НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

**КУРСОВА РОБОТА**

з дисципліни «Основи програмування. Курсова робота»

на тему: «Пошук маршруту в лабіринті»

Студента групи ТВ-31

Фельчина М.Б.

(прізвище, ініціали)

Керівник доц., к.т.н. Шпурик В.В.

(науковий ступінь, посада, прізвище, ініціали)

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_ Оцінка ECTS \_\_\_\_\_

Члени комісії:

\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_доц., к.т.н. Шпурик В.В.\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_асистент Голець В.О.\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Київ- 2024 рік

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

Інститут Навчально-науковий інститут атомної та теплової енергетики

(повна назва)

Кафедра Інженерії програмного забезпечення в енергетиці

(повна назва)

Спеціальність 121 Інженерія програмного забезпечення

**З А В Д А Н Н Я**

**НА КУРСОВУ РОБОТУ СТУДЕНТУ**

Фельчину Миколі Борисовичу

1. Тема роботи Програмна реалізація логічної головоломки «Пошук маршруту в лабіринті».

керівник курсової роботи – Голець Владислав Олександрович.

1. Строк подання студентом роботи 7 червня 2024 року.
2. Вихідні дані до роботи: середовище розробки – Visual Studio Code.
3. Зміст розрахунково-пояснювальної записки – файловий і користувацький ввід та вивід; алгоритми пошуку; перевірки вводу; візуалізація алгоритмів.

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів виконання курсової роботи | Строк виконання етапів роботи | Примітк**и** |
| 1 | Затвердження обраної теми роботи | 1 тиждень |  |
| 2 | Вивчення та аналіз задачі | 2 – 3 тижні |  |
| 3 | Розробка алгоритмів реалізації | 4 – 5 тижні |  |
| 4 | Програмна реалізація роботи | 6 – 10 тижні |  |
| 5 | Тестування програми | 10 – 11 тижні |  |
| 6 | Вдосконалення коду програми | 12 – 14 тижні |  |
| 7 | Оформлення пояснювальної записки | 15 – 16 тижні |  |
| 8 | Захист | 17 – 18 тижні |  |

Студент \_\_\_\_\_\_\_\_\_\_ \_\_Фельчин Микола \_

(підпис) (ім’я, прізвище)

Керівник курсової роботи \_\_\_\_\_\_\_\_\_\_ \_\_Владислав Голець\_\_

(підпис) (ім’я, прізвище)

**АНОТАЦІЯ**

Програма «FindRouteInMaze» дозволяє знайти найкоротший маршрут в лабіринті будь-якого рівня складності за оптимальний час. Програма використовує декілька алгоритмів, таких як пошук у ширину, пошук у ширину з двох кінців та мурашиний.

Перші два алгоритми дають однаковий результат та працюють за схожим принципом. Принцип їх роботи простіше пояснити, як бак з водою, що перевернувся та на кожній черговій ітерації затоплює сусідні клітинки [2]. Відмінність алгоритмів полягає в умові зупинення пошуку та кількості баків. Як можна зрозуміти з назви, пошук з двох кінців еквівалентний двом бакам з водою, у той час, як звичайний пошук у ширину — одному. Якщо говорити про умову зупинення, то пошук у ширину буде продовжуватись, допоки не затопить всі доступні клітинки або не дійде до кінця [2]. Зі свого боку, пошук з двох кінців буде продовжуватись, допоки всі доступні клітинки не затоплені або вода з двох кінців не перетнеться.

Останній неописаний алгоритм, що використовується у програмі, а саме мурашиний, працює за відмінним принципом від попередніх двох алгоритмів. Принцип можна описати простим відомим правилом. Правилом лівої руки. Суть правила полягає у тому, що при проходженні лабіринту ліва рука має контактувати зі стіною лабіринту [1]. Тобто, потрібно почати лабіринт тримаючи ліву руку на лівій стіні та рухатися вздовж стінки, тримаючи руку на ній. При цьому пріоритет дій такий: повернути наліво, йти прямо, повернути направо [1]. При умові що у лабіринті є шлях до виходу, слідуючи цим правилам, можна знайти шлях до виходу. Але результат не гарантовано буде найкоротшим шляхом.

Код програми написаний мовою програмування C++ та реалізований у середовищі розробки Visual Studio Code за допомогою компілятора Mingw-w64 1.23.0. Зчитування даних можливо як з консолі, так і з окремого файлу.

Обсяг пояснювальної записки 84 аркушів, кількість ілюстрацій - 9 та 3 додатки.

**ANNOTATION**

FindRouteInMaze allows you to find the shortest route in a maze of any complexity level at the optimal time. The program uses several algorithms, such as breadth-first search, breadth-first search from both ends, and ant.

The first two algorithms give the same result and work on a similar principle. It is easier to explain how they work as an overturned water tank that floods neighboring squares at each iteration [2]. The difference between the algorithms lies in the condition for stopping the search and the number of tanks. As the name implies, a two-end search is equivalent to two water tanks, while a regular BFS is equivalent to one. Speaking about the stopping condition, the BFS will continue until it floods all available squares or reaches the end [2]. The BFS from both ends will continue until all available squares are flooded or the water at both ends crosses.

The last undescribed algorithm used in the program, the ant algorithm, works on a different principle from the previous two algorithms. The principle can be described by a simple, well-known rule. The left-hand rule. The essence of the rule is that when passing the maze, the left hand should be in contact with the maze wall [1]. That is, you need to start the labyrinth with your left hand on the left wall and move along the wall, keeping your hand on it. The priority of actions is as follows: turn left, go straight, turn right [1]. Provided that there is a way out of the maze, by following these rules, you can find a way out. But the result will not be guaranteed to be the shortest path.

The program code is written in C++ and implemented in the Visual Studio Code development environment using the Mingw-w64 1.23.0 compiler. Data can be read both from the console and from a file.

The volume of the explanatory note is 84 sheets, the number of illustrations is 9, and there are 3 appendices.

**ЗМІСТ**

[ВСТУП 6](#_Toc168528783)

[1 АНАЛІЗ ЗАВДАННЯ 7](#_Toc168528784)

[2 ПРОГРАМНА РЕАЛІЗАЦІЯ 11](#_Toc168528785)

[2.1 Початкові рішення 11](#_Toc168528786)

[2.2 Діаграми 12](#_Toc168528787)

[2.3 Написання коду та опис програми 14](#_Toc168528788)

[2.4 Оптимізація, тестування та налагодження програми 18](#_Toc168528789)

[2.5 Лабіринти та результати 19](#_Toc168528790)

[3 КЕРІВНИЦТВО КОРИСТУВАЧА 21](#_Toc168528791)

[ВИСНОВКИ 24](#_Toc168528792)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 26](#_Toc168528793)

[ДОДАТОК 1 27](#_Toc168528794)

[Специфікація 27](#_Toc168528795)

[ДОДАТОК 2 30](#_Toc168528796)

[Код програми 30](#_Toc168528797)

[ДОДАТОК 3 81](#_Toc168528798)

[Опис програми 81](#_Toc168528799)

# ВСТУП

Актуальність: алгоритми пошуку найкоротшого маршруту мають важливе значення та широке застосування в сучасному світі [4]. У даному завданні алгоритми використовуються для знаходження маршруту в лабіринті. При цьому, лабіринт може відображати автомобільні дороги, план будинку, мапу ігрового світу. Тобто алгоритми реалізовані у програмі, можуть бути застосовані для навігаційних систем, переміщення робототехніки, вказівки шляху гравцю.

Мета: дослідити та порівняти ефективність різних алгоритмів пошуку маршруту на прикладі пошуку маршруту в лабіринті. Використати різні набори лабіринтів для порівняння результатів. Оцінити алгоритми за чотирма критеріями, а саме кількість кроків, час виконання, оптимальність маршруту та використання пам’яті. Ці критерії допоможуть визначити ефективність алгоритмів.

Завдання: написати програму мовою програмування C++, котра реалізує різноманітні алгоритми пошуку маршруту в лабіринті. Перевірити результати роботи програми на коректність. Після підтвердження коректності результатів, використати набір лабіринтів для кожного реалізованого алгоритму та отримати результати роботи. Порівняти результати роботи реалізованої програми.

Методи дослідження:

-вивчення книг, статей та інших джерел, які описують алгоритми пошуку маршруту в лабіринті;

-аналіз алгоритмів пошуку маршруту в лабіринті;

-використання реалізованої програми для візуалізації роботи алгоритмів, що дозволить краще зрозуміти процес пошуку маршруту в лабіринті;

-використання набору тестових лабіринтів для порівняння ефективності різних алгоритмів за результатами роботи програми;

-перевірка результату алгоритму непрохідним лабіринтом.

# 1 АНАЛІЗ ЗАВДАННЯ

Пошук маршруту в лабіринті є класичною алгоритмічною задачею. Це завдання передбачає знаходження шляху від стартової точки до кінцевої точки в заданому лабіринті.

Оскільки, потрібно знайти шлях у великій кількості лабіринтів, доцільно розробити та реалізувати комп’ютерну програму. Для точності та швидкості виконання завдання використаю мову програмування C++. Як компілятор, що охоплює потрібні бібліотеки, буде Cygwin.

Для того, щоб програма могла знайти маршрут їй необхідні дві складові лабіринт для якого потрібно шукати маршрут та алгоритм для знаходження маршруту.

У описі головоломки написано, що лабіринт заданий прямокутною матрицею з нулів та одиниць, де одиниця означає прохід, а нуль — стіну [2]. Також у завданні сказано, що матриця зчитується з файлу. Тому, одразу зрозуміло, що ввід буде реалізований з окремих файлів. Додатково варто реалізувати ввід із консолі від користувача. Тобто у програмі буде два різних методи вводу лабіринту для подальшої обробки програмою.

Через те, що користувач може припуститися помилки, перед обробкою буде реалізовано перевірку шляху до файлу та лабіринтів на коректність. Із завдання випливає, що перший і останній рядки та стовпці лабіринту мають бути нулями, тому в фінальній програмі буде відповідна функція для перевірки лабіринту. Також у цю функцію варто додати перевірку на те чи розташована одиниця на початку лабіринту, бо інакше, якщо на початку лабіринту стіна це може викликати непередбачуваний результат.

Для перевірки шляху до файлу з лабіринтом на коректність буде реалізована функція з використанням регулярних виразів [10]. Буде створено шаблон шляху до файлу на windows та подальша перевірка на збіг шляху введеного користувачем із шаблоном. Така функція необхідна для захисту від ін’єкцій в консоль. Такі ін’єкції полягають у тому, що зловмисник замість, наприклад, шляху до файлу вводить шкідливу команду, яка буде виконана незахищеною програмою [11].

