BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC SƯ PHẠM THÀNH PHỐ HỒ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

MÔN LÝ THUYẾT ĐỒ THỊ VÀ ỨNG DỤNG

---ꕥ---

TIỂU LUẬN

Triển khai thuật toán Dijkstra trong việc giải bài toán tìm lối thoát mê cung

Học phần: 2511COMP170102 – Lý thuyết đồ thị và ứng dụng

Thành phố Hồ Chí Minh, ngày 08 tháng 12 năm 2025

BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC SƯ PHẠM THÀNH PHỐ HỒ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

MÔN LÝ THUYẾT ĐỒ THỊ VÀ ỨNG DỤNG

---ꕥ---

TIỂU LUẬN

Triển khai thuật toán Dijkstra trong việc giải bài toán tìm lối thoát mê cung

Học phần: 2511COMP170102 – Lý thuyết đồ thị và ứng dụng

Nhóm sinh viên thực hiện : Nhóm 5 anh em siêu nhân

Phạm Ngọc Mỹ Huyền	50.01.103.028
Nguyễn Bình Phương Mi	50.01.103.039
Thạch Võ Diễm Ngọc	50.01.103.050
Phùng Thị Mỹ Quyên	50.01.103.063
Nguyễn Nhất Xuân	50.01.103.091

Giảng viên hướng dẫn: TS. Nguyễn Viết Hưng

Thành phố Hồ Chí Minh, ngày 08 tháng 12 năm 2025

MỤC LỤC

NHIỆM VỤ THÀNH VIÊN NHÓM

Họ tên	Công việc được giao	Mức độ hoàn thành
Phạm Ngọc Mỹ Huyền	Xử lý mê cung: lập trình chức năng xử lý dữ liệu mê cung, viết báo cáo và kiểm thử; thuyết trình.	100%
Nguyễn Bình Phương Mi	Xây dựng giao diện (GUI): thiết kế giao diện, viết báo cáo và kiểm thử; làm file HDSD.	100%
Thạch Võ Diễm Ngọc	Thuật toán Dijkstra: triển khai thuật toán, viết báo cáo và kiểm thử; thuyết trình.	100%
Phùng Thị Mỹ Quyên	Thiết kế slide trình bày, xây dựng hàm main, viết báo cáo và kiểm thử.	100%
Nguyễn Nhất Xuân	Kiểm thử chức năng và xử lý lỗi, phát triển GUI, viết báo cáo; thuyết trình.	100%

DANH MỤC CÁC KÝ HIỆU VÀ CHỮ VIẾT TẮT

Chữ viết tắt	Nguyên mẫu	Diễn giải
KÝ HIỆU TOÁN HỌC/ĐỒ THỊ		
G(V, E)	Graph	Ký hiệu chung cho Đồ thị, mô hình hóa mê cung.
V	Vertices (Tập hợp các đỉnh)	Đại diện cho các ô/vị trí có thể di chuyển trong mê cung.
E	Edges (Tập hợp các cạnh)	Đại diện cho các đường đi kết nối giữa các ô liền kề.
w(u, v)	Weight (Trọng số)	Chi phí để di chuyển từ đỉnh u đến đỉnh v.
O(...)	Big O Notation	Ký hiệu dùng để chỉ độ phức tạp thời gian hoặc bộ nhớ của thuật toán.
d[v]	Distance (Khoảng cách)	Khoảng cách ngắn nhất được tính từ đỉnh nguồn đến đỉnh v.
CHỮ VIẾT TẮT THUẬT TOÁN/CÔNG NGHỆ		
Dijkstra	Thuật toán Dijkstra	Thuật toán cốt lõi được triển khai để tìm đường đi ngắn nhất.
A*	Thuật toán A-Star	Thuật toán tìm đường đi tối ưu (được đề cập trong phần mở rộng).
PQ	Priority Queue	Hàng đợi Ưu tiên, cấu trúc dữ liệu được sử dụng để tối ưu hiệu suất Dijkstra.
CNTT	Công nghệ Thông tin	
I/O	Input/Output	Quá trình nhập/xuất dữ liệu (thường là đọc file và in kết quả).

GUI	Graphical User Interface	Giao diện người dùng đồ họa (Phần hiển thị mê cung bằng Python).

DANH MỤC CÁC BẢNG BIỂU

DANH MỤC CÁC HÌNH VẼ

MỞ ĐẦU

1. Lý do chọn đề tài

Lý thuyết đồ thị là một lĩnh vực nền tảng của khoa học máy tính hiện đại, bắt nguồn từ công trình kinh điển của Leonhard Euler vào thế kỷ XVIII và ngày nay được xem là công cụ quan trọng trong việc mô hình hóa các hệ thống phức tạp trong vận tải, mạng máy tính, trí tuệ nhân tạo và tối ưu hóa. Theo Bondy và Murty (Graph Theory with Applications, 1976), các bài toán về đường đi ngắn nhất giữ vai trò cốt lõi trong nhóm bài toán tối ưu trên đồ thị, vì chúng xuất hiện trong hầu hết các ứng dụng thực tế có liên quan đến định tuyến, dẫn đường và xử lý không gian trạng thái.

