BỘ GIÁO DỤC VÀ ĐÀO TẠO

**TRƯỜNG ĐẠI HỌC NHA TRANG**

**KHOA CÔNG NGHỆ THÔNG TIN**

A blue and black logo

Description automatically generated

**BÁO CÁO THỰC TẬP CƠ SỞ**

**PHÁT TRIỂN TRÒ CHƠI FLOWFREE**

**TÍCH HỢP GIẢI TỰ ĐỘNG SỬ DỤNG BFS**

**VÀ QUY HOẠCH ĐỘNG BITMASK**

**Giảng viên hướng dẫn: Th.S Bùi Chí Thành**

**Sinh viên thực hiện: Nguyễn Quang Vinh**

**Mã số sinh viên: 64132989**

Khánh Hoà – 2025

BỘ GIÁO DỤC VÀ ĐÀO TẠO

**TRƯỜNG ĐẠI HỌC NHA TRANG**

**KHOA CÔNG NGHỆ THÔNG TIN**

A blue and black logo

Description automatically generated

**BÁO CÁO THỰC TẬP CƠ SỞ**

**PHÁT TRIỂN TRÒ CHƠI FLOWFREE   
TÍCH HỢP GIẢI TỰ ĐỘNG SỬ DỤNG BFS   
VÀ QUY HOẠCH ĐỘNG BITMASK**

GVHD: Th.S Bùi Chí Thành

SVTH: Nguyễn Quang Vinh

MSSV: 64132989

Khánh Hoà – Tháng 1/2025

|  |  |
| --- | --- |
| TRƯỜNG ĐẠI HỌC NHA TRANG  **KHOA CÔNG NGHỆ THÔNG TIN** |  |

**PHIẾU THEO DÕI TIẾN ĐỘ VÀ ĐÁNH GIÁ THỰC TẬP CƠ SỞ**   
**(Dùng cho CBHD và nộp cùng báo cáo thực tập cơ sở của sinh viên)**

**Tên đề tài:** Phát triển trò chơi FlowFree tích hợp giải tự động sử dụng BFS và Quy hoạch động Bitmask

**Giảng viên hướng dẫn:** Th.S.Bùi Chí Thành

**Sinh viên được hướng dẫn:** Nguyễn Quang Vinh **MSSV:** 64132989

**Khóa:** 64 **Ngành:** Công nghệ thông tin

|  |  |  |  |
| --- | --- | --- | --- |
| ***Lần KT*** | ***Ngày*** | ***Nội dung*** | ***Nhận xét của GVHD*** |
| 1 | 07/12/2024 | Hoàn thành việc xây dựng cơ bản trò chơi FlowFree. |  |
| 2 | 14/12/2024 | Nghiên cứu và lựa chọn thuật toán phù hợp để tự động giải các màn chơi FlowFree |  |
| 3 | 21/12/2024 | Hiện thực thuật toán đã chọn vào trò chơi FlowFree. |  |
| 4 | 28/12/2024 | Tích hợp tất cả các thành phần và hoàn thiện sản phẩm. |  |
| 5 | 04/01/2024 | Hoàn thành báo cáo, gửi đến GVHD để nhận xét |  |
| 6 |  |  |  |

**Nhận xét chung** (sau khi sinh viên hoàn thành thực tập cơ sở):

Điểm hình thức:……/10 Điểm nội dung:......./10 **Điểm tổng kết**:………/10

Kết luận sinh viên: Được bảo vệ: Không được bảo vệ:

|  |  |  |
| --- | --- | --- |
|  | | *Khánh Hòa, ngày…….tháng…….năm……*  **Cán bộ hướng dẫn**  *(Ký và ghi rõ họ tên)*  **Bùi Chí Thành** |
| TRƯỜNG ĐẠI HỌC NHA TRANG  **KHOA CÔNG NGHỆ THÔNG TIN** |  | |

**PHIẾU CHẤM ĐIỂM THỰC TẬP CƠ SỞ**   
**(Dành cho cán bộ chấm phản biện)**

**1. Họ tên người chấm:**

**2. Sinh viên thực hiện:** **MSSV:**

**3. Tên đề tài:**

**4. Nhận xét**

a) Kết quả thực hiện đề tài

...............................................................................................................................................

...............................................................................................................................................

...............................................................................................................................................

b) Báo cáo thực tập

- Hình thức:

- Nội dung:

Điểm hình thức:……/10 Điểm nội dung:......./10 **Điểm tổng kết**:………/10

|  |  |
| --- | --- |
|  | *Khánh Hòa, ngày…….tháng…….năm……*  **Cán bộ chấm phản biện**  *(Ký và ghi rõ họ tên)* |

MỤC LỤC

[MỤC LỤC iii](#_Toc188055052)

[DANH MỤC HÌNH ẢNH iv](#_Toc188055053)

[Chương 1. TỔNG QUAN VỀ ĐỀ TÀI 1](#_Toc188055054)

[1.1. Trò chơi FlowFree 1](#_Toc188055055)

[1.2. Hướng phát triển đề tài 1](#_Toc188055056)

[1.3. Các công nghệ và thuật toán sử dụng trong đề tài 2](#_Toc188055057)

[Chương 2. CƠ SỞ LÝ THUYẾT 3](#_Toc188055058)

[2.1. Ngôn ngữ lập trình Python và lập trình hướng đối tượng 3](#_Toc188055059)

[2.2. Thư viện PyQT5 3](#_Toc188055060)

[2.2. Thuật toán tìm kiếm theo chiều rộng 4](#_Toc188055061)

[2.3. Quy hoạch động 6](#_Toc188055062)

[2.4. Quy hoạch động Bitmask 8](#_Toc188055063)

[Chương 3. CÀI ĐẶT ỨNG DỤNG 11](#_Toc188055064)

[3.1. Thiết kế ứng dụng 11](#_Toc188055065)

[3.2. Thuật toán tự động giải màn chơi 15](#_Toc188055066)

[3.3. Thiết kế giao diện 19](#_Toc188055067)

[3.4. Xử lý tự động giải 21](#_Toc188055068)

[3.5. Hướng dẫn sử dụng 23](#_Toc188055069)

[Chương 4. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN 24](#_Toc188055070)

[4.1. Kết quả đạt được 24](#_Toc188055071)

[4.2. Hạn chế của ứng dụng 25](#_Toc188055072)

[4.3. Hướng phát triển 25](#_Toc188055073)

[TÀI LIỆU THAM KHẢO 26](#_Toc188055074)

DANH MỤC HÌNH ẢNH

[Hình 1. Trò chơi FlowFree 1](#_Toc187290248)

[Hình 2. Cấu trúc file trong ứng dụng và mối quan hệ 11](#_Toc187290249)

[Hình 3. Sơ đồ lớp của ứng dụng 12](#_Toc187290250)

[Hình 4. Kết quả tự giải màn chơi khi hiện trong ứng dụng 17](#_Toc187290251)

[Hình 5. Giao diện ứng dụng 20](#_Toc187290252)

[Hình 6. Thư mục ứng dụng sau khi giải nén 23](#_Toc187290253)

[Hình 7. Giao diện trò chơi 24](#_Toc187290254)

# Chương I. TỔNG QUAN VỀ ĐỀ TÀI

Đề tài này tập trung vào việc phát triển trò chơi Flow Free kết hợp với khả năng tự động giải bằng các thuật toán BFS và Quy hoạch động trạng thái. Chương này sẽ cung cấp cái nhìn tổng quan về trò chơi, hướng phát triển của đề tài, cũng như các công nghệ và thuật toán được sử dụng.

## 1.1. Trò chơi FlowFree

Flow Free là một trò chơi giải đố logic đơn giản nhưng rất hấp dẫn. Mục tiêu của người chơi là nối các cặp điểm cùng màu bằng các đường ống sao cho tất cả các ô trên bảng đều được lấp đầy mà không có bất kỳ đường ống nào chồng chéo lên nhau. Luật chơi rất đơn giản: người chơi chỉ cần kéo đường ống từ một điểm màu đến điểm màu tương ứng. Các đường ống không được cắt nhau và phải lấp đầy toàn bộ bảng. Trò chơi có nhiều cấp độ khó khác nhau, từ các bảng nhỏ đến các bảng lớn hơn với nhiều cặp màu hơn, đòi hỏi người chơi phải có khả năng tư duy logic và lập kế hoạch tốt.