Окрім лабіринту та перевірок програмі необхідні алгоритми для слідування. Завдання вимагає використання пошуку у ширину, тож першим алгоритмом буде пошук у ширину. Другий алгоритм для реалізації та порівняння буде відносно схожий, а саме, алгоритм пошуку у ширину, що починається з обох кінців лабіринту. Додатковим, третім, алгоритмом буде мурашиний алгоритм, що не дає гарантії знаходження найкоротшого шляху, але цей алгоритм може виявитися менш затратним по часу та ресурсах, а отже він буде корисним для порівняння [1].

З результатів аналізу статей про пошук в ширину та пошук в ширину, що починається з обох кінців, знадобиться черга [5]. Тому чергу теж потрібно реалізувати, окремим класом з усіма можливими перевірками, щоб забезпечити нормальне функціонування програми.

Четвертим необов’язковим для порівняння способом розв’язання лабіринту буде вирішення власноруч. Цей спосіб дозволить користувачам відчути себе в ролі виконавців та допоможе краще зрозуміти алгоритми.

Проаналізувавши основне завдання, можна перейти до вимог програми та результатів її роботи. Тому, що нам необхідно порівнювати алгоритми за деякими критеріями, потрібно щоб вивід програми був змістовним та включав необхідні критерії. А саме час виконання, кількість кроків, використання пам’яті та оптимальність маршруту.

Час виконання алгоритму буде реалізований за допомогою окремого класу секундоміра. Вивід буде відбуватися в секундах після завершення алгоритму. Порівнюючи, перевага буде у того алгоритму, що знайшов маршрут за меншу кількість секунд. Секундомір буде містити всі необхідні перевірки, щоб запобігти неправильному використанню.

Кількість кроків буде рахуватися окремо для кожного алгоритму за схожим принципом. Затоплення сусідньої клітинки у випадку пошуку у ширину та пошуку у ширину з двох кінців еквівалентне одному кроку. У випадку мурашиного алгоритму і вирішення власноруч переміщення на одну клітинку еквівалентне одному кроку. Вивід кроків, також, буде відбуватися по завершенню алгоритму. Перевага буде в алгоритму, що знайшов шлях за меншу кількість кроків.

Використання пам’яті важче порівняти, бо це потребує певних знань та аналітичних навичок. Для цього необхідно буде вивчити та проаналізувати алгоритми. Після цього можна буде робити висновки про використання пам’яті. Перевага у того алгоритму, що використовує менше пам’яті.

Щодо оптимальності маршруту теж все може бути непросто, але при обраних алгоритмах оцінка буде зводитися до того чи знаходить алгоритм найкоротший шлях. Якщо так, то у такого алгоритму перевага над тим алгоритмом, що не знаходить найкоротший шлях для набору лабіринтів.

Після реалізації програми її необхідно протестувати на коректність, тому буде вибрано стандартний тестовий лабіринт та на ньому перевірено результати виконання кожного алгоритму. Результати буде порівняно з виконанням алгоритму користувачем. Для виконання користувачем ролі виконавця алгоритмів можна скористатися проходженням лабіринту власноруч. Якщо у результаті порівняння буде виявлено неточності, то робота продовжиться після їх усунення.

Для програми, що коректно працює, можна сформувати набір лабіринтів для подальшого розв’язання реалізованими алгоритмами. Набір має включати щонайменше 5 лабіринтів. Перший невеликий, щоб перевірити чи буде перевага у якогось алгоритму у випадку невеликого лабіринту. Другий лабіринт повинен бути навпаки, якомога більшим, для знаходження найкращого алгоритму для великих лабіринтів. Третій лабіринт повинен бути без маршруту до виходу, щоб перевірити як будуть поводити себе алгоритми, якщо виходу немає. Четвертий та п’ятий будуть вибрані навмання для більшої вибірки.

Отриманні результати роботи можна буде занести в таблицю для зручного порівняння. У результаті порівняння за різноманітними критеріями для різних лабіринтів можна буде зробити висновки. Висновки щодо того який алгоритм кращий у певних ситуаціях чи навпаки який алгоритм краще не використовувати для певних вхідних даних.

Висновки будуть мати важливе значення. Тому що алгоритми пошуку маршруту в лабіринті мають широке застосування не лише у головоломках. Вони допомагають прокладати оптимальні маршрути на карті між двома точками, що важливо для автомобільної навігації та навігації у цілому [4]. Роботи використовують алгоритми пошуку маршруту для переміщення у просторі, уникнення перешкод та виконання завдань у складних середовищах. Ігрова індустрія використовує ці алгоритми для створення розумних ботів, які орієнтуються у віртуальному світі.

# 2 ПРОГРАМНА РЕАЛІЗАЦІЯ

У цьому розділі буде описано процес розробки та реалізації програми пошуку найкоротшого маршруту в лабіринті. Починається розділ з вибору технологій, далі аналіз діаграм та змін, опис програми та процесу оптимізації. Завершується розділ описом тестування та налагодження програми з подальшим виконанням поставленої задачі.

## 2.1 Початкові рішення

Програма вимагає порівняння ефективності алгоритмів, тому краще щоб мова програмування теж була швидка. Такою мовою у цьому випадку буде слугувати C++. У комбінації з компілятором Cygwin.

C++ дозволяє реалізувати парадигму об’єктно-орієнтованого програмування, котра базується на класах, об’єктах класу їх полях та методах [6]. Дана парадигма має ряд переваг, що будуть дуже корисними у програмі. Ось деякі переваги:

1. ООП дозволяє розділити програму на класи. Кожен клас має власні зміні та методи, які оброблюють ці зміні. Це спрощує розробку, тестування та масштабування коду, оскільки зміни одного класу не впливають на інші.
2. ООП дозволяє реалізувати поліморфізм [6]. Що є досить зручно, тому що з’являється можливість обробки об’єктів різних класів з використанням загального інтерфейсу. Це дозволяє створювати більш розширюваний та зручний код.
3. Інкапсуляція дозволяє заборонити доступ до змінних об’єкта напряму [6]. Взаємодіяти з таким об’єктом можливо ззовні за допомогою визначеного інтерфейсу. Це забезпечує правильну взаємодію з об’єктом.
4. ООП містить в собі наслідування, що дозволяє розширювати класи шляхом вже наявних [6].

## 2.2 Діаграми

Під час розробки програми було зроблено декілька діаграм, що дозволило краще продумати загальну структуру проекту. У цьому підпункті буде розглянуто деякі з них.

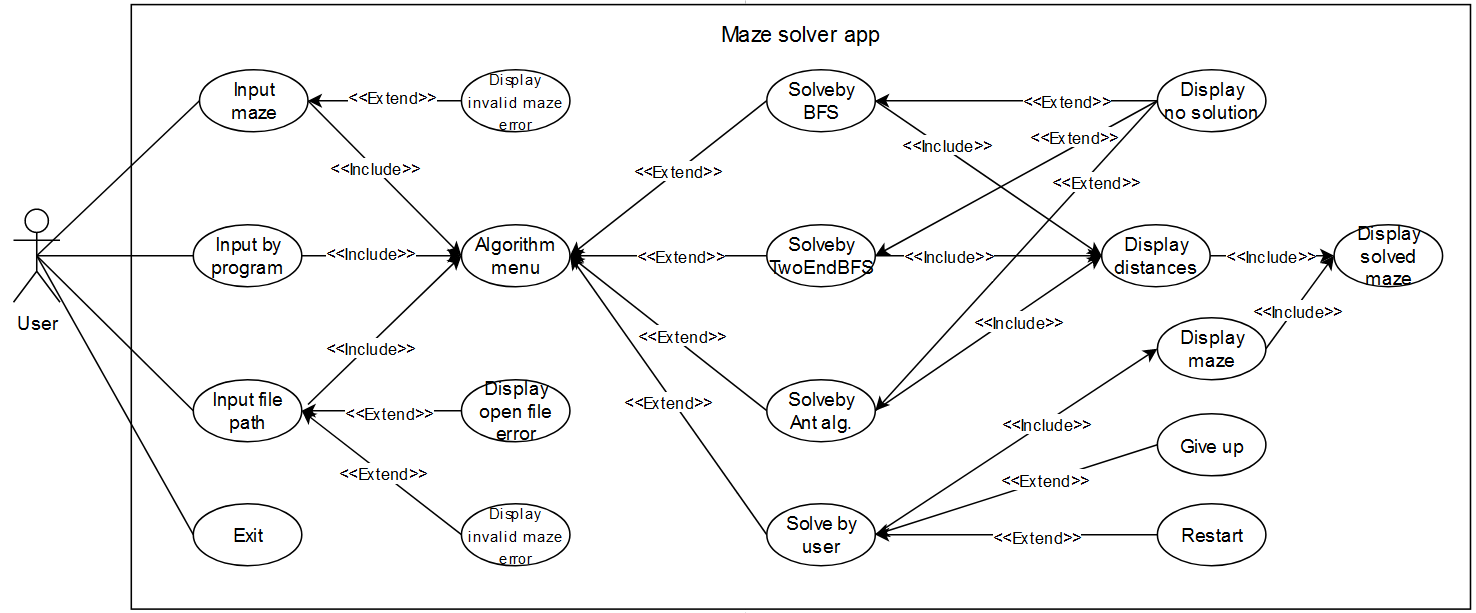
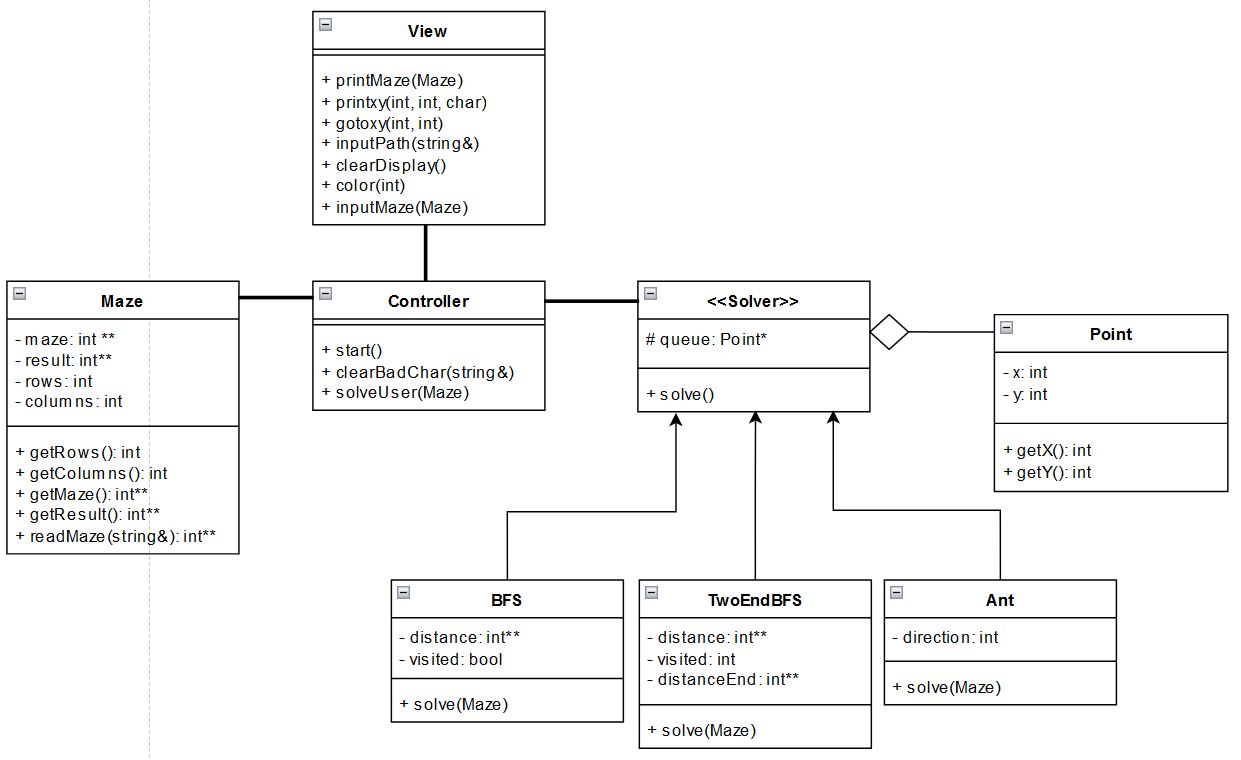
Перша діаграма це діаграма використання, що відображає взаємодію між користувачем та програмою (рисунок 2.1). З цією діаграмою можна чітко побачити, що потрібно реалізувати у програмі. А саме користувацьке меню, алгоритм пошуку в ширину, алгоритм пошуку в ширину з двох кінців, мурашиний алгоритм, рішення власноруч, вивід помилок, вивід дистанцій, відображення лабіринту та вивід шляху.