Trong số đó, thuật toán **Dijkstra (1959)** là một trong những thuật toán được sử dụng phổ biến và hiệu quả nhất để tìm đường đi ngắn nhất trên đồ thị có trọng số. Khi kết hợp với mô hình mê cung (maze) - một dạng đồ thị hình lưới. Thuật toán Dijkstra cho phép xác định đường đi tối ưu từ điểm xuất phát đến điểm đích. Việc triển khai Dijkstra trên mê cung không chỉ giúp củng cố kiến thức lý thuyết, mà còn hỗ trợ người học hình dung trực quan mối liên hệ giữa cấu trúc dữ liệu và thuật toán. Chính vì vậy, nhóm chúng em lựa chọn đề tài **"Triển khai thuật toán Dijkstra trong việc giải bài toán tìm lối thoát mê cung"** nhằm vận dụng kiến thức đã học vào một bài toán thực tế cụ thể.

2. Mục tiêu và nhiệm vụ nghiên cứu

Mục tiêu chính của đề tài là xây dựng chương trình tìm đường đi ngắn nhất trong mê cung sử dụng thuật toán Dijkstra, cho phép người dùng nhập mê cung từ file và hiển thị kết quả đường đi, bao gồm cả các trường hợp có hoặc không tồn tại lối thoát.

Các nhiệm vụ cụ thể gồm:

- Nghiên cứu nguyên lý hoạt động và cài đặt thuật toán Dijkstra.
- Thiết kế và xây dựng các module chức năng (đọc mê cung, xử lý dữ liệu, tìm đường, hiển thị kết quả).

– Tiến hành kiểm thử tích hợp và xây dựng các bộ test case để đánh giá tính chính xác và hiệu quả của chương trình.

3. Đối tượng và phạm vi nghiên cứu

3.1. Đối tượng nghiên cứu

Thuật toán Dijkstra và mô hình mê cung dạng lưới (grid-based graph), với mỗi ô là một đỉnh và các bước di chuyển là cạnh.

3.2. Phạm vi nghiên cứu

– Chỉ xét đồ thị có trọng số không âm (điều kiện đúng cho Dijkstra theo CLRS).
– Mê cung là ma trận ký tự đọc từ file *.txt*.
– Không xét các thuật toán khác như A*, BFS, Ford-Bellman.

4. Phương pháp nghiên cứu

Đề tài sử dụng các phương pháp:

– **Phân tích lý thuyết:** dựa trên các tài liệu kinh điển về thuật toán đồ thị như CLRS, Sedgewick.
– **Mô hình hóa:** chuyển đổi mê cung sang đồ thị lưới có trọng số.
– **Thực nghiệm:** triển khai C++, chạy thử nghiệm trên nhiều file input.
– **Kiểm thử và đánh giá:** tiến hành kiểm thử với các bộ dữ liệu khác nhau để đánh giá tính đúng đắn.

5. Kết cấu của đề tài

Nội dung bài tiểu luận được xây dựng gồm các phần sau:

– **Chương 1:** Tổng quan.
– **Chương 2:** Cơ sở lý thuyết.
– **Chương 3:** Thiết kế và cài đặt chương trình.
– **Chương 4:** Demo và kết quả thực nghiệm.
– **Chương 5:** Phân tích và thảo luận.
– **Chương 6:** Kết luận.

CHƯƠNG 1. TỔNG QUAN

1.1. Giới thiệu đề tài

Mê cung là một trong những mô hình tiêu biểu của các bài toán tìm đường trong không gian bị giới hạn bởi các ràng buộc về di chuyển. Trong bài toán này, mê cung được xem như một một đồ thị có trọng số. Ta ký hiệu đồ thị G=(V,E), trong đó:

- V là tập các **đỉnh**, mỗi đỉnh tương ứng với một ô có thể đứng.
- E là tập các **cạnh**, mỗi cạnh biểu diễn một lối đi hợp lệ giữa hai ô kề nhau.
- Mỗi cạnh e ∈ E được gán **trọng số**, tượng trưng cho chi phí di chuyển giữa hai ô tương ứng.

Với mô hình này, bài toán tìm đường trong mê cung trở thành bài toán tìm đường đi ngắn nhất trên đồ thị. Khi đó, thuật toán Dijkstra có thể được triển khai để tìm đường ra bằng cách mở rộng lần lượt từ đỉnh xuất phát sang các đỉnh có chi phí tạm thời nhỏ nhất.