A screenshot of a game

Description automatically generated

Hình . Trò chơi FlowFree

*(Nguồn: Github - Huzecong/FlowFree: Qt Project - Flow Free Game, 2015)*

## 1.2. Hướng phát triển đề tài

Đề tài này hướng đến hai mục tiêu chính. Thứ nhất, xây dựng một phiên bản trò chơi Flow Free hoàn chỉnh với khả năng tùy chỉnh độ khó và các chức năng hỗ trợ trải nghiệm người dùng (UX/UI), đảm bảo phần mềm trực quan và thân thiện. Điều này bao gồm việc thiết kế giao diện hấp dẫn, dễ sử dụng và cung cấp các tùy chọn cho người chơi như lựa chọn kích thước bảng, mức độ khó. Mục tiêu thứ hai và cũng là trọng tâm của đề tài là tích hợp khả năng tự động giải trò chơi và thời gian giải phải ở ngưỡng chấp nhận được (~1s). Hệ thống sẽ được thiết kế để có thể giải bất kỳ màn chơi nào mà người dùng lựa chọn, cung cấp giải pháp một cách tự động.

## 1.3. Các công nghệ và thuật toán sử dụng trong đề tài

Để đạt được các mục tiêu trên, đề tài sẽ sử dụng các công nghệ và thuật toán sau:

### 1.3.1. Công nghệ sử dụng

Đề tài sẽ sử dụng ngôn ngữ lập trình Python kết hợp với phương pháp lập trình hướng đối tượng (OOP). OOP giúp cấu trúc code rõ ràng, dễ bảo trì và mở rộng. Bên cạnh đó, thư viện PyQt5 sẽ được sử dụng để xây dựng giao diện người dùng (GUI). PyQt5 là một framework mạnh mẽ cho phép tạo ra các ứng dụng desktop đa nền tảng với giao diện đẹp mắt và khả năng tùy biến cao.

### 1.3.2. Thuật toán sử dụng

Hai thuật toán chính được sử dụng trong đề tài là thuật toán tìm đường đi BFS (Breadth-First Search - Tìm kiếm theo chiều rộng) và thuật toán Quy hoạch động trạng thái. BFS là một thuật toán duyệt đồ thị được sử dụng để tìm đường đi ngắn nhất giữa hai đỉnh trong đồ thị không trọng số. Trong bối cảnh trò chơi Flow Free, BFS sẽ được áp dụng để tìm đường đi giữa các điểm cùng màu. Thuật toán Quy hoạch động trạng thái sẽ được sử dụng để tối ưu hóa quá trình tìm kiếm giải pháp, đặc biệt là trong các trường hợp bảng chơi phức tạp. Quy hoạch động giúp tránh việc tính toán lại các bài toán con đã được giải trước đó, từ đó tăng hiệu suất của chương trình. Việc kết hợp hai thuật toán này sẽ giúp hệ thống tự động giải tìm ra giải pháp tối ưu cho trò chơi Flow Free.

# Chương II. CƠ SỞ LÝ THUYẾT

## 2.1. Ngôn ngữ lập trình Python và lập trình hướng đối tượng

Python là một ngôn ngữ lập trình bậc cao, thông dịch, đa năng, được thiết kế với ưu điểm mạnh về khả năng dễ đọc, dễ học và cú pháp rõ ràng. Python hỗ trợ nhiều phong cách lập trình, bao gồm lập trình hướng đối tượng (OOP), lập trình thủ tục và lập trình hàm.

Lập trình hướng đối tượng (OOP) là một phương pháp lập trình dựa trên khái niệm "đối tượng", có thể chứa dữ liệu, dưới dạng các trường, và mã, dưới dạng các thủ tục. Các đối tượng được tạo ra từ các "lớp", là các bản thiết kế. OOP tập trung vào việc tổ chức code thành các đơn vị độc lập, giúp code dễ quản lý, tái sử dụng và mở rộng. Các khái niệm chính trong OOP có thể kể đến như lớp (Class) là khuôn mẫu tạo ra đối tượng (Object). Mỗi đối tượng mang thuộc tính (Attribute) (dữ liệu) và phương thức (Method) (hành động). Kế thừa (Inheritance) cho phép lớp con dùng lại của lớp cha. Đa hình (Polymorphism) là khả năng một đối tượng có nhiều dạng. Đóng gói (Encapsulation) bảo vệ dữ liệu bằng cách giới hạn truy cập trực tiếp.

## 2.2. Thư viện PyQT5

PyQt5 là một binding Python cho framework Qt, một framework ứng dụng đa nền tảng mạnh mẽ. PyQt5 cho phép các nhà phát triển tạo ra các ứng dụng desktop với giao diện người dùng phong phú và khả năng tùy biến cao. PyQt5 hỗ trợ nhiều tính năng của Qt, bao gồm widgets, layouts, signals/slots, đồ họa 2D/3D, và nhiều hơn nữa. Các khái niệm quan trọng trong PyQt5 có thể kể đến như Widgets, là các thành phần giao diện người dùng như nút bấm, nhãn, hộp văn bản, v.v. Layouts là các thành phần sắp xếp các widgets trong cửa sổ ứng dụng, hay Signals và Slots: Cơ chế giao tiếp giữa các đối tượng trong ứng dụng. Signals được phát ra khi có một sự kiện xảy ra (ví dụ: click chuột), và slots là các hàm được gọi để xử lý các signals đó. Ví dụ một đoạn mã sử dụng PyQT5:

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QVBoxLayout

def nut\_duoc\_bam():

    print("Nút đã được bấm!")

app = QApplication(sys.argv)

window = QWidget()

window.setWindowTitle("Ví dụ PyQt5")

layout = QVBoxLayout()

nut = QPushButton("Bấm vào đây")

# Kết nối signal clicked với slot nut\_duoc\_bam

nut.clicked.connect(nut\_duoc\_bam)

layout.addWidget(nut)

window.setLayout(layout)

window.show()

sys.exit(app.exec\_())

Đầu tiên, mã nhập các module cần thiết từ thư viện PyQt5, bao gồm QApplication (quản lý ứng dụng), QWidget (cửa sổ), QPushButton (nút bấm) và QVBoxLayout (bố cục theo chiều dọc). Hàm nut\_duoc\_bam() được định nghĩa để in thông báo khi nút được nhấp.

Sau đó, một đối tượng QApplication được tạo để quản lý vòng lặp sự kiện của ứng dụng. Một cửa sổ chính (QWidget) được tạo và tiêu đề của nó được đặt thành "Ví dụ PyQt5". Một bố cục theo chiều dọc (QVBoxLayout) được tạo để sắp xếp các widget bên trong cửa sổ. Một nút bấm (QPushButton) với nhãn "Bấm vào đây" được tạo và được thêm vào bố cục.

Điểm quan trọng là dòng mã nut.clicked.connect(nut\_duoc\_bam). Dòng này kết nối tín hiệu clicked của nút bấm với hàm nut\_duoc\_bam(). Điều này có nghĩa là mỗi khi nút được nhấp, hàm nut\_duoc\_bam() sẽ được gọi.

Cuối cùng, bố cục được đặt cho cửa sổ, cửa sổ được hiển thị và ứng dụng bắt đầu vòng lặp sự kiện của nó bằng app.exec\_(). sys.exit() được sử dụng để đảm bảo ứng dụng thoát đúng cách khi cửa sổ được đóng.

## 2.2. Thuật toán tìm kiếm theo chiều rộng

Tìm kiếm theo chiều rộng (Breadth-First Search hay BFS) là một thuật toán duyệt đồ thị hoặc cây. Về cơ bản, cách áp dụng BFS cho cây và đồ thị là tương tự nhau. Điểm khác biệt chính là đồ thị có thể chứa chu trình (đường đi khép kín), dẫn đến việc duyệt lại một nút nhiều lần. Để tránh điều này, BFS sử dụng một mảng boolean (hoặc tập hợp) để theo dõi các nút đã được "ghé thăm". BFS sử dụng cấu trúc dữ liệu hàng đợi (queue) để quản lý quá trình duyệt và đặc biệt hữu ích trong việc tìm đường đi ngắn nhất (tính theo số cạnh) trong đồ thị.

Một cách triển khai BFS tiêu chuẩn sẽ phân loại mỗi đỉnh của đồ thị thành hai trạng thái: "đã thăm" (visited) và "chưa thăm" (not visited). Mục tiêu của thuật toán là đánh dấu tất cả các đỉnh là "đã thăm" để tránh việc duyệt lại các đỉnh, đặc biệt là trong đồ thị có chu trình. Các bước thực hiện như sau:

**Bước 1.** Khởi đầu: Chọn một nút làm điểm bắt đầu và đưa nó vào hàng đợi. Ngay lập tức, nút này được đánh dấu là "đã duyệt".

**Bước 2.** Vòng lặp chính, sẽ thực hiện các bước sau cho đến khi hàng đợi trở nên trống rỗng:

**Bước 2.1.** Lấy nút từ hàng đợi: Lấy nút ở đầu hàng đợi ra khỏi hàng đợi.

**Bước 2.2.** Xác định các nút lân cận: Tìm tất cả các nút kết nối trực tiếp với nút vừa được lấy ra (các nút kề).

**Bước 2.3.** Thêm nút lân cận vào hàng đợi: Đối với mỗi nút lân cận chưa được duyệt, đánh dấu nó là "đã duyệt" và thêm nó vào cuối hàng đợi.