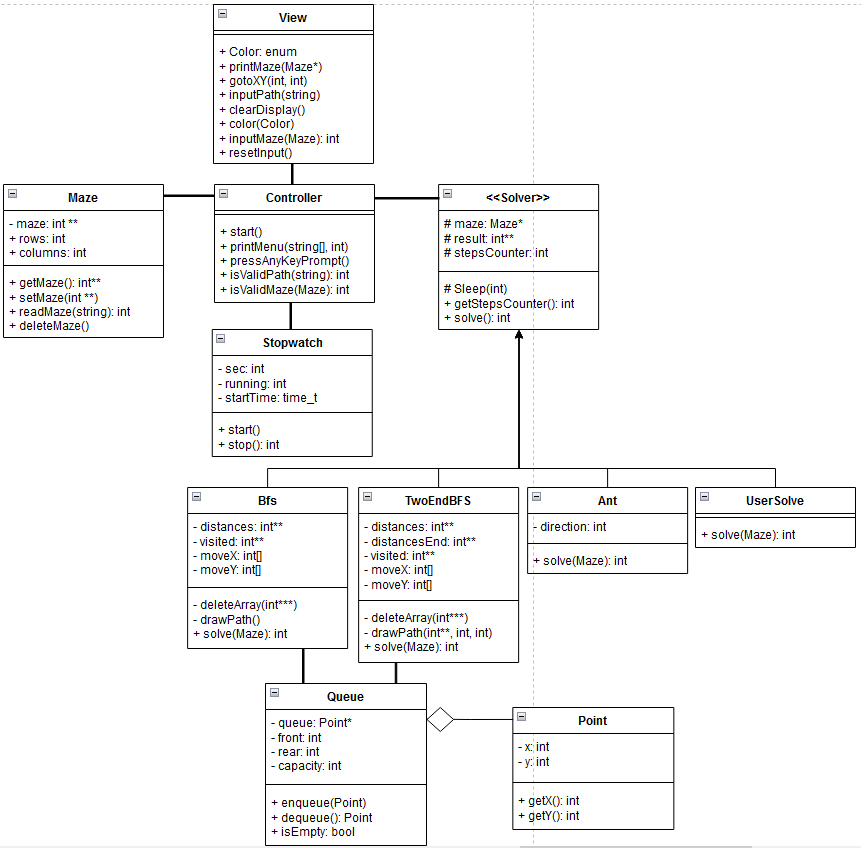
Рисунок 2.1 – Діаграма використання

Друга діаграма це діаграма класів, котра відображає класи та взаємодію, між ними (рисунок 2.2). На цій діаграмі можна побачити кожен клас з його полями та методами. Потрібно звернути увагу на особливості програми, що видно з діаграми. В ній використовується архітектурний шаблон «модель-представлення-контролер», де модель відповідає за дані, визначає структуру, представлення відповідає за взаємодію з користувачем, тобто ввід та вивід, контролер відповідає за зв’язок між моделлю та представленням [9]. Також у програмі можна використати динамічний поліморфізм, що є можливим завдяки трьом алгоритмічним класам, що наслідують один абстрактний клас та перевантажують чисто віртуальну функцію [3].

Рисунок 2.2 – Діаграма класів

Через, що я маю недостатньо досвіду, зроблена мною діаграма класів у ході реалізації програми змінювалась. Варто відзначити, що діаграма використання не змінилася до фінальної версії програми, тому що в першу чергу я дотримувався діаграми використання та змінював діаграму класів, щоб реалізувати весь зображений функціонал якомога краще.

На рисунку 2.3 наведено кінцеву діаграму класів, що відповідає фінальній версії програми. У процесі розробки виникла необхідність у додаткових модулях, щоб програма була більш гнучкою та простішою для розуміння. Тому кількість модулів зросла. Також можна побачити, що solveUser через складність реалізації було реалізовано як клас замість функції. Так само з чергою, яка винесена окремим класом. Додались безліч нових функцій та змінних, які були необхідні для спрощення коду, що повторювався. У класі Solver додані додаткові змінні що характерні для всіх похідних класів. Що до самих алгоритмів, то у класах де це необхідно функцію solve розділено на дві функції solve та drawPath. Ще одним важливим оновленням є зміна модифікатора доступу для rows та columns.

Рисунок 2.3 – Оновлена діаграма класів

Після розробки діаграм у мене була можливість розпочинати реалізацію програми. Тому я почав реалізовувати програму модуль за модулем, слідуючи діаграмам наскільки це було можливо.

## 2.3 Написання коду та опис програми

Якщо розділити код програми на умовні модулі, то можна написати послідовність написання модулів. Модулі були написані у такій послідовності:

1. Користувацьке меню.
2. Клас для зберігання лабіринту та ввід з файлу.
3. Відображення лабіринту.
4. Алгоритм пошуку в ширину.
5. Вивід дистанцій та шляху.
6. Ввід з консолі, перевірки та вивід помилок.
7. Інші алгоритми пошуку та вирішення власноруч.

Реалізація програми у такому порядку дозволила тестувати деякі модулі після написання модулів виводу інформації. Тому, що модулі призначені для відображення тієї чи іншої інформації були реалізовані, якомога раніше. Тобто код написаний раніше можна було візуально тестувати. Це виявилось досить корисним, тому що більшість помилок програмування були виправлені на ранніх етапах.

Користувацьке меню було реалізоване таким чином, що користувач обирає потрібний пункт за допомогою стрілок. Для цього знадобилася бібліотека conio.h, щоб функцією getch() зчитувати клавіші без натискання клавіші введення [8]. Дана бібліотека призвела до помилок, адже компілятор Cygwin не має в собі цієї бібліотеки. Тому додатково довелося змінити компілятор на Mingw-w64. Даний компілятор дозволив реалізувати меню, що керується стрілками вниз та вгору.

Наступним важливим модулем є зчитування з файлу. Але неможливо було реалізувати зчитування з файлу до того як буде реалізований клас для зберігання зчитаних даних. Отже, наступними були реалізовані класи для зберігання даних, функція вводу з файлу та функція відображення лабіринту. Візуалізація дозволила зручно протестувати реалізацію вводу з файлу та виправити наявні помилки.

Після реалізації меню, класу для зберігання даних, вводу з файлу та відображення лабіринту було реалізовано алгоритм пошуку в ширину. Для роботи цього алгоритму необхідна черга, яку вже було реалізовано. Тому алгоритм починається зі створення черги всередині функції. Початок лабіринту розташований у лівому верхньому куті [2]. Точка, що відповідає початку, додається в чергу та відмічається як відвідана [7]. Далі для всіх точок в черзі, допоки черга не порожня або маршрут до правого нижнього кута не знайдений, перевіряються сусідні точки. Якщо сусідня точка невідвідана та не є стінкою алгоритм рахує відстань для неї, як відстань до поточної точки збільшена на одиницю. Також алгоритм додає цю точку до черги та помічає як відвідану. Відстані та відвідуваність у цьому та алгоритмі пошуку в ширину з двох кінців зберігаються за допомогою вказівника на вказівник на ціле число. У результаті роботи описаного алгоритму буде знайдено найкоротший маршрут або не знайденого жодного маршруту, що буде означати що маршрут не існує [7].

Далі було написано код для виводу дистанцій та маршруту для алгоритму пошуку в ширину. Надалі й для інших алгоритмів за схожим принципом. Всередині функції алгоритм отримує координати поточної точки. Далі за допомогою функції переміщення курсора та отриманих координат виводить відстань у зарання виведеному лабіринті. Також всередині функції перед її завершенням викликається допоміжна функція виводу маршруту. Що починається із кінцевої клітинки. Беручи відстані дана функція шукає сусідню клітинку поточної точки з меншою відстанню. Після знаходження такої клітинки вона береться як поточна та за її координатами у виведеному лабіринті виводиться шлях.

Після реалізації виводу відстаней та шляху можна було перевірити алгоритм на правильність. Алгоритм пошуку в ширину працював правильно, тому виправлення не знадобилися. Інші алгоритми було перевірено після їх реалізації.