Bài toán tìm đường đi ngắn nhất trong mê cung được đặt ra như sau:

Cho một mê cung đã được mô hình hóa thành đồ thị G=(V, E), với một đỉnh xuất phát S tương ứng vị trí ban đầu và một đỉnh kết thúc E tương ứng với lối thoát. Yêu cầu là tìm một đường đi hợp lệ từ S đến E sao cho tổng chi phí di chuyển là nhỏ nhất. Trong trường hợp không tồn tại bất kỳ đường đi nào từ S đến E, bài toán xem như không có nghiệm.

1.2. Mục tiêu và ý nghĩa của đồ án

Mục tiêu của đồ án là xây dựng một mô hình mê cung dưới dạng đồ thị có trọng số và áp dụng thuật toán Dijkstra để tìm đường đi ngắn nhất từ vị trí xuất phát đến lối thoát. Đồ án hướng đến việc trình bày rõ ràng quá trình mô hình hóa, phân tích và triển khai thuật toán tìm đường trong mê cung, đồng thời tạo ra một chương trình mô phỏng trực quan cho phép quan sát kết quả tìm đường.

Đồ án giúp củng cố và mở rộng kiến thức về lý thuyết đồ thị, mô hình hóa đường đi và thuật toán Dijkstra. Việc chuyển một mê cung thực tế thành một bài toán trên đồ thị cũng giúp chúng em rèn luyện tư duy trừu tượng và khả năng áp dụng lý thuyết vào thực tế. Ngoài ra, bài toán tìm đường cũng được ứng dụng nhiều trong công nghệ, như định tuyến mạng, robot tự hành, lập kế hoạch đường đi trong trí tuệ nhân tạo và trò chơi điện tử. Về thực tiễn, đồ án này góp phần làm rõ cách một thuật toán cổ điển có thể được sử dụng để giải quyết một vấn đề.

1.3. Giới thiệu đồ án mê cung và mô hình hóa đồ thị

Trong đồ án này, mê cung được xem như một không gian gồm các vị trí có thể đứng và di chuyển giữa chúng. Mỗi vị trí đều bị giới hạn bởi các tường, tạo nên cấu trúc phức tạp đặc trưng của mê cung. Để thuật toán có thể xử lý được bài toán tìm đường, mê cung cần được biểu diễn dưới dạng một mô hình có cấu trúc rõ ràng thay cho hình ảnh trực quan ban đầu.

Để áp dụng các thuật toán tìm đường, mê cung cần được mô hình hoá dưới dạng đồ thị. Trong mô hình này, mỗi ô có thể di chuyển trong mê cung được xem như một **đỉnh** của đồ thị, được ký hiệu theo chỉ số và tọa độ. những lối đi hợp lệ giữa hai ô kề nhau được biểu diễn dưới dạng **cạnh**, thể hiện khả năng di chuyển trực tiếp giữa hai đỉnh. Mỗi cạnh được gán một **trọng số**, là đại diện cho chi phí di chuyển giữa hai ô.

Cách mô hình hóa này cho phép chuyển bài toán tìm lối thoát mê cung thành bài toán tìm đường đi ngắn nhất trên một đồ thị có trọng số dương. Nhờ đó, thuật toán Dijkstra có thể được áp dụng một cách trực tiếp và hiệu quả để xác định đường đi có tổng chi phí nhỏ nhất từ vị trí xuất phát đến lối thoát.

1.4. Phạm vi thực hiện và yêu cầu kĩ thuật

Phạm vi thực hiện:

Để đảm bảo mục tiêu nghiên cứu, đồ án tập trung vào mô hình mê cung 2D dưới dạng lưới ô vuông, mỗi ô có hai trạng thái là tường hoặc đường đi. Các bước di chuyển chỉ theo bốn hướng cơ bản là lên - xuống - trái - phải, không mở rộng di chuyển chéo và môi trường 3D. Chỉ có thuật toán Dijkstra được áp dụng để tìm đường đi ngắn nhất từ vị trí xuất phát đến lối thoát; đồng thời phạm vi triển khai phần mềm giới hạn ở **ngôn ngữ lập trình C++** và **Python**, chạy trên máy tính cá nhân, với giao diện mô phỏng cơ bản.

Yêu cầu kỹ thuật:

Chương trình yêu cầu mô hình hóa mê cung thành đồ thị trọng số, với mỗi ô là đỉnh, mỗi lối đi hợp lệ là cạnh, và trọng số thể hiện chi phí di chuyển. Đồng thời, thuật toán Dijkstra phải được cài đặt đúng, đảm bảo xác định được đường đi tối ưu nếu tồn tại. Kết quả cuối cùng phải hiển thị trực quan đường đi trong mê cung, thể hiện rõ vị trí xuất phát, lối thoát và các bước di chuyển.