**Lưu ý quan trọng:** Một đồ thị có thể chứa nhiều thành phần liên thông riêng biệt (các phần không kết nối với nhau). Để đảm bảo duyệt qua toàn bộ đồ thị, cần kiểm tra và thực hiện BFS trên từng thành phần liên thông nếu cần. Nói cách khác, nếu sau khi chạy BFS từ một đỉnh mà vẫn còn đỉnh chưa được thăm, hãy chọn một đỉnh chưa được thăm khác và tiếp tục chạy BFS từ đỉnh đó.

Trong trò chơi FlowFree, mục tiêu là nối các cặp điểm cùng màu bằng đường ống ngắn nhất, không chồng chéo. BFS phù hợp vì nó luôn tìm đường đi ngắn nhất đầu tiên và duyệt một cách có hệ thống, giúp tránh tạo ra các đường đi phức tạp, chồng chéo, điều mà DFS (Tìm kiếm theo chiều sâu) dễ gặp phải do cách duyệt sâu vào từng nhánh. Vì vậy, BFS giúp giải trò chơi FlowFree hiệu quả hơn. Ví dụ dưới đây đoạn mã Python cài đặt thuật toán BFS trên ma trận (Tìm đường đi trong mê cung):

from collections import deque

def is\_valid(row, col, rows, cols):

    return 0 <= row < rows and 0 <= col < cols

def bfs\_matrix(matrix, start\_row, start\_col, end\_row, end\_col):

    rows = len(matrix)

    cols = len(matrix[0])

    visited = set()

    queue = deque([(start\_row, start\_col)])

    visited.add((start\_row, start\_col))

    while queue:

        row, col = queue.popleft()

        if row == end\_row and col == end\_col:

            return True

        # Các hướng di chuyển (lên, xuống, trái, phải)

        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        for dr, dc in directions:

            nr, nc = row + dr, col + dc

            if is\_valid(nr, nc, rows, cols) and matrix[nr][nc] == 0 and (nr, nc) not in visited: # 0 là đường đi, 1 là tường

                queue.append((nr, nc))

                visited.add((nr, nc))

    return False

matrix = [

    [0, 0, 1, 0],

    [1, 0, 0, 1],

    [0, 0, 0, 0],

    [1, 1, 0, 0]

]

if bfs\_matrix(matrix, 0, 0, 3, 3):

  print("\nĐến được đích.") # Output: Đến được đích.

else:

  print("\nKhông đến được đích.")

## 2.3. Quy hoạch động

Quy hoạch động (Dynamic Programing hay DP) là một phương pháp thiết kế thuật toán được sử dụng để giải quyết các bài toán tối ưu bằng cách phân rã bài toán lớn thành các bài toán con nhỏ hơn, chồng chéo nhau. Thay vì tính toán lại các bài toán con nhiều lần, quy hoạch động lưu trữ kết quả của các bài toán con đã được giải quyết vào một bảng (thường là mảng hoặc bảng băm) để có thể tái sử dụng chúng một cách hiệu quả. Hai đặc điểm quan trọng để một bài toán có thể áp dụng quy hoạch động là:

Cấu trúc con tối ưu (Optimal Substructure): Một bài toán có cấu trúc con tối ưu nếu nghiệm tối ưu của bài toán lớn có thể được xây dựng từ các nghiệm tối ưu của các bài toán con của nó.

Các bài toán con chồng chéo (Overlapping Subproblems): Một bài toán có các bài toán con chồng chéo nếu việc giải bài toán lớn bao gồm việc giải cùng một bài toán con nhiều lần.

Ví dụ với bài toán Dãy con chung dài nhất , đề bài cho hai chuỗi ký tự và . Tìm độ dài của dãy con chung dài nhất của hai chuỗi này. Dãy con không nhất thiết phải liên tiếp nhau trong chuỗi gốc.

Ví dụ:

= "ABCBDAB"

= "BDCABA"

Dãy con chung dài nhất là "BCBA" với độ dài 4.

**Cách giải bằng DP:**

Chúng ta sử dụng một bảng dp hai chiều, trong đó lưu độ dài dãy con chung dài nhất của và .

**Công thức truy hồi:**

Nếu thì (ký tự cuối cùng của hai chuỗi giống nhau, ta tăng độ dài LCS lên 1).

Nếu thì (ta lấy độ dài LCS lớn nhất trong hai trường hợp: bỏ ký tự cuối của hoặc bỏ ký tự cuối của ).

**Cài đặt thuật toán với Python:**

def lcs(s1, s2):

    n = len(s1)

    m = len(s2)

    dp = [[0] \* (m + 1) for \_ in range(n + 1)]

    for i in range(1, n + 1):

        for j in range(1, m + 1):

            if s1[i - 1] == s2[j - 1]:

                dp[i][j] = dp[i - 1][j - 1] + 1

            else:

                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[n][m]

s1 = "ABCBDAB"

s2 = "BDCABA"

print(f"Độ dài dãy con chung dài nhất: {lcs(s1, s2)}")  # Output: 4

## 2.4. Quy hoạch động Bitmask

Quy hoạch động trạng thái (Bitmask Dynamic Programing hay Bitmask DP) là một kỹ thuật mạnh mẽ trong thiết kế thuật toán, thường được sử dụng để giải quyết các bài toán tối ưu liên quan đến tập hợp con. Điểm đặc biệt của QHĐTT là cách biểu diễn trạng thái của bài toán bằng một tập hợp các bit, thường được mã hóa thành một số nguyên. Mỗi bit trong số nguyên này đại diện cho sự hiện diện hoặc vắng mặt của một phần tử trong tập hợp.

Trong QHĐTT, chúng ta biểu diễn mỗi trạng thái của bài toán bằng một số nguyên, gọi là "mask" (mặt nạ). Mỗi bit trong biểu diễn nhị phân của số nguyên này tương ứng với một phần tử trong tập hợp.

Nếu bit thứ i của mask là 1, điều đó có nghĩa là phần tử thứ i thuộc tập hợp con đang được xét.

Nếu bit thứ i của mask là 0, điều đó có nghĩa là phần tử thứ i không thuộc tập hợp con đang được xét.

Ví dụ:

Giả sử chúng ta có một tập hợp gồm 4 phần tử {A, B, C, D}.

mask = 0 (0000 trong hệ nhị phân): Tập hợp con rỗng {}.

mask = 1 (0001 trong hệ nhị phân): Tập hợp con {D}.

mask = 3 (0011 trong hệ nhị phân): Tập hợp con {C, D}.

mask = 8 (1000 trong hệ nhị phân): Tập hợp con {A}.

mask = 15 (1111 trong hệ nhị phân): Tập hợp con {A, B, C, D}.

Quá trình áp dụng thuật toán Quy hoạch động trạng thái thường bao gồm các bước sau:

**Bước 1.** Xác định trạng thái: Bước đầu tiên là biểu diễn trạng thái của bài toán bằng một mask thích hợp.

**Bước 2.** Xây dựng công thức truy hồi: Bước quan trọng nhất là xác định mối quan hệ giữa các trạng thái khác nhau.

**Bước 3.** Khởi tạo giá trị ban đầu: Xác định giá trị của các trạng thái cơ sở, tức là các trạng thái đơn giản nhất mà từ đó ta có thể xây dựng các trạng thái phức tạp hơn.

**Bước 4.** Tính toán kết quả: Sử dụng công thức truy hồi đã xây dựng để tính toán giá trị cho tất cả các trạng thái, thường bắt đầu từ các trạng thái cơ sở và tiến dần đến trạng thái đích.

**Lưu ý:** Chỉ hiệu quả với số lượng phần tử nhỏ (thường là ) do độ phức tạp không gian là . Khó áp dụng cho các bài toán mà trạng thái không thể biểu diễn bằng bitmask một cách hiệu quả.

Ví dụ với bài toán Người du lịch (TSP) với số lượng thành phố nhỏ: Cho n thành phố và ma trận khoảng cách giữa chúng. Tìm chu trình Hamilton (đi qua tất cả các thành phố đúng một lần và quay trở lại điểm bắt đầu) có tổng khoảng cách nhỏ nhất.

**Giới hạn:** Ví dụ này giả sử n nhỏ (ví dụ:) để có thể sử dụng QHĐTT.

Cách giải bằng QHĐTT như sau:

Chúng ta sử dụng một bảng dp hai chiều, trong đó:

dp[mask][i] là chi phí nhỏ nhất để đi qua các thành phố được biểu diễn bởi mask (mask là một số nguyên, bit thứ j là 1 nếu thành phố j đã được thăm) và kết thúc ở thành phố i.