Для підвищення стабільності програми наступними для реалізації були перевірки та вивід помилок. Була реалізована перевірка шляху до файлу з використанням бібліотеки regex, що дозволяє використовувати регулярні вирази. Регулярні вирази потужний інструмент, що дозволяє легко перевірити збіг тексту з шаблоном. Перевірка лабіринту зводиться до перевірки відповідності лабіринту умовам. На випадок якщо з лабіринтом чи шляхом до файлу є якісь проблеми зроблено вивід помилок.

Далі було реалізовано ввід лабіринту з консолі, що використовує ті самі функції перевірок. Та вже разом все перевірено на наявність помилок.

На даному етапі у програмі було реалізовано майже все за виключенням алгоритмів. Отже, наступними для реалізації були три класи з таким самим інтерфейсом, що й у класі пошуку в ширину. Надалі спільний інтерфейс та наслідування дозволили використати динамічний поліморфізм у контролері. Це полегшило код та подальше розширення програми.

Другим з чотирьох алгоритмів був реалізований мурашиний алгоритм. Алгоритм має в собі поле напряму. Тобто уявний виконавець алгоритму рухається у заданому напрямі. На кожній ітерації виконавець намагається повернути наліво та зробити крок у новому напрямі [1]. Якщо виконавець не може цього зробити, то він йде прямо, не змінюючи напрям. Якщо ж виконавець не може рухатися ні наліво, ні прямо він повертає праворуч. Під час того як виконавець рухається за його координатами в лабіринті виводиться шлях, так само як і в алгоритмі пошуку в ширину. Алгоритм продовжується, доки лабіринт не буде пройдено або зміна для підрахунку кроків по вже пройденим клітинкам не зменшиться до нуля. Дана зміна зменшується на одиницю на кожній ітерації.

Наступним був реалізований клас для вирішення лабіринту власноруч. Даний клас використовує ту саму бібліотечну функцію, що і контролер для управління меню. За допомогою цієї функції користувач проходить лабіринт. Після натискання клавіші стрілки шлях продовжується у вибраному напрямі, якщо там є прохід. Шлях одразу виводиться в консоль. Цей процес продовжується допоки лабіринт не буде пройденим або користувач не натисне клавішу виходу.

Останнім алгоритмом був алгоритм пошуку в ширину з двох сторін. Також цей алгоритм є найважчим для реалізації. Він працює так само як алгоритм пошуку в ширину, але одразу додає до черги точку початку та точку кінця [5]. Тобто це еквівалентно двом пошукам в ширину що починаються з початку та з кінця. На цьому відмінності не закінчуються. Алгоритм має дві окремих зміні для зберігання відстаней. Це зроблено для того, щоб після вдалого завершення алгоритму можна було вивести шлях, так само як для звичайного алгоритму пошуку в ширину. Також хоч алгоритм має одну зміну для запису відвідуваності, відвідуваність позначається різними числами для точок, що почалися з початку та кінця. Таким чином алгоритм рахує відстані, позначаючи відвідуваність. Алгоритм завершує свою роботу у двох випадках. Перший, коли черга спорожніє це означає що маршрут не існує. Другий випадок це перетин двох пошуків у ширину [5]. Тоді найкоротший маршрут буде проходити через точку їх перетину. Програма розуміє, що два пошуки перетнулися, коли один пошук натрапляє на точку відвідану іншим пошуком.

Фінальним етапом розширення функціональності програми було додавання секундоміра з виводом часу, додавання підрахунку кількості кроків у кожен алгоритм та додавання затримки у кожен алгоритм, що дозволило зробити вивід покроковим.

## 2.4 Оптимізація, тестування та налагодження програми

Програма працює. Але на цьому процес написання коду не завершується. По завершенню програма виявилася неоптимізованою. Тому деякий час пішов на те, щоб додати макрозміні та enum, замість «магічних чисел», функції, щоб замінити ними частини коду що повторювалися. Також було додано коментарі та форматовано код для спрощення розуміння написаного.

Програма написана та оптимізована, тому можна було розпочинати тестування. У результаті тестування було виявлено деякі помилки.

Секундомір виводив від’ємний час. Ця проблема полягала у тому, що у функції від часу запуску секундоміра віднімався час зупинки. Змінивши час запуску та зупинки місцями проблему було вирішено.

У результаті вирішення лабіринту власноруч виводилось вдвічі більше кроків. Проблема полягала у тому, що при натисканні клавіші getch() зчитував її, але замість одної ітерації відбувалися дві. Перша для зчитаного символу та друга для зайвого. Додаванням ще одного getch() для зчитування зайвого символу було вирішено проблему.

Після завершення пошуку в ширину з двох сторін програма вилітала через використання не тої функції в деструкторі класу. Заміна функції на правильну вирішила проблему.

Після налагодження програми її було ще раз перевірено та не виявлено ніяких проблем.

## 2.5 Лабіринти та результати

Обрано 5 лабіринтів для тестування ефективності алгоритмів. Лабіринти записано у 5 файлів. Кожен лабіринт буде зчитано з файлу та по черзі запущено кожен алгоритм для знаходження маршруту. Результати занесено у дві таблиці. Таблицю часу, що знадобився алгоритмам для знаходження маршруту (таблиця 2.1). Таблицю кількості кроків, що знадобилися для знаходження маршруту (таблиця 2.2).

Таблиця 2.1 – Час виконання алгоритмів

|  |  |  |  |
| --- | --- | --- | --- |
| № | Пошук у ширину | Пошук у ширину з двох кінців | Мурашиний |
| 1 | 8 c. | 7 c. | 3 c. |
| 2 | 65 c. | 45 c. | 14 c. |
| 3 | 1 c. | 4 c. | 15 c. |
| 4 | 9 c. | 9 c. | 6 c. |
| 5 | 9 c. | 10 c. | 6 c. |

Таблиця 2.2 – Кількість кроків алгоритмів

|  |  |  |  |
| --- | --- | --- | --- |
| № | Пошук у ширину | Пошук у ширину з двох кінців | Мурашиний |
| 1 | 17 | 16 | 9 |
| 2 | 164 | 103 | 42 |
| 3 | 6 | 12 | 31 |

Продовження таблиці 2.2

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 17 | 16 | 13 |
| 5 | 21 | 19 | 14 |

Варто зазначити, що у двох з п’яти лабіринтів мурашиний алгоритм знайшов найкоротший маршрут. Інші алгоритми знайшли найкоротший маршрут у всіх лабіринтах, де це було можливо.

Якщо говорити про використання пам’яті, то найменш затратним є мурашиний алгоритм, адже він додатково зберігає лише напрям руху. Найбільш затратним є алгоритм пошуку в ширину з двох кінців, через те що він окрім черги та відстаней, що також використовує звичайний алгоритм пошуку в ширину, додатково зберігає відстані для кінцевого пошуку в ширину.

# 3 КЕРІВНИЦТВО КОРИСТУВАЧА

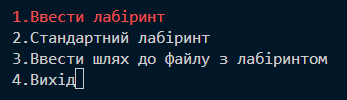
При запуску програми вас зустріне головне меню програми (рисунок 3.1). У цьому меню вам необхідно обрати зручний спосіб введення лабіринту. Обирати пункти потрібно за допомогою стрілок вверх та вниз, підтверджувати вибір клавішею введення.

Рисунок 3.1 – Головне меню програми

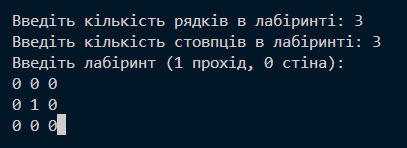
Якщо ви обрали варіант вводу лабіринту власноруч, то варто пам’ятати, що крайні стовпці та рядки мають бути заповнені нулями (рисунок 3.2). Також на початку лабіринту має бути одиниця. Вам необхідно вводити інформацію що вимагає програма. У разі отримання помилки уважно прочитайте її та введіть лабіринт правильно.

Рисунок 3.2 – Ввід лабіринту власноруч

Якщо ви обрали введення шляху до файлу з лабіринтом, то вимоги до лабіринту будуть ті самі як і для введення власноруч (рисунок 3.3). У разі отримання помилки пов’язаної з лабіринтом змініть лабіринт у файлі. Варто відзначити, що програма підтримує лише шляхи до файлу шаблону windows.

Рисунок 3.3 – Ввід шляху до файлу з лабіринтом

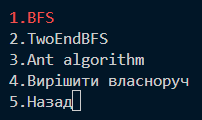
Після введення лабіринту будь-яким способом ви побачите меню вибору алгоритмів. Керування те саме (рисунок 3.4).

Рисунок 3.4 – Меню вибору алгоритму

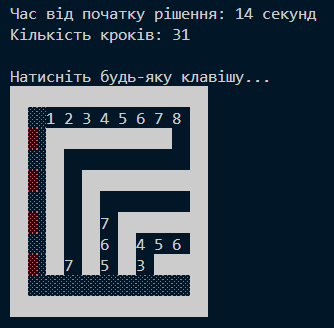
Тут вам необхідно обрати один з трьох алгоритмів або вирішення власноруч. Після вибору одного з трьох алгоритмів ви побачите покроковий вивід. У кінці алгоритму виведеться корисна інформація та знайдений шлях (рисунок 3.5).

Рисунок 3.5 – Результат роботи алгоритму

На рисунку 3.5 можна побачити час виконання в секундах, кількість кроків та знайдений маршрут. Щоб закрити результат натисніть будь-яку клавішу. Ви потрапите в меню алгоритмів.

Якщо ви обрали вирішення лабіринту власноруч, то ви побачите підказки з керування (рисунок 3.6). Рухатися стрілками, вихід подвійне натискання клавіші E. Якщо ви успішно прокладете маршрут вам виведеться така сама корисна інформація, як і після завершення алгоритму.

Рисунок 3.6 – Вирішення лабіринту власноруч

Щоб з меню алгоритмів потрапити у головне меню виберіть пункт «Назад». Далі щоб правильно вийти з програми у головному меню виберіть пункт «Вихід».

# ВИСНОВКИ

Маючи табличку часу та відстані, зроблю загальну табличку (таблиця 1).