CHƯƠNG 2. CƠ SỞ LÝ THUYẾT

2.1. Khái niệm về thuật toán tìm đường ngắn nhất

Trong lĩnh vực lý thuyết đồ thị, bài toán tìm đường đi ngắn nhất đóng vai trò nền tảng cho việc xác định đường đi tối ưu và được ứng dụng nhiều trong thực tế như thiết kế mạng lưới giao thông, tìm lối thoát trong mê cung hay mô phỏng lộ trình tối ưu. Hiểu rõ các thuật toán giúp chúng ta đánh giá được ưu – nhược điểm của từng phương pháp, từ đó lựa chọn giải pháp phù hợp nhất cho mô hình bài toán đang nghiên cứu. Thuật toán tìm đường đi ngắn nhất là thuật toán xác định đường đi giữa hai đỉnh sao cho tổng trọng số các cạnh là nhỏ nhất. Tùy thuộc vào đặc trưng của đồ thị như có hướng hay vô hướng, có trọng số hay không trọng số, trọng số có âm hay không, người ta sẽ áp dụng những thuật toán khác nhau. Một số thuật toán phổ biến để tìm đường đi ngắn nhất trong đồ thị là: Dijkstra, Ford-Bellman, BFS, A*,... Mỗi thuật toán áp dụng một phương pháp tìm kiếm riêng, được thiết kế để phát huy hiệu quả tối đa và mỗi cách phù hợp với từng loại dữ liệu hay mô hình bài toán cụ thể. Do đó, việc nắm vững bản chất của từng thuật toán là điều cần thiết trước khi đi sâu vào lựa chọn thuật toán tối ưu cho bài toán tìm lối thoát trong mê cung mà nhóm chúng tôi sẽ thực hiện.

2.2. Thuật toán Dijkstra – Nguyên lý hoạt động

Trong số các thuật toán tìm đường đi ngắn nhất được nêu trên, Dijkstra là một trong những thuật toán tiêu biểu và được sử dụng rộng rãi nhờ tính chính xác cao, khả năng xử lý hiệu quả để tìm đường đi ngắn nhất từ một đỉnh nguồn đến các đỉnh còn lại trong đồ thị có trọng số dương. Thuật toán đảm bảo tìm được đường đi có trọng số nhỏ nhất và được ứng dụng đa dạng trong thực tế như: Tìm lối thoát mê cung, tìm chi phí nhỏ nhất trong mạng lưới vận tải hoặc giao thông, tối ưu hóa lộ trình di chuyển trong bản đồ,... Những ứng dụng này không chỉ đòi hỏi độ chính xác tuyệt đối mà còn cần khả năng xử lý nhanh chóng và hiệu quả, đặc biệt khi đồ thị có kích thước lớn.Vì lý do này, Dijkstra được xem là lựa chọn tin cậy và thường được ưu tiên sử dụng trong các tình huống thực

tế, nơi việc xác định giải pháp tối ưu là cần thiết.

Về phương pháp, thuật toán Dijkstra tiến hành xác định các đỉnh theo thứ tự khoảng cách tăng dần từ đỉnh bắt đầu (u0), nghĩa là lần lượt chọn những đỉnh có khoảng cách ngắn nhất chưa được cố định, rồi cập nhật các khoảng cách tạm thời đến các đỉnh kề, đảm bảo rằng mỗi bước tiến đều dẫn đến kết quả tối ưu. Nói cách khác, thuật toán sẽ xác định tuần tự các đỉnh có khoảng cách đến u0 từ nhỏ đến lớn. Nhờ cách tiếp cận này, thuật toán không chỉ tìm được đường đi ngắn nhất đến một đích cụ thể mà còn cung cấp toàn bộ thông tin về khoảng cách tối ưu từ đỉnh bắt đầu đến tất cả các đỉnh khác trong đồ thị.

Thuật toán Dijkstra:

Bước 1. i:=0, S:=V\{u0}, L(u0):=0, L(v):= với mọi v S và đánh dấu đỉnh v bởi (,-). Nếu n=1 thì xuất d(u0,u0)=0=L(u0).

Bước 2. Với mọi v S và kề với ui(nếu đồ thị có hướng thì v là đỉnh sau của ui), đặt L(v):=min{L(v),L(ui)+w(ui,v)}. Xác định k =minL(v) ,vS.

Nếu k=L(vj) thì xuất d(u0,vj)=k và đánh dấu vj bởi (L(vj);ui).

ui+1:=vj S:=S\{ui+1}

Bước 3. i:=i+1

Nếu i = n-1 thì kết thúc

Nếu không thì quay lại Bước 2

Nguyên lý hoạt động:

Bước 1. Trước tiên đỉnh có khoảng cách nhỏ nhất đến u0 là u0.