Công thức truy hồi: dp[mask][i] = min(dp[mask ^ (1 << i)][j] + dist[j][i]) với mọi j sao cho bit thứ j trong mask là 1 và j != i.

import sys

def tsp(dist):

    n = len(dist)

    dp = [[sys.maxsize] \* n for \_ in range(1 << n)]

    # Khởi tạo giá trị cho trạng thái ban đầu (chỉ thăm thành phố 0)

    for i in range(n):

        dp[1 << i][i] = dist[0][i] if i != 0 else 0

    # Duyệt qua tất cả các trạng thái

    for mask in range(1, 1 << n):

        for i in range(n):

            if (mask & (1 << i)): # Kiểm tra bit thứ i có được bật hay không

                for j in range(n):

                    if (mask & (1 << j)) and j != i:

                        dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + dist[j][i])

    #Tìm đường đi trở về điểm bắt đầu

    min\_cost = sys.maxsize

    for i in range(n):

        min\_cost = min(min\_cost, dp[(1<<n)-1][i] + dist[i][0])

    return min\_cost

# Ví dụ:

dist = [

    [0, 2, 9, 10],

    [1, 0, 6, 4],

    [15, 7, 0, 8],

    [6, 3, 12, 0]

]

print(f"Chi phí tối thiểu cho TSP: {tsp(dist)}") # Output: 21

**Giải thích:**

* mask: Biểu diễn trạng thái các thành phố đã được thăm.
* dp[mask][i]: Chi phí tối thiểu để đến thành phố i sau khi đã thăm các thành phố trong mask.
* Công thức truy hồi tìm chi phí tối thiểu bằng cách duyệt qua tất cả các thành phố j đã được thăm trước đó và chọn đường đi có chi phí nhỏ nhất.
* Trong code trên, vòng lặp cuối cùng dùng để tìm đường đi trở về điểm xuất phát (thành phố 0) và tính chi phí cuối cùng.

# Chương III. CÀI ĐẶT ỨNG DỤNG

## 3.1. Thiết kế ứng dụng

Phần này mô tả kiến trúc của ứng dụng Flow Free, tập trung vào cấu trúc tĩnh (các lớp và mối quan hệ giữa chúng) và một số tương tác động quan trọng. Kiến trúc được thiết kế theo hướng đối tượng, với các lớp được phân chia rõ ràng để đảm bảo tính module hóa, dễ bảo trì và mở rộng.

### 3.1.1. Kiến trúc tổng quan

Ứng dụng được cấu trúc thành các thành phần chính sau:

**Giao diện người dùng (UI):** Được quản lý bởi lớp MainWindow, chịu trách nhiệm hiển thị trò chơi và tương tác với người dùng thông qua các widget của PyQt5.

**Logic trò chơi:** Được xử lý bởi lớp GameController, chứa các quy tắc của trò chơi, xử lý sự kiện người dùng và cập nhật trạng thái trò chơi.

**Hiển thị đồ họa:** Được đảm nhiệm bởi lớp GameRenderer, sử dụng QPainter để vẽ các thành phần đồ họa của trò chơi.

**Quản lý dữ liệu:** Được thực hiện bởi lớp GridData, lưu trữ thông tin về lưới trò chơi, các điểm đầu và đường đi.

**Thuật toán giải:** Được hiện thực trong hàm Slove (chính xác là Solve), sử dụng thuật toán tìm kiếm theo chiều rộng (BFS) để giải bài toán.

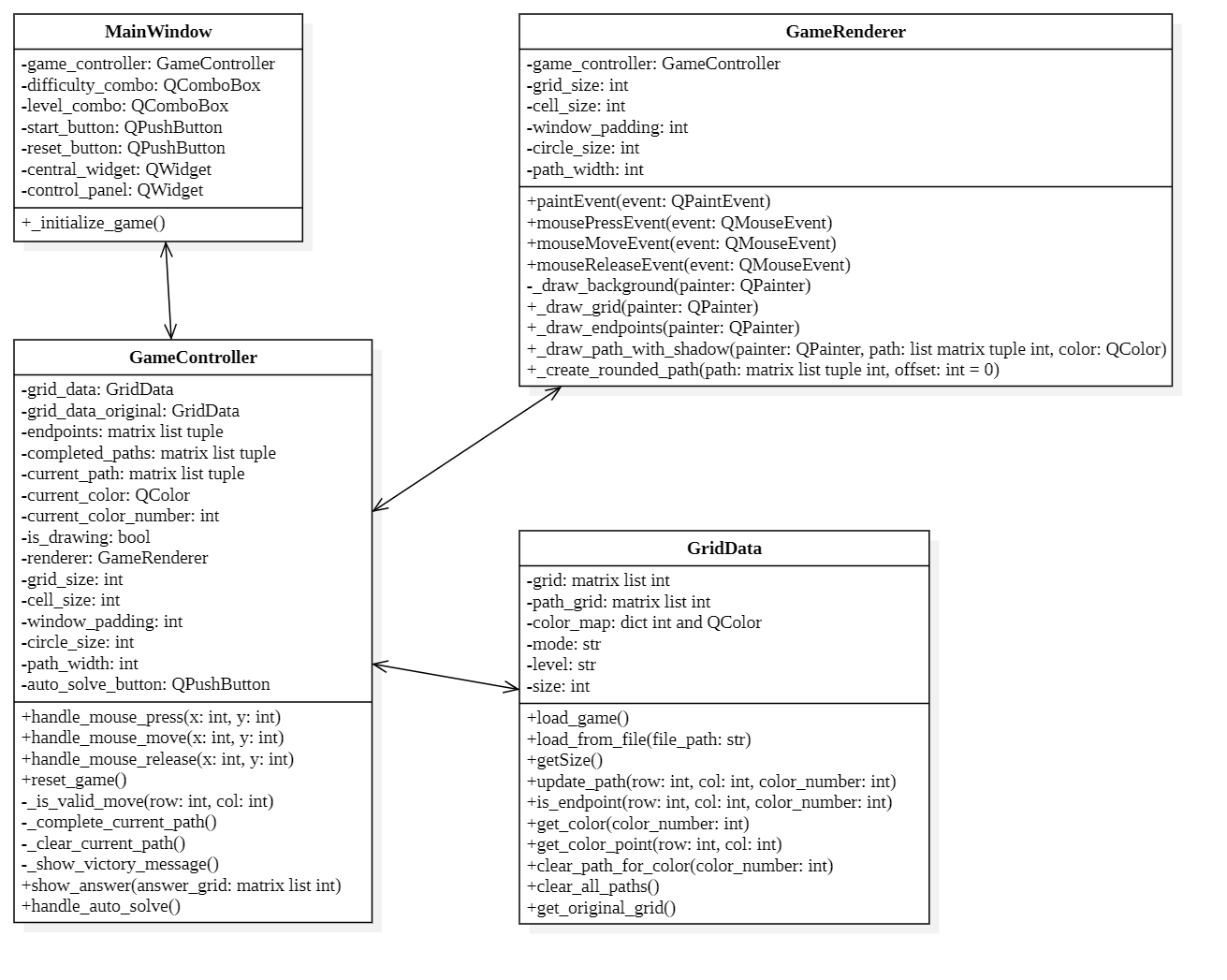
A screenshot of a computer

Description automatically generated

Hình . Cấu trúc file trong ứng dụng và mối quan hệ

### 3.1.2. Sơ đồ lớp (Class Diagram)

Sơ đồ lớp (Class Diagram) là một biểu đồ tĩnh trong UML (Unified Modeling Language - Ngôn ngữ Mô hình hóa Thống nhất) mô tả cấu trúc của hệ thống bằng cách thể hiện các lớp, thuộc tính, phương thức và mối quan hệ giữa chúng. Nó cung cấp một cái nhìn trực quan về các thành phần của hệ thống và cách chúng tương tác với nhau, giúp cho việc hiểu, phân tích, thiết kế và bảo trì hệ thống trở nên dễ dàng hơn.



Hình . Sơ đồ lớp của ứng dụng

Vì Slove vẫn là một hàm, không phải là một lớp, và hàm này chỉ được GameController gọi, nên phần 3.2. sẽ nói rõ hơn

### 3.1.3. Lớp MainWindow

Lớp này quản lý giao diện chính của ứng dụng. Nó chứa các thành phần giao diện người dùng như các nút bấm, hộp chọn độ khó, và chịu trách nhiệm khởi tạo và quản lý GameController.

**Thuộc tính:**

- game\_controller: GameController: Lưu trữ một thể hiện của lớp GameController để quản lý logic trò chơi.

- difficulty\_combo: QComboBox: Hộp chọn độ khó của trò chơi.