Таблиця 1 – Загальна таблиця

|  |  |  |  |
| --- | --- | --- | --- |
| Критерій | Пошук у ширину | Пошук у ширину з двох кінців | Мурашиний |
| Час, с | 92 | 75 | 44 |
| Кроки | 225 | 166 | 109 |
| Всі маршрути найкоротші | + | + | - |
| Використання пам’яті | 2 | 3 | 1 |

Отже, я написав програму мовою програмування C++. Програма відповідає початковій діаграмі використання (рисунок 2.1). Реалізував 3 алгоритми пошуку маршруту в лабіринті пошук у ширину, пошук у ширину з двох сторін та мурашиний алгоритм. Перевірив, налагодив та оптимізував програму. Після налагодження протестував ефективність алгоритмів 5 лабіринтами та результати заніс у таблиці. Зараз аналізуючи загальну таблицю можна написати висновки до алгоритмів.

Найшвидшим алгоритмом, що до того ж використовує найменше пам’яті є мурашиний алгоритм. Але є нюанс цей алгоритм не завжди знаходить найкоротший маршрут та не підходить для лабіринтів у яких немає виходу. Якщо не стоїть задача знайти саме найкоротший маршрут та лабіринт гарантовано буде мати маршрут до виходу, то краще використовувати мурашиний алгоритм.

Найкращим алгоритмом, що знаходить найкоротший маршрут є пошук у ширину з двох кінців. Адже цей алгоритм швидше за звичайний пошук у ширину на 22.6% та знаходить маршрут за меншу кількість кроків. Але варто відзначити, що даний алгоритм хоч і є оптимальним, бо швидко знаходить найкоротший шлях, він також є найбільш затратним по ресурсах.

Тому, якщо є обмеження по ресурсах, що не дозволяє використати пошук у ширину з двох сторін, але потрібно знайти саме найкоротший шлях, то краще скористатися звичайним пошуком у ширину.

Програма виконує заплановане завдання та має потенціал для масштабування внаслідок модульності ООП. Тобто у майбутньому можна додавати нові алгоритми для порівняння. Та використовувати вже наявні зміні та функції.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Алгоритм розв'язування лабіринтів. *Wikipedia*. URL: <http://surl.li/ugrgc> (дата звернення: 06.06.2024).
2. Теми курсових робіт. *Google Sheets*. URL: <http://surl.li/ugqyc> (дата звернення: 06.06.2024).
3. Шпурик В. В. Unit 11.run-time polymorphism. *Fundamentals of Programming-II*. URL: <http://surl.li/ugslw> (date of access: 06.06.2024).
4. Applications, Advantages and Disadvantages of Breadth First Search (BFS). *Geeks for geeks*. URL: <http://surl.li/ugucx> (date of access: 06.06.2024).
5. Bidirectional search. *Geeks for geeks*. URL: <http://surl.li/ugrzw> (date of access: 06.06.2024).
6. Object oriented programming in C++. *Geeks for geeks*. URL: <http://surl.li/ugseh> (date of access: 06.06.2024).
7. Breadth-first search. *Wikipedia*. URL: <http://surl.li/ugsqz> (date of access: 06.06.2024).
8. Getch() function in C with Examples. *Geeks for geeks*. URL: <http://surl.li/ugsos> (date of access: 06.06.2024).
9. MVC design pattern. *Geeks for geeks*. URL: <http://surl.li/ugskd> (date of access: 06.06.2024).
10. Regular expressions. *cplusplus.com*. URL: <http://surl.li/ugrwb> (date of access: 06.06.2024).
11. Weilin Z. Command injection. *Owasp*. URL: <http://surl.li/ugrxe> (date of access: 06.06.2024).

# ДОДАТОК 1

## Специфікація

НТУУ «КПІ ім.І.Сікорського» ННІАТЕ ІПЗЕ ТВ-31

Листів 2

Київ – 2024

|  |  |  |
| --- | --- | --- |
| Призначення | Найменування | Примітки |
| Головний файл програми | Main.cpp | Виклик функції Start() |
| Оголошення класу Ant, що виконує алгоритм мурашиного пошуку | Ant.h | Клас є нащадком класу Solver |
| Перевизначає метод solve(), який виконує алгоритм мурашиного пошуку | Ant.cpp | Відображає кроки проходження мурашки в консоль |
| Оголошення класу Bfs, що виконує алгоритм пошуку в ширину | Bfs.h | Клас є нащадком класу Solver |
| Перевизначає метод solve(), який виконує алгоритм пошуку в ширину | Bfs.cpp | Виводить відстані та маршрут, виділяє пам’ять |
| Оголошення макрозміних вводу та функцій перевірки, виведення меню | Controller.h | Включає бібліотеки conio.h, regex, об’єднує всі модулі |
| Містить функції керування меню, вибору алгоритму та перевірок | Controller.cpp | Зв’язує Viem та Maze |
| Оголошення класу Maze, що представляє лабіринт | Maze.h | Включає бібліотеки iostream, fstream, new |
| Відповідає за створення зберігання та обробку лабіринту | Maze.cpp | Динамічно виділяє пам’ять |
| Оголошення класу Point, що представляє точку | Point.h | Точка має координати x та y |
| Містить конструктор точок та методи для отримання координат | Point.cpp | Перевантаження конструктора |
| Оголошення класу Queue, що представляє чергу | Queue.h | Черга точок |
| Містить визначення функцій черги | Queue.cpp | Методи enqueue(), dequeue(), isEmpty() |

|  |  |  |
| --- | --- | --- |
| Призначення | Найменування | Примітки |
| Оголошення макрозміних та абстрактного класу Solver, що є загальним інтерфейсом для алгоритмів пошуку | Solver.h | Містить чисто віртуальну функцію Solve(), включає бібліотеки thread та chrono |
| Містить базовий клас для рішення лабіринту | Solver.cpp | Методи Sleep(), getStepCounter() |
| Оголошення класу Stopwatch, що представляє секундомір | Stopwatch.h | Включає бібліотеку ctime |
| Містить визначення функцій секундоміра | Stopwatch.cpp | Зберігає час в секундах, методи start(), stop() |
| Оголошення класу TwoEndBfs, що виконує алгоритм двонапрямленого пошуку в ширину | TwoEndBfs.h | Клас є нащадком класу Solver |
| Перевизначає метод solve(), який виконує алгоритм двонаправленого пошуку в ширину | TwoEndBfs.cpp | Виводить відстані та маршрут, виділяє пам’ять, методи drawPath(), deleteArray() |
| Оголошення класу UserSolve, для розв’язання лабіринту користувачем | UserSolve.h | Клас є нащадком класу Solver |
| Перевизначає метод solve() | UserSolve.cpp | Вирішення власноруч |
| Оголошення функцій відображення, керування кольором, вводу та виводу | View.h | Оголошує макрозміні символів для виводу та enum кольорів |
| Взаємодія з користувачем та вивід лабіринту у консоль. | View.cpp | Методи printMaze(), gotoXY(), inputPath(), clearDisplay(), color(), inputMaze(), resetInput() |

# ДОДАТОК 2

## Код програми

НТУУ «КПІ ім.І.Сікорського» ННІАТЕ ІПЗЕ ТВ-31

Листів 50

Київ – 2024

#ifndef \_ANT\_H\_

#define \_ANT\_H\_

//макрозміна обмеження, на випадок неможливості пройти лабіринт

#define MAX\_VISITED\_STEPS 40

#include "Solver.h"

class Ant: public Solver{

//напрям руху

int direction;

public:

Ant(Maze &Maze);

//перевизначення наслідуваної функції

int solve() override;

};

#endif

#ifndef \_BFS\_H\_

#define \_BFS\_H\_

#include "Solver.h"

#include "Queue.h"

class Bfs: public Solver{

//відстані

int \*\*distances;

//відвідуваність

int \*\*visited;

//всі можливі напрями руху

int moveX[4]={0, 0, -1, 1};

int moveY[4]={-1, 1, 0, 0};

//функція для очищення динамічно виділеної пам'яті

void deleteArray(int\*\*\* array);

//функція для виводу знайденого шляху

void drawPath() const;

public:

Bfs(Maze &maze);

~Bfs();

//перевизначення наслідуваної функції

int solve() override;

};

#endif

#ifndef \_CONTROLLER\_H\_

#define \_CONTROLLER\_H\_

//макрозміні

#define KEY\_UP 72

#define KEY\_DOWN 80

#define KEY\_LEFT 75

#define KEY\_RIGHT 77

#define ENTER '\r'

//шлях до файлу стандартного лабіринту

#define STD\_PATH "mazes\\maze1.txt"

#include <conio.h>

#include <regex>

#include "Maze.h"

#include "View.h"

#include "Stopwatch.h"

#include "Solver.h"

#include "Bfs.h"

#include "TwoEndBfs.h"

#include "Ant.h"

#include "UserSolve.h"

void start();

//вивід меню

void printMenu(const string menu[], const unsigned int size);

//запит натиснути будь-яку клавішу

void pressAnyKeyPrompt();

//перевірка шляху до файлу на правильність

int isValidPath(const string &path);

//перевірка лабіринту на правильність

int isValidMaze(Maze &maze);

#endif

#ifndef \_MAZE\_H\_

#define \_MAZE\_H\_

#include <iostream>

#include <fstream>

#include <new>

using namespace std;

class Maze{

//масив для зберігання лабіринту

int \*\*maze;

public:

int rows;

int columns;

Maze();

~Maze();

int \*\*getMaze() const;

void setMaze(int \*\* const array);

//зчитати лабіринт з файлу

int readMaze(const string &fileName);

//видалит масив лабіринт

void deleteMaze();

};

#endif

#ifndef \_POINT\_H\_

#define \_POINT\_H\_

class Point{

//координати

int x;

int y;

public:

Point();

Point(const int i, const int j);

//гетери

int getX() const;

int getY() const;

};

#endif

#ifndef \_QUEUE\_H

#define \_QUEUE\_H

#include <iostream>

#include <new>

#include "Point.h"

using namespace std;

class Queue {

//змінна для зберігання точок

Point \*queue;

//фронт черги

int front;

//тил черги

int rear;

//вмістимість черги

int capacity;

public:

Queue(const int n);

~Queue();

//додати в чергу

void enqueue(const Point &p);

//витягнути з черги

Point dequeue();

//перевірка чи пуста черга

bool isEmpty() const;

};

#endif

#ifndef \_SOLVER\_H\_

#define \_SOLVER\_H\_

//макрозміна нескінченої відстані

#define INFINITY 2147483647

//макрозміна затримки

#define SLEEP\_TIME 300

#include <thread>

#include <chrono>

#include "Maze.h"

#include "View.h"

class Solver{

protected:

//посилання на лабіринт

Maze \*maze;

//лабіринт зі шляхом

int \*\*result;

//змінна для підрахунку кількості ходів

int stepsCounter;

//функція для затримки

void Sleep(const int millisec) const;

public:

Solver(Maze &maze);