Bước 2. Trong V\{u0} tìm đỉnh có khoảng cách đến u0 nhỏ nhất (đỉnh này phải là một trong các đỉnh kề với u0) giả sử đó là u1.

Bước 3. Trong V\{u0,u1} tìm đỉnh có khoảng cách đến u0 nhỏ nhất (đỉnh này phải là một trong các đỉnh kề với u0 hoặc u1) giả sử đó là u2.

Bước 4. Tiếp tục như trên cho đến bao giờ tìm được khoảng cách từ u0 đến mọi đỉnh.

Nếu G có n đỉnh thì:

0 = d(u0,u0) < d(u0,u1) d(u0,u2) ... d(u0,un-1)

2.3. Phân tích độ phức tạp và so sánh với BFS, A*

Bảng so sánh:

Thuật toán	Độ phức tạp	Ưu điểm	Nhược điểm
Dijkstra (sử dụng priority queue)	O(E log V)	Tìm đường ngắn nhất chính xác trong đồ thị có trọng số dương.	Không sử dụng được cho đồ thị có trọng số âm
Dijkstra (danh sách kề + hàng đợi ưu tiên)	O((V + E) * log(V))	Tìm đường ngắn nhất hiệu quả trên đồ thị thưa có trọng số dương.	Không sử dụng được cho đồ thị có trọng số âm.
BFS	O(V+E)	– Luôn tìm đường đi ngắn nhất. – Hiệu quả với đồ thị có kích thước nhỏ và trung bình.	Yêu cầu bộ nhớ lớn khi giải các mê cung lớn.
A*	O(b^d)	– Sử dụng hàm – heuristic, tìm đường nhanh chóng. – Tìm được đường đi tối ưu trong phần lớn trường hợp.	– Phụ thuộc vào chất lượng của hàm heuristic. – Yêu cầu bộ nhớ lớn.

Bảng 1. Bảng so sánh các thuật toán

Chú thích:

O: Độ phức tạp

E: Tập cạnh E

V: Tập đỉnh V

b: Độ rộng nhánh trong thuật toán A*

d: Độ sâu của nút mục tiêu (số bước cần để đến đích)

Trong mô hình mê cung dạng lưới của đề tài, nơi các ô được biểu diễn bằng giá trị 0 (tường) và 1 (ô có thể di chuyển), việc lựa chọn thuật toán tìm đường phù hợp giữ vai trò quan trọng nhằm tối ưu hóa hiệu quả xử lý. Sau khi phân tích các thuật toán phổ biến, nhóm chúng em quyết định sử dụng Dijkstra, bởi thuật toán này phù hợp với đặc trưng mê cung có trọng số không đổi và hoạt động hiệu quả khi kết hợp với hàng đợi ưu tiên (priority queue), giúp nhanh chóng lựa chọn ô có khoảng cách nhỏ nhất tại mỗi bước mở rộng.

Các thuật toán khác mặc dù có ưu điểm riêng nhưng lại chưa đáp ứng tối ưu cho bối cảnh bài toán. Cụ thể, BFS có thể tìm đường ngắn nhất trên đồ thị không trọng số, song lại thiếu cơ chế ưu tiên và khó tối ưu khi áp dụng trên mê cung lớn do nhu cầu bộ nhớ cao. A* tuy định hướng đích hiệu quả, nhưng yêu cầu xây dựng hàm heuristic phù hợp - một bước không cần thiết đối với mê cung có trọng số đồng nhất và không có đặc trưng hình học phức tạp. Thuật toán Ford–Bellman cũng không phải lựa chọn phù hợp do độ phức tạp thời gian lớn, vốn chỉ thích hợp với đồ thị có trọng số âm hoặc bài toán tìm đường giữa mọi cặp đỉnh.

Vì những lý do trên, Dijkstra kết hợp priority queue trở thành lựa chọn hợp lý và tối ưu nhất cho bài toán, nhờ sự cân bằng giữa tính chính xác tuyệt đối, hiệu quả xử lý và tính đơn giản trong triển khai, đáp ứng đầy đủ yêu cầu của mô hình mê cung mà nhóm chúng em thực hiện.

2.4. Ánh xạ mê cung thành đồ thị

Đồ thị là một trong những mô hình toán học quan trọng trong lĩnh vực khoa học máy tính, được biểu diễn dưới dạng $G=(V,E)$, trong đó V là tập hợp các đỉnh và E là tập hợp các cạnh nối giữa các đỉnh. Cấu trúc này cho phép mô tả trực quan mối quan hệ liên kết giữa các đối tượng, cũng như hỗ trợ việc áp dụng các thuật toán tìm kiếm và tối ưu.