- level\_combo: QComboBox: Hộp chọn cấp độ của trò chơi.

- start\_button: QPushButton: Nút bắt đầu trò chơi.

- reset\_button: QPushButton: Nút chơi lại trò chơi.

- central\_widget: QWidget: Widget trung tâm chứa trò chơi.

- control\_panel: QWidget: Widget chứa các nút điều khiển.

**Phương thức:**

+ \_initialize\_game(): Khởi tạo trò chơi, bao gồm tạo thể hiện của GameController và thiết lập các thành phần giao diện.

### 3.1.4. GameController

Lớp này chịu trách nhiệm chính về logic của trò chơi. Nó xử lý các sự kiện người dùng, cập nhật trạng thái trò chơi, kiểm tra điều kiện thắng và tương tác với GridData và GameRenderer.

**Thuộc tính:**

- grid\_data: GridData: Lưu trữ dữ liệu của lưới trò chơi.

- grid\_data\_original: GridData: Lưu trữ dữ liệu gốc của lưới trò chơi.

- endpoints: list[tuple[int, int]]: Lưu trữ tọa độ các điểm đầu cuối.

- completed\_paths: list[list[tuple[int, int]]] : Lưu trữ các đường đi đã hoàn thành.

- current\_path: list[tuple[int, int]]: Lưu trữ đường đi hiện tại đang được vẽ.

- current\_color: QColor: Màu hiện tại đang được sử dụng để vẽ đường đi.

- current\_color\_number: int: Số thứ tự của màu hiện tại.

- is\_drawing: bool: Cờ cho biết người dùng có đang vẽ đường đi hay không.

- renderer: GameRenderer: Lưu trữ một thể hiện của lớp GameRenderer để vẽ giao diện trò chơi.

- grid\_size: int: Kích thước lưới.

- cell\_size: int: Kích thước ô.

- window\_padding: int: Khoảng cách từ cửa sổ đến lưới.

- circle\_size: int: Kích thước hình tròn ở điểm đầu cuối.

- path\_width: int: Độ rộng đường đi.

- auto\_solve\_button: QPushButton: Nút tự động giải.

**Phương thức:**

+ handle\_mouse\_press(x: int, y: int): Xử lý sự kiện nhấn chuột.

+ handle\_mouse\_move(x: int, y: int): Xử lý sự kiện di chuyển chuột.

+ handle\_mouse\_release(x: int, y: int): Xử lý sự kiện nhả chuột.

+ reset\_game(): Khởi tạo lại trò chơi.

- \_is\_valid\_move(row: int, col: int): Kiểm tra xem bước di chuyển có hợp lệ hay không.

- \_complete\_current\_path(): Hoàn thành đường đi hiện tại.

- \_clear\_current\_path(): Xóa đường đi hiện tại.

- \_show\_victory\_message(): Hiển thị thông báo chiến thắng.

+ show\_answer(answer\_grid: list[list[int]]): Hiển thị đáp án.

+ handle\_auto\_solve(): Xử lý sự kiện tự động giải.

### 3.1.5. GameRenderer

Lớp này chịu trách nhiệm vẽ giao diện đồ họa của trò chơi, bao gồm lưới, các điểm đầu cuối và các đường đi.

**Thuộc tính:**

- game\_controller: GameController: Lưu trữ một thể hiện của lớp GameController để lấy thông tin về trạng thái trò chơi.

- grid\_size: int: Kích thước lưới.

- cell\_size: int: Kích thước ô.

- window\_padding: int: Khoảng cách từ cửa sổ đến lưới.

- circle\_size: int: Kích thước hình tròn ở điểm đầu cuối.

- path\_width: int: Độ rộng đường đi.

**Phương thức:**

+ paintEvent(event: QPaintEvent): Vẽ lại giao diện trò chơi.

+ mousePressEvent(event: QMouseEvent): Xử lý sự kiện nhấn chuột.

+ mouseMoveEvent(event: QMouseEvent): Xử lý sự kiện di chuyển chuột.

+ mouseReleaseEvent(event: QMouseEvent): Xử lý sự kiện nhả chuột.

- \_draw\_background(painter: QPainter): Vẽ nền.

- \_draw\_grid(painter: QPainter): Vẽ lưới.

- \_draw\_endpoints(painter: QPainter): Vẽ các điểm đầu cuối.

- \_draw\_path\_with\_shadow(painter: QPainter, path: list[tuple[int, int]], color: QColor): Vẽ đường đi với hiệu ứng bóng đổ.

- \_create\_rounded\_path(path: list[tuple[int, int]], offset: int = 0): Tạo đường đi được bo tròn.

### 3.1.6. GridData

Lớp này lưu trữ và quản lý dữ liệu của lưới trò chơi, bao gồm vị trí các điểm đầu cuối và trạng thái của các đường đi.

**Thuộc tính:**

- grid: list[list[int]]: Ma trận biểu diễn trạng thái của lưới.

- path\_grid: list[list[int]]: Ma trận biểu diễn các đường đi trên lưới.

- color\_map: dict[int, QColor]: Bản đồ màu.

- mode: str: Chế độ chơi.

- level: str: Cấp độ chơi.

- size: int: Kích thước lưới.

**Phương thức:**

+ load\_game(): Tải dữ liệu trò chơi.

+ load\_from\_file(file\_path: str): Tải dữ liệu từ tệp.

+ getSize(): Lấy kích thước lưới.

+ update\_path(row: int, col: int, color\_number: int): Cập nhật đường đi tại vị trí (row, col).

+ is\_endpoint(row: int, col: int, color\_number: int): Kiểm tra xem vị trí (row, col) có phải là điểm cuối không.

+ get\_color(color\_number: int): Lấy màu theo số thứ tự.

+ get\_color\_point(row: int, col: int): Lấy số thứ tự màu tại vị trí (row, col).

+ clear\_path\_for\_color(color\_number: int): Xóa đường đi cho màu.

+ clear\_all\_paths(): Xóa tất cả đường đi.

+ get\_original\_grid(): Lấy dữ liệu lưới ban đầu.

## 3.2. Thuật toán tự động giải màn chơi

Thuật toán được sử dụng ở đây là sự kết hợp của Quy hoạch động trạng thái (Dynamic Programming) và một dạng Tìm kiếm theo chiều rộng (Breadth-First Search - BFS) được ngầm thực hiện trong quá trình xây dựng trạng thái.

### 3.2.1. Tư tưởng thuật toán

Ý tưởng cốt lõi của thuật toán là duyệt qua tất cả các trạng thái có thể có của bảng Flow Free một cách có hệ thống, từ đó tìm ra đường đi hợp lệ kết nối các cặp điểm màu. Thay vì duyệt theo kiểu backtracking (thử và sai), thuật toán sử dụng quy hoạch động để lưu trữ và tái sử dụng các trạng thái đã được tính toán, giúp tránh việc tính toán trùng lặp và tăng hiệu suất. Cụ thể như sau:

**Biểu diễn trạng thái:** Trạng thái của bảng được biểu diễn bằng một số nguyên lớn, trong đó mỗi ô trên bảng được mã hóa bằng một số bit (trong code là BITS = 4). Giá trị của mỗi ô có thể là:

* 0: Ô trống.
* 1: Điểm đầu của một đường.
* 2: Điểm cuối của một đường.
* 2: Ô đã được kết nối và thuộc một đường đi nào đó (giá trị này cho biết đường đi đó bắt đầu từ điểm màu nào).

**Xây dựng trạng thái:** Thuật toán duyệt qua bảng từ trái sang phải, từ trên xuống dưới. Tại mỗi ô, thuật toán xét tất cả các trạng thái có thể đến được ô đó từ các ô trước đó. Các trạng thái này được lưu trữ trong mảng . (với ) lưu trữ các trạng thái có thể đạt được sau khi xử lý đến ô .

**Quy hoạch động:** Việc lưu trữ các trạng thái trong h cho phép thuật toán tránh việc tính toán lại các trạng thái đã được xét. Khi xét một ô mới, thuật toán chỉ cần xem xét các trạng thái đã được lưu trữ cho các ô trước đó, thay vì phải duyệt lại toàn bộ không gian trạng thái.

**Tìm kiếm:** Quá trình xây dựng trạng thái thực chất là một dạng tìm kiếm. Bằng cách duyệt qua tất cả các trạng thái có thể, thuật toán ngầm thực hiện một tìm kiếm trên không gian trạng thái. Nếu tồn tại một trạng thái mà tất cả các đường đi đều được kết nối (tức là trạng thái kết thúc), thuật toán sẽ tìm thấy lời giải.