//віртуальний деструктор

virtual ~Solver();

//гетер

int getStepsCounter();

//повністю віртуальна функція вирішення

virtual int solve()=0;

};

#endif

#ifndef \_STOPWATCH\_H\_

#define \_STOPWATCH\_H\_

#include <iostream>

#include <ctime>

using namespace std;

class Stopwatch{

//секунди

int sec;

//зберігання стану секундоміра

int running;

//зберігання часу запуску секундоміра

time\_t startTime;

public:

Stopwatch();

void start();

int stop();

};

#endif

#ifndef \_TWOENDBFS\_H\_

#define \_TWOENDBFS\_H\_

#include "Solver.h"

#include "Queue.h"

class TwoEndBfs: public Solver{

//відстані bfs, що починається з початку

int \*\*distances;

//відстані bfs, що починається з кінця

int \*\*distancesEnd;

//відвідуваність

int \*\*visited;

//всі можливі напрями руху

int moveX[4]={0, 0, -1, 1};

int moveY[4]={-1, 1, 0, 0};

//функція для очищення динамічно виділеної пам'яті

void deleteArray(int\*\*\* array);

//функція для виводу знайденого шляху

void drawPath(int \*\*dist, int curX, int curY) const;

public:

TwoEndBfs(Maze &maze);

~TwoEndBfs();

//перевизначення наслідуваної функції

int solve() override;

};

#endif

#ifndef \_USERSOLVE\_H\_

#define \_USERSOLVE\_H\_

#include "Solver.h"

#include "Controller.h"

class UserSolve: public Solver{

public:

UserSolve(Maze &maze);

//перевизначення наслідуваної функції

int solve() override;

};

#endif

#ifndef \_VIEW\_H\_

#define \_VIEW\_H\_

//символи стінки

#define WALL "\xe2\x96\x88\xe2\x96\x88"

//символи шляху

#define PATH "\xe2\x96\x91\xe2\x96\x91"

#include <iostream>

#include "Maze.h"

//enum кольорів

enum Color { RESET, RED };

//вивід лабіринту

void printMaze(const Maze \*maze);

//переміщення курсора в консолі

void gotoXY(const int x, const int y);

//ввід шляху до файлу

void inputPath(string &path);

//очищення консолі

void clearDisplay();

//зміна кольору виводу

void color(const Color cl);

//ввід лабіринту

int inputMaze(Maze &maze);

//очищення потоку вводу

void resetInput();

#endif

#include "Ant.h"

Ant::Ant(Maze &maze) : Solver(maze) {}

int Ant::solve(){

//перевірка на ініціалізацію

if(!maze) { return 1; }

//вивід лабіринту

printMaze(maze);

//всі можливі напрями руху

const int moveX[4] = {-1, 0, 1, 0};

const int moveY[4] = {0, 1, 0, -1};

int x = 1;

int y = 1;

//початковий напрям

int direction = 1;

gotoXY(1+y\*2, 5+x);

color(RED);

cout<<PATH;

color(RESET);

int counter=MAX\_VISITED\_STEPS;

//цикл, доки лабіринт не буде пройдений або counter не зменшиться до нуля

while (result[maze->rows-2][maze->columns-2] != 2 && counter>0) {

//затримка для покрокового виводу

Sleep(SLEEP\_TIME);

//спробувати повернути наліво

int newDirection = (direction + 3) % 4;

if (result[x+moveX[newDirection]][y+moveY[newDirection]] != 0) {

direction = newDirection;

//збільшуємо кількість кроків на одиницю

stepsCounter++;

x += moveX[direction];

y += moveY[direction];

//якщо рух неможливий то крок прямо

} else if (result[x+moveX[direction]][y + moveY[direction]] != 0) {

//збільшуємо кількість кроків на одиницю

stepsCounter++;

x += moveX[direction];

y += moveY[direction];

//якщо і попереду стінка, то поворот направо

} else {

direction = (direction + 1) % 4;

}

//якщо клітинка під мурахою вже була пройдена, то conter зменшується на одиницю

if(result[x][y]==2) counter--;

//позначення клітинки, як відвіданої

result[x][y]=2;

//вивід відвіданої клітинки

gotoXY(1+y\*2, 5+x);

color(RED);

cout<<PATH;

color(RESET);

}

//якщо counter зменшився до нуля, то алгоритм не знайшов шлях за обмежену кількість кроків по відвіданих клітинках

if(counter==0){

gotoXY(0,3);

cout<<"Не знайшов рішення за обмежену кількість кроків по відвіданих клітинках."<<endl;

}

return 0;

}

#include "Bfs.h"

Bfs::Bfs(Maze &maze) : Solver(maze) {

//перевірка на ініціалізацію

if(!maze.getMaze()) {

distances=nullptr;

visited=nullptr;

} else {

//виділення пам'яті

try{

distances=new int\*[maze.rows];

visited=new int\*[maze.rows];

for(int i=0; i<maze.rows; i++){

distances[i]=new int[maze.columns]();

visited[i]=new int[maze.columns]();

for(int j=0; j<maze.columns; j++){

distances[i][j]=INFINITY;

visited[i][j]=false;

}

}

} catch(const bad\_alloc &e){

cerr<<"Помилка виділення пам'яті: " << e.what() << endl;

//очищення виділеної пам'яті в разі помилки

deleteArray(&distances);

deleteArray(&visited);

}

}

}

Bfs::~Bfs() {

deleteArray(&distances);

deleteArray(&visited);

}

void Bfs::deleteArray(int\*\*\* array){

if (\*array) {

for (int k = 0; k < maze->rows; k++) {

if ((\*array)[k]) {

delete[] (\*array)[k];

}

}

delete[] \*array;

\*array = nullptr;

}

}

int Bfs::solve() {

//перевірка на ініціалізацію

if(!maze) { return 1; }

//вивід лабіринту

printMaze(maze);

//створення черги

Queue queue(maze->rows \* maze->columns);

//початкова точка

Point current, src(1, 1);

//додавання початкової точки в чергу

queue.enqueue(src);

//призначення 0 як відстані до початкової точки

distances[src.getX()][src.getY()]=0;

//позначення початкової точки як відвіданої

visited[src.getX()][src.getY()]=true;

int newX, newY;

gotoXY(1+src.getY()\*2, 5+src.getX());

cout<<distances[src.getX()][src.getY()];

//цикл, доки черга не пуста або не знайдений шлях

while(!queue.isEmpty() && distances[maze->rows-2][maze->columns-2]==INFINITY){

//береться точка з черги

current=queue.dequeue();

//цикл для всіх сусідніх клітинок

for(int i=0; i<4; i++){

newX=current.getX() + moveX[i];

newY=current.getY() + moveY[i];

//перевірка чи можливий рух в сусідню клітинку та чи не була вона відвідана

if(maze->getMaze()[newX][newY]==1 && !visited[newX][newY]){

//затримка

Sleep(SLEEP\_TIME);

//збільшуємо кількість кроків на одиницю

stepsCounter++;

//для сусідньої клітинки відстань збільшуємо на одиницю

distances[newX][newY] = distances[current.getX()][current.getY()]+1;

//позначаємо як відвідану

visited[newX][newY] = true;

//виводимо відстань

gotoXY(1+newY\*2, 5+newX);

cout<<distances[newX][newY];

//додаємо точку в чергу

Point newPoint(newX, newY);

queue.enqueue(newPoint);

}

}

}

//якщо алгоритм не знайшов відстань до кінцевої точки, то шлях не існує

if(distances[maze->rows-2][maze->columns-2]==INFINITY){

gotoXY(0,3);

cout<<"Рішення не існує."<<endl;

return 1;

}

//вивід шлях

drawPath();

return 0;

}

void Bfs::drawPath() const {

int newX, newY, currentX=maze->rows-2, currentY=maze->columns-2;

result[currentX][currentY]=2;

gotoXY(1+currentY\*2, 5+currentX);

color(RED);

cout<<PATH;

color(RESET);

//допоки не дійшли до стартової точки, відстань до якої 0

while(distances[currentX][currentY]){

//цикл для всіх сусідніх клітинок

for(int i=0; i<4; i++){

newX=currentX+moveX[i];

newY=currentY+moveY[i];

//перевірка чи відстань до сусідньої клітинки менша

if(distances[newX][newY]<distances[currentX][currentY]){

//позначаємо та виводимо шлях

result[newX][newY]=2;

currentX=newX;

currentY=newY;

gotoXY(1+currentY\*2, 5+currentX);

color(RED);

cout<<PATH;

color(RESET);

//затримка для покрокового виводу

Sleep(SLEEP\_TIME);

}

}

}

}

#include "Controller.h"

//змінна вибору пункта меню

unsigned int counter = 0;

void start(){

//масиви пунктів меню

string menu[] = {"1.Ввести лабіринт", "2.Стандартний лабіринт", "3.Ввести шлях до файлу з лабіринтом", "4.Вихід"};

string algMenu[] = {"1.BFS", "2.TwoEndBFS", "3.Ant algorithm", "4.Вирішити власноруч", "5.Назад"};

unsigned int algExit = 0,

//розміри меню

sizeMenu = sizeof(menu) / sizeof(menu[0]),

sizeAlgMenu = sizeof(algMenu) / sizeof(algMenu[0]);

char key;

while(true){

clearDisplay();

//вивід головного меню

printMenu(menu, sizeMenu);

key = getch();

//вибір пункта

if(key == KEY\_UP && counter>0){ counter--; }

if(key == KEY\_DOWN && counter <sizeMenu-1){ counter++; }

if(key == ENTER){

clearDisplay();

Maze maze;

string path = STD\_PATH;

gotoXY(0, 5);

switch (counter){

case 0:

//ввід та перевірка лабіринту

if(inputMaze(maze) || !isValidMaze(maze)){

pressAnyKeyPrompt();

continue;

}

break;

case 1:

//зчитування та перевірка лабіринту зі стандартного файлу

if(maze.readMaze(path) || !isValidMaze(maze)){

pressAnyKeyPrompt();

continue;

}

break;

case 2:

//зчитування та первірка лабіринту

inputPath(path);

if(!isValidPath(path)){

cout<<"Некоректний шлях до файлу."<<endl;

pressAnyKeyPrompt();

continue;

}

if(maze.readMaze(path) || !isValidMaze(maze)){

pressAnyKeyPrompt();

continue;

}

break;

case 3:

//вихід

color(RESET);

return;

default:

break;

}

counter=0;

algExit=0;

while(!algExit){

clearDisplay();