Trong bài toán tìm lối thoát khỏi mê cung, các thuật toán tìm đường như Dijkstra chỉ có thể hoạt động trên cấu trúc đồ thị. Do đó, để thuật toán có thể vận hành đúng, mê cung phải được mô hình hóa thành một đồ thị tương ứng.

Thành phần của mê cung:

- **Ô trống:** là những vị trí có thể di chuyển tới. Các ô này được chuyển thành các đỉnh trong đồ thị.
- **Tường:** là những ô không thể đi qua; ngăn cản việc hình thành cạnh giữa các ô lân cận.
- **Điểm bắt đầu và điểm kết thúc:** Là hai ô đặc biệt, nhưng về bản chất vẫn được xem là các ô trống khi ánh xạ sang đồ thị, do đó chúng cũng trở thành các đỉnh.

Ánh xạ các thành phần cơ bản của mê cung thành đồ thị:

- **Đỉnh:** Mỗi ô trống sẽ tương ứng với một đỉnh thuộc tập V.
- **Cạnh:** Một cạnh $(u,v) \in E$ được tạo ra nếu có thể di chuyển trực tiếp giữa hai ô u và v, tức là chúng là hai ô kề nhau theo bốn hướng cơ bản: lên, xuống, trái, phải.

Nhờ ánh xạ này, cấu trúc mê cung được chuyển đổi mạch lạc thành cấu trúc đồ thị – nơi thuật toán Dijkstra có thể được áp dụng một cách hiệu quả.

Để triển khai thuật toán tìm đường, cần mô tả một cách chính xác hai thành phần chính của đồ thị: tập đỉnh V và tập cạnh E.

Về tập đỉnh *V*, mỗi đỉnh trong đồ thị tương ứng với một ô trống và được biểu diễn bằng tọa độ (x,y). Một ô chỉ được đưa vào tập đỉnh nếu thỏa điều kiện *không phải là tường*.

$$maze[x][y]!=wall$$

Như vậy, tập V bao gồm toàn bộ các ô trống, bao gồm cả ô bắt đầu và ô kết thúc, đảm bảo rằng đồ thị phản ánh đầy đủ các vị trí có thể di chuyển.

Về tập cạnh E và trọng số của nó, một cạnh $(u,v) \in E$ tồn tại nếu và chỉ nếu:

1. u và v là các đỉnh thuộc tập đỉnh *V*.
2. u và v là các ô kề nhau theo bốn hướng:
3. Cả hai ô đều không phải là tường.

$$(x2,y2) \in \{ (x1+1,y1), (x1-1,y1), (x1,y1+1), (x1,y1-1) \};$$

4. Vì khả năng di chuyển trong mê cung là hai chiều, đồ thị thu được là **đồ thị vô hướng**:

$$(u,v) \in E \Leftrightarrow (v,u) \in E$$

5. Trong bài toán cơ bản, mỗi bước di chuyển giữa hai ô liền kề đều có chi phí như nhau, do đó trọng số được gán:

$$w(u,v)=1.$$

CHƯƠNG 3. THIẾT KẾ VÀ CÀI ĐẶT CHƯƠNG TRÌNH

3.1. Cấu trúc tổng thể hệ thống

3.1.1. Tổng quan kiến trúc

Hệ thống được chia thành hai phần chính:

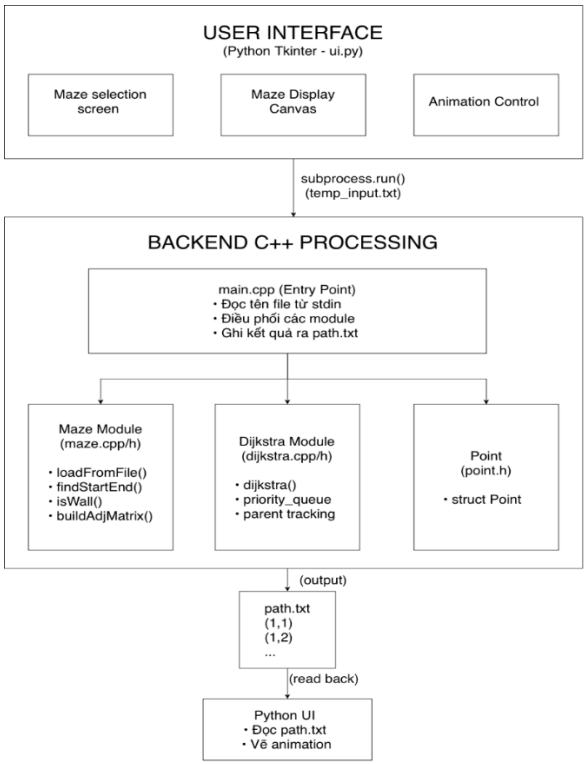
Backend (C++): Xử lý thuật toán và logic tính toán

- Module Maze: Quản lý dữ liệu mê cung
- Module Dijkstra: Thuật toán tìm đường ngắn nhất
- Module Main: Điều phối và tích hợp các module

Frontend (Python): Giao diện người dùng và trực quan hoá

- Module UI: Hiển thị đồ họa và tương tác với người dùng

3.1.2. Sơ đồ tổng quan



Hình 1. Sơ đồ tổng quan

3.1.3. Luồng dữ liệu

Hệ thống tìm đường trong mê cung gồm ba luồng chính: luồng đầu vào, luồng xử lý, và luồng đầu ra, phối hợp giữa giao diện Python và chương trình xử lý C++.