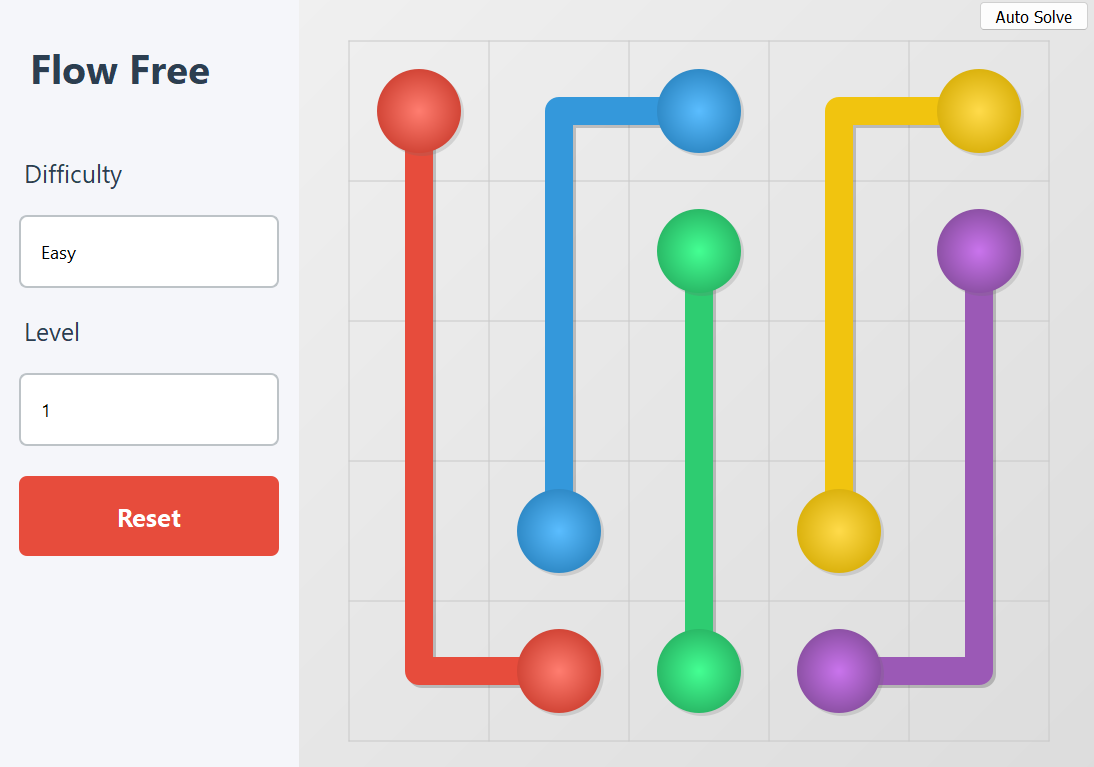
counterpart(): Hàm này rất quan trọng, nó giúp xác định điểm đầu/cuối tương ứng của một đường đi, từ đó đảm bảo các đường đi được kết nối chính xác theo yêu cầu của trò chơi.

### 3.2.2. Các bước giải chi tiết

Ta có ví dụ Input và Output như sau:

|  |  |
| --- | --- |
| **Đầu vào** | **Đầu ra cần đạt được** |
| 1 0 3 0 4  0 0 2 0 5  0 0 0 0 0  0 3 0 4 0  0 1 2 5 0 | 1 3 3 4 4  1 3 2 4 5  1 3 2 4 5  1 3 2 4 5  1 1 2 5 5 |

Khi hiển thị trong ứng dụng, ta sẽ được như hình:



Hình . Kết quả tự giải màn chơi khi hiện trong ứng dụng

Hàm Slove(original\_board, n, m) thực hiện các bước sau:

**Bước 1. Khởi tạo:**

* Khởi tạo các biến toàn cục board, color, conn, h, dir.
* Sao chép dữ liệu từ original\_board vào board (với offset 1).
* Khởi tạo trạng thái ban đầu: h[1][State(0)] = (State(0), Info()).

**Bước 2. Duyệt và xây dựng trạng thái (Quy hoạch động):**

* Duyệt qua từng ô của bảng theo thứ tự từ trái sang phải, từ trên xuống dưới.
* Tính .
* Duyệt qua tất cả các trạng thái cur\_state đã được lưu trữ trong .

Với mỗi cur\_state, tạo ra các trạng thái next\_state mới bằng cách xét các trường hợp sau:

**Trường hợp 1:**  (Ô trống):

* Nếu ô bên phải cũng trống, tạo trạng thái mới bằng cách đặt một đường đi ngang (1 ở j, 2 ở j+1).
* Nếu ô bên phải đã có đường đi , tạo trạng thái mới bằng cách tiếp tục đường đi đó hoặc kết thúc đường đi hiện tại ở ô j.

**Trường hợp 2:**  (Ô có số):

Tạo trạng thái mới bằng cách kết nối ô hiện tại với ô bên phải hoặc ô bên dưới, tuân theo quy tắc của trò chơi (các đường cùng màu phải nối với nhau). Sử dụng counterpart() để tìm điểm đầu/cuối tương ứng.

Các trường hợp đặc biệt (1-1, 2-2, 2-1, a>2 và b>2, a>2 và b<=2, a<=2 và b>2) được xử lý riêng để đảm bảo tính chính xác.

Lưu trạng thái mới next\_state vào cùng với thông tin kết nối Info (lưu hướng kết nối).

**Bước 3: Kiểm tra kết quả và truy vết ngược:**

* Kiểm tra xem trạng thái kết thúc State(0) có tồn tại trong h[n \* m + 1] hay không.
* Nếu có, thực hiện truy vết ngược từ h[n \* m + 1] về h[1] bằng cách sử dụng thông tin (\_cur, Info) đã lưu trong h. Thông tin Info được gán cho conn[x][y] để lưu lại hướng đi.

**Bước 4: Tô màu và cắt ma trận:**

* Sử dụng hàm bfs để tô màu các đường đi dựa trên thông tin kết nối conn.
* Gọi trim\_matrix(color) để loại bỏ các hàng/cột thừa.

**Mã giả:**

Hàm Solve(original\_board, n, m):

// Khởi tạo

board = sao chép original\_board (với offset)

h[1] = {trạng thái ban đầu (tất cả ô trống)}

// Duyệt bảng và xây dựng trạng thái

Cho i từ 1 đến n:

Cho j từ 1 đến m:

id = (i - 1) \* m + j

Cho mỗi cur\_state trong h[id]:

Nếu board[i][j] == 0: // Ô trống

Nếu ô bên phải (j+1) cũng trống:

next\_state = tạo trạng thái mới (đường ngang)

h[id+1].thêm(next\_state)

Nếu ô bên phải đã có đường đi:

next\_state = tạo trạng thái mới (tiếp tục/kết thúc đường đi)

h[id+1].thêm(next\_state)

Ngược lại (board[i][j] != 0): // Ô có số

next\_state = tạo trạng thái mới (kết nối theo quy tắc)

h[id+1].thêm(next\_state)

// Kiểm tra kết quả và truy vết

Nếu trạng thái kết thúc tồn tại trong h[n\*m+1]:

truy\_vết\_đường\_đi (từ h[n\*m+1] về h[1])

trả về đường đi

Ngược lại:

trả về không tìm thấy giải pháp

Hàm truy\_vết\_đường\_đi(h):

//Duyệt ngược từ trạng thái cuối về trạng thái đầu để tái tạo đường đi dựa trên thông tin đã lưu trong h

state = trạng thái kết thúc trong h[n\*m + 1]

Cho x từ n đến 1:

Cho y từ m đến 1:

id = (x - 1) \* m + y

conn[x][y] = thông tin kết nối từ h[id+1][state] // Lưu hướng đi

state = trạng thái trước đó được lưu trong h[id+1][state] // Lấy trạng thái trước đó

## 3.3. Thiết kế giao diện

Ứng dụng Flow Free được xây dựng bằng PyQt5, cung cấp một giao diện trực quan và thân thiện với người dùng. Phần này mô tả chi tiết thiết kế giao diện, đặc biệt là cách xử lý tương tác người dùng để tuân thủ luật chơi (không vẽ đường chéo) và các khía cạnh UX/UI quan trọng.

### 3.3.1. Các thành phần chính

**Vùng hiển thị trò chơi (Game Area):** Hiển thị lưới ô vuông và các đường đi. Sử dụng QGraphicsView và QGraphicsScene để vẽ đồ họa tùy chỉnh.