//вивід меню алгоритмів

printMenu(algMenu, sizeAlgMenu);

key = getch();

//вибір пункта

if(key == KEY\_UP && counter>0){ counter--; }

if(key == KEY\_DOWN && counter<sizeAlgMenu-1){ counter++; }

if(key == '\r'){

//створення вказівника батьківського класу, для використання динамічного поліморфізму

Solver \*solv=nullptr;

clearDisplay();

gotoXY(0, 5);

//створення об'єкта класу наслідника

switch (counter){

case 0:

solv = new Bfs(maze);

break;

case 1:

solv = new TwoEndBfs(maze);

break;

case 2:

solv = new Ant(maze);

break;

case 3:

solv = new UserSolve(maze);

break;

case 4:

counter=0;

algExit=1;

continue;

default:

break;

}

Stopwatch stopwatch;

//запуск секундоміра

stopwatch.start();

//виклик перевантаженої функції

solv->solve();

//зупинка секундоміра та вивід часу

gotoXY(0,1);

cout<<"Час від початку рішення: ";

cout<<stopwatch.stop();

cout<<" секунд"<<endl;

//вивід кількості кроків

cout<<"Кількість кроків: ";

cout<<solv->getStepsCounter()<<endl;

//видалення об'єкта

delete solv;

gotoXY(0, 4);

pressAnyKeyPrompt();

}

}

}

}

}

void printMenu(const string menu[], const unsigned int size){

//якщо номер пункту збігається з counter то він виводиться червоним

for(unsigned int i=0; i<size; i++){

gotoXY(10, 5+i);

if(counter==i){

color(RED);

}else color(RESET);

cout<<menu[i];

}

}

void pressAnyKeyPrompt(){

cout<<"Натисніть будь-яку клавішу..."<<endl;

getch();

}

int isValidPath(const string &path){

//патерн шляху до файлу на відновс

regex pattern(R"(^([a-zA-Z]:\\)?(\\?[a-zA-Z0-9\_.-]+)+\\?$)");

return regex\_match(path, pattern);

}

int isValidMaze(Maze &maze){

//перевірка крайніх стовпців

for (int i = 0; i < maze.rows; i++) {

if (maze.getMaze()[i][0] != 0 || maze.getMaze()[i][maze.columns - 1] != 0) {

cout << "Перший і останній стовпець мають бути заповнені 0." << endl;

maze.deleteMaze();

return 0;

}

}

//перевірка крайніх рядків

for (int j = 0; j < maze.columns; ++j) {

if (maze.getMaze()[0][j] != 0 || maze.getMaze()[maze.rows - 1][j] != 0) {

cout << "Перший і останній рядок мають бути заповнені 0." << endl;

maze.deleteMaze();

return 0;

}

}

//перевірка початкової клітинки

if (maze.getMaze()[1][1] != 1) {

cout << "На початку (1, 1) лабіринту має бути 1." << endl;

maze.deleteMaze();

return 0;

}

return 1;

}

#include "Controller.h"

int main(){

//виклик функції з меню

start();

return 0;

}

#include "Maze.h"

Maze::Maze(){

maze=nullptr;

rows=0;

columns=0;

}

Maze::~Maze(){

if(maze){

deleteMaze();

}

}

//гетер

int \*\*Maze::getMaze() const { return maze; }

//сетер з видаленням попереднього лабіринту якщо він існує

void Maze::setMaze(int \*\* const array){

if(maze){

for(int i=0; i<rows; i++) {

delete[] maze[i];

}

delete[] maze;

}

maze=array;

}

int Maze::readMaze(const string &fileName){

ifstream file;

//відкриття файлу

file.open(fileName);

//перевірка чи відкрився файл

if(!file.is\_open()){

cerr<<"Помилка відкриття файлу."<<endl;

return 1;

}

string line;

//зчитування полінійно файлу для підрахунку рядків та перевірка на недопустимі символи

while(getline(file, line)){

for (char c : line) {

if (c != '0' && c != '1') {

cerr << "Помилка файл містить неприпустимі символи." << endl;

file.close();

return 1;

}

}

rows++;

columns=line.size();

}

//виділення пам'яті для лабіринту

try{

maze=new int\*[rows];

for(int i=0; i<rows; i++){

maze[i]=new int[columns];

}

}catch(const bad\_alloc &e){

cerr<<"Помилка виділення пам'яті: " << e.what() << endl;

if (maze) {

for (int k = 0; k < rows; k++) {

if (maze[k]) {

delete[] maze[k];

}

}

delete[] maze;

}

file.close();

return 1;

}

//зчитування лабіринту з початку файлу в масив

file.clear();

file.seekg(0, ios::beg);

rows=0;

while(getline(file, line)){

for(int i=0; i<columns; i++){

maze[rows][i]=line[i]-'0';

}

rows++;

}

file.close();

return 0;

}

void Maze::deleteMaze(){

for (int k = 0; k < rows; k++) {

delete[] maze[k];

}

delete[] maze;

maze=nullptr;

}

#include "Point.h"

Point::Point(){

x=-1;

y=-1;

}

Point::Point(const int i, const int j){

x=i;

y=j;

}

//гетери

int Point::getX() const { return x; }

int Point::getY() const { return y; }

#include "Queue.h"

Queue::Queue(const int n){

front=0;

rear=0;

capacity=n;

try{

queue=new Point[n];

} catch(const bad\_alloc &e){

cerr<<"Помилка виділення пам'яті: " << e.what() << endl;

throw;

}

}

Queue::~Queue(){

delete[] queue;

queue = nullptr;

}

//додати в чергу

void Queue::enqueue(const Point &p){

if (rear < capacity) {

queue[rear++] = p;

} else {

cerr<<"Черга переповнена.";

}

}

//витягнути з черги

Point Queue::dequeue() {

if (!isEmpty()) {

return queue[front++];

}

cerr<<"Черга порожня.";

return Point(0,0);

}

//перевірка чи пуста черга

bool Queue::isEmpty() const {

return front == rear;

}

#include "Solver.h"

//функція для затримки

void Solver::Sleep(const int millisec) const {

this\_thread::sleep\_for(chrono::milliseconds(millisec));

}

Solver::Solver(Maze &maze){

//починаємо з нуля кроків

stepsCounter=0;

//перевірка на ініціалізацію

if(!maze.getMaze()) {

cerr<<"Лабіринт не ініціалізовано."<<endl;

this->maze=nullptr;

result=nullptr;

} else {

//запис посилання на лабіринт

this->maze=&maze;

//виділення пам'яті

try{

result=new int\*[maze.rows];

for(int i=0; i<maze.rows; i++){

result[i]=new int[maze.columns]();

for(int j=0; j<maze.columns; j++){

result[i][j]=maze.getMaze()[i][j];

}

}

} catch(const bad\_alloc &e){

cerr<<"Помилка виділення пам'яті: " << e.what() << endl;

//очищення виділеної пам'яті в разі помилки

if (result) {

for (int k = 0; k < maze.rows; k++) {

if (result[k]) {

delete[] result[k];

}

}

delete[] result;

result=nullptr;

}

}

}

}

Solver::~Solver(){

if(result){

for (int i = 0; i < maze->rows; i++) {

delete[] result[i];

}

delete[] result;

result = nullptr;

}

}

int Solver::getStepsCounter(){

return stepsCounter;

}

#include "Stopwatch.h"

Stopwatch::Stopwatch(){

sec = 0;

running = 0;

}

//запустити секундомір

void Stopwatch::start(){

if (!running) {

startTime = time(NULL);

running = 1;

} else {

cout << "Секундомір вже запущений." << endl;

}

}

//зупинити секундомір

int Stopwatch::stop(){

if (running) {

sec = time(nullptr)-startTime;

running = 0;

return sec;

} else {

cout << "Секундомір не запущений." << endl;

return 0;

}

}

#include "TwoEndBfs.h"

TwoEndBfs::TwoEndBfs(Maze &maze) : Solver(maze) {

//перевірка на ініціалізацію

if(!maze.getMaze()) {

distances=nullptr;

distancesEnd=nullptr;

visited=nullptr;

} else {

//виділення пам'яті

try{

distances=new int\*[maze.rows];

distancesEnd=new int\*[maze.rows];

visited=new int\*[maze.rows];

for(int i=0; i<maze.rows; i++){

distances[i]=new int[maze.columns]();

distancesEnd[i]=new int[maze.columns]();

visited[i]=new int[maze.columns]();

for(int j=0; j<maze.columns; j++){

distances[i][j]=INFINITY;

distancesEnd[i][j]=INFINITY;

visited[i][j]=false;

}

}

} catch(const bad\_alloc &e){

cerr<<"Помилка виділення пам'яті: " << e.what() << endl;

//очищення виділеної пам'яті в разі помилки

deleteArray(&distances);

deleteArray(&distancesEnd);

deleteArray(&visited);

}

}

}

TwoEndBfs::~TwoEndBfs(){

deleteArray(&distances);

deleteArray(&distancesEnd);

deleteArray(&visited);

}

void TwoEndBfs::deleteArray(int\*\*\* array){

if (\*array) {

for (int k = 0; k < maze->rows; k++) {

if ((\*array)[k]) {

delete[] (\*array)[k];

}

}

delete[] \*array;

\*array = nullptr;

}

}

int TwoEndBfs::solve(){

//перевірка на ініціалізацію

if(!maze) { return 1; }

//вивід лабіринту

printMaze(maze);

//створення черги

Queue queue(maze->rows \* maze->columns);

//початкова точка

Point src(1, 1);

//кінцева точка точка

Point dst(maze->rows-2, maze->columns-2);

//додавання початкової точки в чергу

queue.enqueue(src);

//додавання кінцевої точки в чергу

queue.enqueue(dst);

//призначення 0 як відстані до початкової та кінцевої точок

//позначення їх як відвіданих, 1 для точок, що почалися з початку, 2 для точок, що почалися з кінця

distances[src.getX()][src.getY()] = 0;

visited[src.getX()][src.getY()] = 1;

gotoXY(1 + src.getY() \* 2, 5 + src.getX());

cout << distances[src.getX()][src.getY()];

distancesEnd[dst.getX()][dst.getY()] = 0;

visited[dst.getX()][dst.getY()] = 2;

gotoXY(1 + dst.getY() \* 2, 5 + dst.getX());

cout << distancesEnd[dst.getX()][dst.getY()];

int intersection = 0;

int currentX, currentY, currentEndX, currentEndY;

//цикл, доки черга не пуста або bfs не перетнуться

while (!queue.isEmpty() && !intersection) {

//береться точка з черги

Point current = queue.dequeue();