(1) Luồng đầu vào – Input Flow

Người dùng khởi chạy giao diện Python và chọn một trong các tệp mê cung có sẵn. Giao diện ghi tên tệp được chọn vào temp_input.txt, sau đó gọi chương trình C++ thông qua subprocess.run(). Chương trình C++ đọc tên tệp từ stdin, thực hiện Maze::loadFromFile() để nạp ma trận mê cung, đồng thời xác định tọa độ điểm bắt đầu và điểm kết thúc.

(2) Luồng xử lý – Processing Flow

Sau khi dữ liệu mê cung được nạp, chương trình C++ kiểm tra tính hợp lệ của mê

cung. Nếu hợp lệ, thuật toán Dijkstra được thực thi để tìm đường đi ngắn nhất. Trong quá trình chạy, chương trình duy trì ma trận dist, ma trận parent và một hàng đợi ưu tiên (min-heap). Khi tìm được đường đi, chương trình truy vết từ đích về nguồn, đảo ngược kết quả và ghi toàn bộ các tọa độ đường đi vào path.txt.

(3) Luồng đầu ra – Output Flow

Giao diện Python đọc nội dung từ path.txt. Nếu có đường đi, danh sách tọa độ được chuyển thành path_data và được hiển thị dưới dạng hoạt ảnh trên Canvas. Animation vẽ từng bước của đường đi theo thời gian, tạo ra kết quả trực quan từ điểm bắt đầu đến điểm kết thúc. Nếu không tồn tại đường đi, giao diện sẽ thông báo cho người dùng.

3.2. Mô tả các module chính

3.2.1. Module Maze (Đọc và xử lý mê cung)

Module Maze chịu trách nhiệm quản lý toàn bộ dữ liệu của mê cung, bao gồm việc đọc mê cung từ file, lưu trữ cấu trúc lưới 2D, xác định vị trí bắt đầu – kết thúc, truy xuất thông tin từng ô và xây dựng biểu diễn đồ thị tương ứng. Đây là lớp nền tảng mà các thuật toán tìm đường (như Dijkstra) dựa vào.

Point.h: Biểu diễn tọa độ trong mê cung

Point.h định nghĩa cấu trúc Point dùng để biểu diễn vị trí trong mê cung dưới dạng cặp tọa độ (x, y).

Cấu trúc Point bao gồm hai thành phần:

- x → chỉ số hàng
- y → chỉ số cột

Các toán tử hỗ trợ:

- operator== → để so sánh hai điểm có trùng vị trí hay không
- operator<< → để in điểm dưới dạng (x, y)

Nhờ cấu trúc này, việc lưu trữ, so sánh và hiển thị tọa độ trong quá trình tìm đường trở nên đơn giản và trực quan hơn.

Maze.h: Khai báo lớp và các hàm xử lý mê cung

Maze.h khai báo toàn bộ thành phần dữ liệu và các hàm giao tiếp cần thiết để các module khác có thể làm việc với mê cung. Cấu trúc maze_data được sử dụng để lưu mê cung dưới dạng ma trận số nguyên, trong đó:

- 1 → tường.
- 0 → ô đi được.
- 2 → điểm bắt đầu.
- 3 → điểm kết thúc.
- num_rows, num_cols → kích thước mê cung.
- start_point, end_point → vị trí lối vào và lối ra.

Những hàm chính được khai báo trong file này gồm:

- loadFromFile(string) → đọc mê cung từ file.
- findStartEnd()→ xác định vị trí start và end.
- getCell(x, y) → trả về giá trị tại vị trí (x, y).
- isWall(x, y)→ kiểm tra ô có phải tường/ngoài phạm vi.
- cellWeight(x, y) → trả về trọng số (mặc định 1)
- getRows(), getCols() → lấy kích thước mê cung

Maze.cpp: Triển khai toàn bộ quá trình xử lý mê cung

Maze.cpp hiện thực hóa các hàm đã khai báo trong **Maze.h.**

3.2.1.1. Đọc mê cung từ file

Hàm loadFromFile() chịu trách nhiệm mở file đầu vào và chuyển đổi nội dung trong file thành ma trận lưu trữ bên trong lớp Maze. Quá trình này bao gồm:

- Đọc kích thước mê cung (số hàng, số cột).
- Đọc từng giá trị trong lưới và gán vào maze_data.
- Nhận diện các giá trị đặc biệt (2 = Start, 3 = End) để hỗ trợ bước xử lý kế tiếp.

