'N': (y, x, y + w, x),
                                  'S': (y, x + w, y + w, x + w)}

                    for direction, coords in directions.items():
                        if not self.maze_map[cell][direction]:
                            l = self._canvas.create_line(*coords,
                            width=2, fill=theme.value[1], tag='line')

    def _redrawCell(self, x, y, theme):

        w = self._cell_width

        cell = (x, y)
```

```
        x = x * w - w + self._LabWidth
        y = y * w - w + self._LabWidth

        directions = {'E': (y + w, x, y + w, x + w),
                      'W': (y, x, y, x + w),
                      'N': (y, x, y + w, x),
                      'S': (y, x + w, y + w, x + w)}

        for direction, coords in directions.items():
            if not self.maze_map[cell][direction]:
                self._canvas.create_line(*coords, width=2,
                     fill=theme.value[1])

    _tracePathList = []

    def _tracePathSingle(self,a,p,kill,showMarked,delay):

        if (a.x,a.y)==(a.goal):
            del mazeGen._tracePathList[0][0][a]


            if mazeGen._tracePathList[0][0]=={}:
                del mazeGen._tracePathList[0]

                if len(mazeGen._tracePathList)>0:
                    self.tracePath(mazeGen._tracePathList[0][0],
                         kill=mazeGen._tracePathList[0][1],
                         delay=mazeGen._tracePathList[0][2])

            return

        if isinstance(p, dict):
            if not p: del mazeGen._tracePathList[0][0][a]

                return

                move_actions = {
                    (0, 1): 1, (0, -1): 3,
                    (-1, 0): 0, (1, 0): 2,
                }

                dx, dy = new[0] - old[0], new[1] - old[1]
                mov = move_actions.get((dx, dy))

                if mov is not None:
                    rotation_count = (mov - o) % 4
                    clockwise_rotations = [2, 1, 3, 2, 1, 2, 3]

                    if rotation_count in {1, 3}: a._rotate
                         (clockwise_rotations[rotation_count - 1])

                    elif rotation_count == 2: a._RCW()
```

```
                a.x, a.y = p[(a.x, a.y)]

            else: del p[(a.x, a.y)]

        else: a.x, a.y = p[(a.x, a.y)]

    if (type(p)==list):

        if(len(p)==0):
            del mazeGen._tracePathList[0][0][a]

            if mazeGen._tracePathList[0][0]=={}:
                del mazeGen._tracePathList[0]

                if len(mazeGen._tracePathList)>0:
                  self.tracePath(mazeGen._tracePathList[0][0],
                                 kill=mazeGen._tracePathList[0][1],
                                 delay=mazeGen._tracePathList[0][2])
            return


            move_actions = {
                2: a._RCW, -2: a._RCW,
                1: a._RCW, -1: a._RCCW,
                3: a._RCCW, -3: a._RCW
            }

            if old[0] == new[0]:

                if old[1] > new[1]: mov = 3; mov = 1

            else:

                if old[0] > new[0]: mov = 0

                else: mov = 2

            if mov-o in move_actions: move_actions[mov-o]()

            elif mov == o: a.x, a.y = p[0]
                del p[0]

        else: a.x,a.y=p[0]
            del p[0]

    self._win.after(delay,

     self._tracePathSingle,a,p,kill,showMarked,delay)

def tracePath(self,d,kill=False,delay=300,showMarked=False):
    self._tracePathList.append((d,kill,delay))
    if mazeGen._tracePathList[0][0]==d:
```

```
            for a,p in d.items():
                if a.goal!=(a.x,a.y) and len(p)!=0:
                    self._tracePathSingle(a,p,kill,
                                          showMarked,delay)
    def run(self): self._win.mainloop()
```

ДОДАТОК Б

ВІДГУК

керівника економічного розділу на кваліфікаційну роботу бакалавра
на тему:
«Реалізація штучного інтелекту для проходження лабіринту мовою Python»
студента групи 122-20-ск-1 Бернацького Микити Дмитровича

ДОДАТОК В

ПЕРЕЛІК ФАЙЛІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Кваліфікаційна робота.docx	Пояснювальна записка до дипломного проекту. Документ Word.
Кваліфікаційна робота.pdf	Пояснювальна записка до дипломного проекту в форматі PDF
Програма	
maze-solver.zip	Архів. Містить коди програми і скомпільовану програму
Презентація	
Презентація.ppt	Презентація дипломного проекту