//вибір масивів і позначень відвідуваності в залежності від bfs початку чи кінця

int \*\*currentDistances = (visited[current.getX()][current.getY()] == 1) ? distances : distancesEnd;

int flag = (visited[current.getX()][current.getY()] == 1) ? 1 : 2;

int otherFlag = (visited[current.getX()][current.getY()] == 1) ? 2 : 1;

//цикл для всіх сусідніх клітинок

for (int i = 0; i < 4; i++) {

int newX = current.getX() + moveX[i];

int newY = current.getY() + moveY[i];

//перевірка чи можливий рух в сусідню клітинку та чи не була вона відвідана

if (maze->getMaze()[newX][newY] == 1) {

//якщо точка була відвідана іншим bfs, то bfs перетнулися

if (visited[newX][newY] != otherFlag) {

if (visited[newX][newY] == 0) {

//затримка

Sleep(SLEEP\_TIME);

//збільшуємо кількість кроків на одиницю

stepsCounter++;

//для сусідньої клітинки відстань збільшуємо на одиницю

currentDistances[newX][newY] = currentDistances[current.getX()][current.getY()] + 1;

//позначаємо як відвідану

visited[newX][newY] = flag;

//виводимо відстань

gotoXY(1 + newY \* 2, 5 + newX);

cout << currentDistances[newX][newY];

//додаємо точку в чергу

queue.enqueue(Point(newX, newY));

}

} else {

//запам'ятовування точки перетину

currentX=newX;

currentY=newY;

currentEndX=newX;

currentEndY=newY;

intersection=1;

}

}

}

}

//якщо не пертнулися, а черга порожня, то рішення не існує

if(!intersection) {

gotoXY(0, 3);

cout<<"Рішення не існує."<<endl;

return 1;

}

//вивід шляху

drawPath(distances, currentX, currentY);

drawPath(distancesEnd, currentEndX, currentEndY);

return 0;

}

void TwoEndBfs::drawPath(int \*\*dist, int curX, int curY) const{

result[curX][curY]=2;

gotoXY(1+curY\*2, 5+curX);

color(RED);

cout<<PATH;

color(RESET);

//допоки не дійшли до стартової точки для bfs, відстань до якої 0

while(dist[curX][curY]!=0){

//цикл для всіх сусідніх клітинок

for(int i=0; i<4; i++){

int newX=curX+moveX[i];

int newY=curY+moveY[i];

//перевірка чи відстань до сусідньої клітинки менша

if(dist[newX][newY]<dist[curX][curY]){

//позначаємо та виводимо шлях

result[newX][newY]=2;

curX=newX;

curY=newY;

gotoXY(1+curY\*2, 5+curX);

color(RED);

cout<<PATH;

color(RESET);

//затримка для покрокового виводу

Sleep(SLEEP\_TIME);

}

}

}

}

#include "UserSolve.h"

UserSolve::UserSolve(Maze &maze) : Solver(maze) {}

int UserSolve::solve(){

//перевірка на ініціалізацію

if(!maze) { return 1; }

char key;

//вивід підказок по управлінню

gotoXY(0, 4);

cout<<"\u2190 \u2191 \u2192 \u2193, двічі 'е'-вихід"<<endl;

//вивід лабіринту

gotoXY(0, 5);

printMaze(maze);

int x=1;

int y=1;

gotoXY(1+y\*2, 5+x);

color(RED);

cout<<PATH;

color(RESET);

//цикл допоки лабіринт не пройдений

while(result[maze->rows-2][maze->columns-2] != 2){

//ігнорування зайвого символу

getch();

//зчитування клавіші та переміщення у відповідному напрямку, якщо можливо

key=getch();

if(key == 'e'){break;}

if(key == KEY\_UP && result[x-1][y]==1){x--;}

if(key == KEY\_DOWN && result[x+1][y]==1){x++;}

if(key == KEY\_LEFT && result[x][y-1]==1){y--;}

if(key == KEY\_RIGHT && result[x][y+1]==1){y++;}

//вивід шляху

result[x][y]=2;

gotoXY(1+y\*2, 5+x);

color(RED);

cout<<PATH;

color(RESET);

//збільшуємо кількість кроків на одиницю

stepsCounter++;

}

//в разі проходження лабіринту вивід повідомлення

if(result[maze->rows-2][maze->columns-2] == 2){

gotoXY(0, 3);

cout<<"Ви пройшли лабіринт!";

}

return 0;

}

#include "View.h"

void printMaze(const Maze \*maze) {

//перевірка чи існує масив лабіринту

if(!maze->getMaze()){ return; }

//кольоровий вивід символа в залежності від числа в клітинці

//0-стіна, 1-прохід, 2-шлях

for(int i=0; i<maze->rows; i++){

for(int j=0; j<maze->columns; j++){

switch (maze->getMaze()[i][j]){

case 0:

cout<<WALL;

break;

case 1:

cout<<" ";

break;

case 2:

color(RED);

cout<<PATH;

color(RESET);

break;

default:

break;

}

}

cout<<endl;

}

return;

}

//переміщення курсора в консолі

void gotoXY(const int x, const int y){

cout << "\033[" << y << ";" << x << "H";

}

//ввід шляху до файлу

void inputPath(string &path){

cout<<"Введіть шлях до файлу з лабіринтом: ";

cin>>path;

resetInput();

}

//очищення консолі

void clearDisplay(){ cout<<"\x1B[2J"; }

//зміна кольору виводу

void color(const Color cl){

switch (cl){

case RESET:

cout << "\033[0m";

break;

case RED:

cout << "\033[31m";

break;

default:

cout << "\033[0m";

break;

}

}

int inputMaze(Maze &maze){

//ввід кількості рядків та стовпців

cout << "Введіть кількість рядків в лабіринті: ";

while (!(cin >> maze.rows) || maze.rows <= 0) {

cout << "Неправильний ввід. Введіть позитивне число: ";

resetInput();

}

resetInput();

cout << "Введіть кількість стовпців в лабіринті: ";

while (!(cin >> maze.columns) || maze.columns <= 0) {

cout << "Неправильний ввід. Введіть позитивне число: ";

resetInput();

}

resetInput();

//виділення пам'яті

int\*\* newMaze;

try{

newMaze = new int\*[maze.rows];

for (int i = 0; i < maze.rows; i++) {

newMaze[i] = new int[maze.columns];

}

}catch(const bad\_alloc &e){

cerr<<"Помилка виділення пам'яті: " << e.what() << endl;

//очищення виділеної пам'яті в разі помилки

if (newMaze) {

for (int k = 0; k < maze.rows; k++) {

if (newMaze[k]) {

delete[] newMaze[k];

}

}

delete[] newMaze;

}

return 1;

}

//перевірка на ввід некориктних символів

cout << "Введіть лабіринт (1 прохід, 0 стіна):" << endl;

for (int i = 0; i < maze.rows; i++) {

for (int j = 0; j < maze.columns; j++) {

while (!(cin >> newMaze[i][j]) || (newMaze[i][j] != 0 && newMaze[i][j] != 1)) {

cout << "Неправильний ввід. Введіть 0 або 1: ";

resetInput();

}

}

}

//збереження лабіринту

maze.setMaze(newMaze);

resetInput();

return 0;

}

void resetInput(){

//очищення помилок

cin.clear();

//ігнорування зайвих символів

cin.ignore(10000, '\n');

}

# ДОДАТОК 3

## Опис програми

НТУУ «КПІ ім.І.Сікорського» ННІАТЕ ІПЗЕ ТВ-31

Листів 3

Київ – 2024

Таблиця 2 – Опис функцій програми

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Назва функції | Вхідні дані | Вихідні дані | Призначення |
| 1 | solve() | - | 0, але якщо помилка, то 1 | Запуск алгоритму класу |
| 2 | deleteArray() | Зміна, яку потрібно видалити | - | Звільняє пам’ять |
| 3 | drawPath() | - | - | Виводить шлях |
| 4 | start() | - | - | Взаємодія з меню та виконання дій |
| 5 | printMenu() | Масив меню та його розмір | - | Виводить меню |
| 6 | pressAnyKeyPrompt() | - | - | Зупинити програму в очікуванні натискання клавіші |
| 7 | isValidPath() | Посилання на шлях | Якщо шлях некоректний 0, інакше 1 | Перевіряє шлях на валідність |
| 8 | isValidMaze() | Посилання на лабіринт | Якщо лабіринт некоректний 0, інакше 1 | Перевіряє лабіринт на правильність |
| 9 | getMaze() | - | Змінна, що зберігає лабіринт | Гетер змінної, що зберігає лабіринту |

Продовження таблиці 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Назва функції | Вхідні дані | Вихідні дані | Призначення |
| 10 | setMaze() | Посилання на посилання на ціле число | - | Сетер змінної збереження лабіринту |
| 11 | readMaze() | Посилання на шлях до файлу | 0, але якщо помилка, то 1 | Зчитує лабіринт з файлу |
| 12 | deleteMaze() | - | - | Видаляє змінну збереження лабіринту |
| 13 | getX() | - | X координату | Повертає координату точки |
| 14 | getY() | - | Y координату | Повертає координату точки |
| 15 | enqueue() | Посилання на точка | - | Додає точку в чергу |
| 16 | dequeue() | - | Точка | Витягує точку з чергу |
| 17 | isEmpty() | - | Булеве значення | Повертає true, якщо черга порожня, інакше false |
| 18 | Sleep() | Мілісекунди | - | Зупиняє програму на отриманий час |
| 19 | getStepsCounter() | - | Кількість кроків | Гетер кількості кроків алгоритму |

Продовження таблиці 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Назва функції | Вхідні дані | Вихідні дані | Призначення |
| 20 | start() | - | - | Запуск секундоміру |
| 21 | stop() | - | Кількість секунд, що пройшла з запуску | Зупиняє секундомір |
| 22 | drawPath() | Відстань, координати перетину | - | Виводить шлях |
| 23 | printMaze() | Вказівник на лабіринт | - | Виводить лабіринт у консоль |
| 24 | gotoXY() | Координати | - | Переміщує курсор |
| 25 | inputPath() | Посилання на шлях | - | Зчитує шлях з консолі |
| 26 | clearDisplay() | - | - | Очищає консоль |
| 27 | color() | Колір enum | - | Змінює колір виводу |
| 28 | inputMaze() | Посилання на лабіринт | 0, але якщо помилка, то 1 | Ввід лабіринту з консолі |
| 29 | resetInput() | - | - | Очищення потоку вводу |