**Bảng điều khiển (Control Panel):**

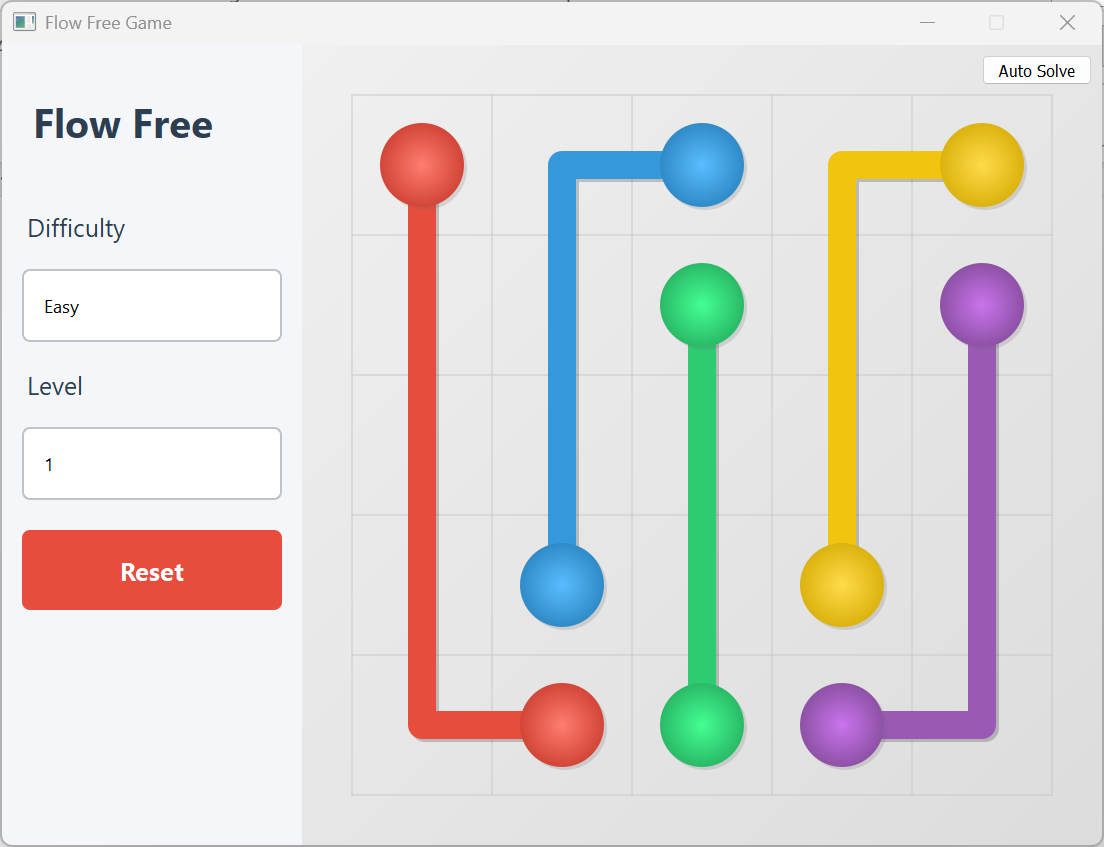
* Nút Bắt đầu (Start): Bắt đầu trò chơi mới.
* Nút Chơi lại (Reset): Khởi tạo lại màn chơi hiện tại.
* Hộp chọn độ khó (Difficulty): Chọn độ khó (Dễ, Trung bình, Khó).
* Hộp chọn màn chơi (Level): Chọn màn chơi cụ thể.
* Nút Tự động giải (Auto Solve): Kích hoạt thuật toán tự động giải.

### 3.3.2. Cấu trúc giao diện

Cấu trúc giao diện được xây dựng bằng các widget của PyQt5. QMainWindow là cửa sổ chính. QWidget (Central Widget) chứa QGraphicsView (vùng trò chơi) và QWidget (bảng điều khiển), được sắp xếp theo chiều dọc bằng QVBoxLayout. QGraphicsView hiển thị QGraphicsScene, nơi chứa các QGraphicsItem (ô vuông, đường đi). Các nút bấm (QPushButton) và hộp chọn (QComboBox) trong bảng điều khiển được sắp xếp bằng QHBoxLayout.

### 3.3.3. Thiết kế trực quan

Thiết kế trực quan tập trung vào sự rõ ràng và dễ nhìn. Nền lưới có màu xám nhạt hoặc trung tính, các ô vuông màu trắng hoặc sáng hơn nền. Đường đi có màu sắc tươi sáng, ngẫu nhiên và khác nhau cho mỗi cặp điểm đầu cuối, được quản lý bằng QColor và lưu trữ trong color\_map. Kích thước ô vuông được tính toán dựa trên kích thước QGraphicsView và kích thước lưới để vừa vặn. Độ dày đường đi có thể tùy chỉnh. Vùng trò chơi chiếm phần lớn cửa sổ, với bảng điều khiển được đặt ở dưới hoặc bên cạnh.



Hình . Giao diện ứng dụng

### 3.3.4. Tương tác người dùng

Tương tác người dùng dựa trên các sự kiện chuột. Khi người chơi nhấp chuột (mousePressEvent), ứng dụng kiểm tra xem ô được nhấp có phải điểm đầu cuối hay không. Nếu có, tọa độ điểm đầu cuối được lưu vào self.start\_point, màu được lưu vào self.current\_color, đường đi tạm thời self.current\_path được khởi tạo, và biến self.is\_drawing được đặt thành True.

Khi người chơi di chuyển chuột (mouseMoveEvent), ứng dụng chỉ xử lý nếu self.is\_drawing là True. Ứng dụng tính toán độ thay đổi tọa độ x và y so với self.start\_point và chỉ cho phép di chuyển nếu di chuyển theo phương ngang ( và ) hoặc phương dọc ( và ), ngăn chặn việc vẽ đường chéo. Tọa độ chuột được chuyển đổi sang tọa độ ô trong lưới. Ứng dụng kiểm tra tính hợp lệ của ô mới bằng hàm self.game\_controller.is\_valid\_move(x, y). Nếu hợp lệ, index của ô mới được thêm vào self.current\_path, self.start\_point được cập nhật thành tọa độ của ô mới (điều này rất quan trọng để đường đi không bị gãy khúc), và self.update() được gọi để vẽ lại đường đi.

Khi người chơi nhả chuột (mouseReleaseEvent), ứng dụng kiểm tra xem chuột có đang ở trên điểm cuối tương ứng với self.start\_point hay không. Nếu có, đường đi được hoàn thành bằng cách gọi self.game\_controller.complete\_path(), và ứng dụng kiểm tra xem tất cả các đường đi đã hoàn thành hay chưa. Nếu không, đường đi bị hủy bằng cách gọi self.game\_controller.clear\_path(). Cuối cùng, self.is\_drawing được đặt thành False, self.start\_point và self.current\_path được đặt lại.

Xử lý xóa đường đi: Ứng dụng hỗ trợ nhiều cách xóa đường đi: Xoá đường đi bằng nút “Reset” kéo ngược đường đi (xử lý trong mouseMoveEvent và mouseReleaseEvent với phản hồi trực quan), và xóa toàn bộ đường đi bằng cách nhấp vào đường đi (xử lý trong mousePressEvent).

Hàm is\_valid\_move(x, y) trong GameController kiểm tra xem ô có nằm trong lưới hay không và ô đã được đi qua bởi đường đi khác hay chưa (ngoại trừ điểm cuối của đường đi đó).

## 3.4. Xử lý tự động giải

### 3.4.1. Kích hoạt và thuật toán tìm kiếm

Khi người dùng nhấp vào nút "Auto Solve", phương thức handle\_auto\_solve() trong lớp GameController được gọi. Quá trình giải bài toán được thực hiện như sau:

* In ra thông báo "Solving..." để báo hiệu quá trình bắt đầu.
* Gọi hàm Solve() (được import từ module Slove) để giải bài toán Flow Free. Hàm Solve() nhận vào lưới ban đầu (self.grid\_data\_original.get\_original\_grid()) và kích thước lưới (self.grid\_size). Hàm này chịu trách nhiệm chính trong việc tìm ra lời giải cho bài toán.
* In ra thông báo "Answer found!" và in ra kết quả answer\_grid (để debug). answer\_grid là một ma trận biểu diễn lời giải, với mỗi ô chứa số màu của đường đi đi qua nó (0 nếu ô đó không thuộc đường đi nào).
* Kiểm tra xem answer\_grid có phải là None hay không. Nếu không phải None (tức là đã tìm thấy lời giải), gọi phương thức show\_answer() để hiển thị lời giải lên giao diện.

**3.4.2. Hiển thị lời giải (show\_answer()):**

Phương thức show\_answer() nhận vào ma trận answer\_grid (kết quả đã được giải bởi Solve()) và thực hiện các bước sau để hiển thị đường đi lên giao diện:

Xóa tất cả các đường đi hiện tại trên lưới bằng cách gọi self.completed\_paths.clear() và reset self.grid\_data.path\_grid.

Tìm điểm đầu và điểm cuối cho mỗi màu từ lưới ban đầu (self.grid\_data) và lưu vào dictionary endpoints. Việc này là cần thiết vì answer\_grid chỉ chứa thông tin về đường đi, không chứa thông tin về vị trí ban đầu của các điểm. Sử dụng setdefault để xử lý việc thêm các điểm vào dictionary một cách ngắn gọn.

Duyệt qua từng màu và cặp điểm đầu cuối trong endpoints:

* Kiểm tra xem mỗi màu có đúng 2 điểm đầu cuối hay không. Nếu không, bỏ qua màu đó (trường hợp lỗi dữ liệu).
* Gọi phương thức \_find\_path() để tái tạo đường đi giữa điểm đầu và điểm cuối dựa trên thông tin đã có trong answer\_grid.
* Nếu tìm thấy đường đi (path khác None):
* Cập nhật self.grid\_data.path\_grid với đường đi vừa tìm được.
* Thêm tuple (path, self.grid\_data.get\_color(color)) vào self.completed\_paths.
* Gọi self.renderer.update() một lần duy nhất sau khi xử lý tất cả các đường đi để vẽ lại toàn bộ giao diện, tối ưu hiệu suất.

### 3.4.3. Tái tạo đường đi bằng thuật toán tìm kiếm theo chiều rộng (\_find\_path())

Phương thức \_find\_path() không giải bài toán. Thay vào đó, nó sử dụng thuật toán BFS để tái tạo lại đường đi giữa hai điểm đã biết trước (điểm đầu và điểm cuối) dựa trên thông tin có trong answer\_grid. Điều này đảm bảo đường đi được hiển thị trên giao diện là chính xác theo lời giải đã được tìm thấy bởi Solve().

Thuật toán BFS trong \_find\_path() hoạt động như sau:

**Bước 1.** Khởi tạo visited và queue.

**Bước 2.** Thêm (start, []) vào queue và start vào visited.

**Bước 3.** Lặp cho đến khi queue rỗng:

* Lấy current và path từ queue.
* Nếu current là end, trả về path + [current].
* Duyệt 4 hướng.
* Kiểm tra:

Ô kế tiếp nằm trong lưới.

answer\_grid[nr][nc] == color: Ô kế tiếp có cùng màu theo lời giải trong answer\_grid.

Ô kế tiếp chưa được ghé thăm.

* Nếu tất cả điều kiện đúng, thêm vào queue và visited.

**Bước 4.** Nếu không tìm thấy đường đi, trả về None.

## 3.5. Hướng dẫn sử dụng

Để chạy ứng dụng, cần cài đặt trình biên dịch Python tại trang sau:

<https://www.python.org/downloads/>

Sau khi cài đặt xong, cần cài đặt các thư viện. Nhập vào máy các lệnh sau để thực hiện cài đặt:

Lệnh cập nhật trình cài đặt gói:

pip install --upgrade pip

Lệnh cài đặt PyQT5:

pip install PyQt5

Sau khi giải nén tệp tin, ta được thư mục như hình sau:

A screenshot of a computer

Description automatically generated

Hình . Thư mục ứng dụng sau khi giải nén

Để chạy ứng dụng, ta chỉ cần mở file main.pyw là được, hoặc chuột phải vào file main.pyw, chọn Open With Python.

# Chương IV. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

## 4.1. Kết quả đạt được

A screenshot of a computer

Description automatically generated

Hình . Giao diện trò chơi

**Giao diện người dùng:** Ứng dụng cung cấp giao diện trực quan, cho phép người chơi lựa chọn độ khó và cấp độ. Giao diện được thiết kế rõ ràng, dễ sử dụng với các widget như QLabel, QPushButton, và QComboBox, cùng với khả năng tùy chỉnh giao diện bằng CSS.

Tương tác chuột: Ứng dụng xử lý chính xác các sự kiện chuột (mousePressEvent, mouseMoveEvent, mouseReleaseEvent), cho phép người chơi vẽ đường đi giữa các điểm đầu cuối một cách tự nhiên và chính xác, đồng thời ngăn chặn việc vẽ đường chéo.

**Logic trò chơi:** Logic cốt lõi của trò chơi được triển khai, bao gồm kiểm tra tính hợp lệ của nước đi (is\_valid\_move), hoàn thành đường đi (complete\_path), xóa đường đi (clear\_path), và kiểm tra chiến thắng.

**Tự động giải:** Ứng dụng tích hợp tính năng tự động giải bằng cách sử dụng thuật toán được cung cấp từ module Solve(). Việc hiển thị lời giải được thực hiện bằng thuật toán BFS (\_find\_path()) để tái tạo đường đi trên giao diện dựa trên kết quả đã được giải.

Khả năng mở rộng: Cấu trúc code được tổ chức tốt, với các class như MainWindow, GameController, GridData, và GameRenderer, tạo điều kiện cho việc mở rộng và bảo trì ứng dụng trong tương lai.

## 4.2. Hạn chế của ứng dụng

Ứng dụng hiện tại còn tồn tại một số vấn đề cần được cải thiện. Về mặt thuật toán, hiệu suất của hàm Solve() chưa được tối ưu, đặc biệt là khi xử lý các bảng có kích thước lớn. Điều này có thể dẫn đến thời gian xử lý kéo dài và ảnh hưởng đến trải nghiệm người dùng. Về mặt đồ họa, giao diện của trò chơi còn khá đơn giản và chưa thực sự bắt mắt, cần được thiết kế lại để thu hút người chơi hơn. Về mặt tính năng, ứng dụng hiện thiếu một số tính năng quan trọng như lưu/tải game để người chơi có thể tiếp tục chơi sau đó, undo/redo để người chơi có thể sửa lỗi và thử nghiệm, hint để hỗ trợ người chơi khi gặp khó khăn, và các chế độ chơi khác nhau để tăng tính đa dạng và thử thách. Cuối cùng, một vấn đề quan trọng khác là ứng dụng chưa được kiểm tra lỗi một cách toàn diện, điều này có thể dẫn đến các lỗi tiềm ẩn ảnh hưởng đến tính ổn định và trải nghiệm người dùng.

## 4.3. Hướng phát triển

Dựa trên những hạn chế trên, các hướng phát triển tiềm năng cho ứng dụng bao gồm:

**Tối ưu thuật toán giải:** Nghiên cứu và áp dụng các thuật toán giải bài toán Flow Free hiệu quả hơn, chẳng hạn như thuật toán backtracking được tối ưu, hoặc các heuristic search algorithms như A\*. Điều này sẽ cải thiện đáng kể hiệu suất của tính năng tự động giải, đặc biệt với các bảng lớn.

**Cải thiện đồ họa:** Nâng cấp đồ họa của trò chơi bằng cách sử dụng hình ảnh đẹp mắt hơn, hiệu ứng chuyển động mượt mà, hoặc hỗ trợ các theme khác nhau. Sử dụng các thư viện đồ họa mạnh mẽ hơn nếu cần.

Để nâng cao trải nghiệm người dùng và chất lượng trò chơi, cần bổ sung các tính năng sau: lưu/tải game để bảo toàn tiến trình; undo/redo để sửa sai và thử nghiệm; gợi ý (hint) khi gặp khó khăn; các chế độ chơi đa dạng hơn (ví dụ: tính giờ, thử thách); hiệu ứng âm thanh và nhạc nền để tăng tính hấp dẫn. Về mặt kỹ thuật, cần kiểm tra và sửa lỗi kỹ lưỡng, viết unit test cho từng thành phần, hỗ trợ đa nền tảng (ví dụ: web, di động), và tối ưu hiệu suất, đặc biệt là hiệu suất vẽ và xử lý sự kiện chuột (ví dụ: hạn chế gọi update() không cần thiết).

TÀI LIỆU THAM KHẢO

[1] BFS (Breadth-first search). (2025). VNOI Wiki. <https://wiki.vnoi.info/algo/graph-theory/breadth-first-search.md>

[2] GeeksforGeeks. (2024, February 9). *Dynamic Programming or DP*. GeeksforGeeks. <https://www.geeksforgeeks.org/dynamic-programming/>

‌[3] GeeksforGeeks. (2017, June 23). *Bitmasking and Dynamic Programming | Travelling Salesman Problem*. GeeksforGeeks. <https://www.geeksforgeeks.org/bitmasking-dynamic-programming-set-2-tsp/>

‌[4] *Dynamic Programming and Bit Masking Tutorials & Notes | Algorithms | HackerEarth*. (2016). HackerEarth. <https://www.hackerearth.com/practice/algorithms/dynamic-programming/bit-masking/tutorial/>

[5] Phạm Thị Kim Ngoan (2023). *Bài giảng Lập trình hướng đối tượng.* Bộ môn Công nghệ phần mềm, Đại học Nha Trang, Khoa Công nghệ thông tin.

[6] Nguyễn Đức Thuần, *Kỹ thuật lập trình*, NXB KHKT, 2022.