Mục tiêu của bước này là đảm bảo toàn bộ dữ liệu trong file được ánh xạ chính xác sang cấu trúc mê cung trong chương trình.

```
bool Maze::loadFromFile(const string &filename)
{
    ifstream file_in(filename);
    if (!file_in.is_open())
    {
        cout<<"Không thể mở file: "<<filename<<endl;
        return false;
    }

    // Đọc hàng và cột
    file_in >> num_rows >> num_cols;

    // Bỏ phần còn lại của dòng đầu
    string dummy;
    getline(file_in, dummy);
```

```
maze_data.clear();
maze_data.push_back(vector<int>());

// Khởi tạo mê cung rỗng
for (int i = 1; i <= num_rows; i++)
{
    vector<int> row(num_cols + 1, 1);
    maze_data.push_back(row);
}

// Đọc từng dòng ký tự của mê cung
for (int i = 1; i <= num_rows; i++)
{
    for (int j = 1; j <= num_cols; j++)
    {
        int val;
        if (!(file_in >> val))    // nếu đọc thất bại (thiếu dữ liệu, EOF, lỗi định dạng)
        {
            file_in.clear();      // xóa cờ lỗi
            val = 1;              // tự động điền tường
```

```
            }


            maze_data[i][j] = val;


            if (val == 2)
                start_point = Point(i, j);
            else if (val == 3)
                end_point = Point(i, j);
        }
    }
    file_in.close();
    return true;
}
```

3.2.1.2. Xác định start và end

Sau khi nạp dữ liệu, hàm findStartEnd() tiến hành quét toàn bộ ma trận để tìm hai điểm quan trọng:

- Điểm bắt đầu
- Điểm kết thúc

Khi tìm thấy, các giá trị này được lưu lại để các thuật toán tìm đường có thể sử dụng. Nếu file đầu vào thiếu Start hoặc End, chương trình có thể phát hiện và báo lỗi.

3.2.1.3. Lưu ma trận mê cung

Tất cả dữ liệu đọc từ file được lưu trong maze_data, một ma trận hai chiều biểu diễn trạng thái của từng ô. Cấu trúc này đảm bảo:

- Truy xuất ô được thực hiện nhanh
- Thao tác kiểm tra tường và phạm vi diễn ra chính xác
- Dễ dàng ánh xạ mê cung thành đồ thị

Phần này là nền tảng để các hàm khác hoạt động đúng.

3.2.1.4. Kiểm tra tường và phạm vi

Các hàm như isWall() và getCell() được triển khai trong Maze.cpp nhằm kiểm tra:

- Liệu một ô có nằm ngoài biên mê cung
- Liệu ô đó có phải tường hay không
- Liệu ô có thể di chuyển vào được hay không

Những kiểm tra này giúp bảo vệ thuật toán không mở rộng sang các vị trí sai hoặc vượt biên.

```
bool Maze::isWall(int x, int y) const
{
   if (x < 1 || x > num_rows   || y < 1 || y > num_cols)
   {
      return true;
   }
   return maze_data[x][y] == 1;
}
```

3.2.1.5. Cung cấp dữ liệu cho Dijkstra

Maze.cpp cung cấp đầy đủ thông tin để thuật toán Dijkstra hoạt động:

- Truy xuất trạng thái từng ô
- Kiểm tra tính hợp lệ khi mở rộng đỉnh
- Trả về kích thước mê cung
- Cung cấp vị trí Start và End
- Trả về trọng số của một ô

Nhờ được cung cấp dữ liệu, thuật toán Dijkstra không cần biết cách mê cung được lưu trữ, chỉ cần gọi các hàm từ Maze.

3.2.2. Module Dijkstra (Tìm đường ngắn nhất)

Module Dijkstra dùng để tìm đường đi ngắn nhất trong mê cung (Maze) từ điểm bắt đầu đến điểm kết thúc. Dữ liệu đầu vào: Mê cung Maze (một lưới 2D, 0 = tường, 1 = ô đi được), điểm bắt đầu và điểm kết thúc. Kết quả đầu ra: Danh sách các ô đi theo đường ngắn nhất (theo số bước).

Dijkstra.h: dùng để khai báo các hàm và kiểu dữ liệu sẽ dùng trong module. Cấu trúc HD dùng trong priority queue để lưu thông tin ô hiện tại (row, col) và khoảng cách (dist) đến điểm bắt đầu (start). Thuật toán Dijkstra vector<pair<int,int>> dijkstra(const Maze &maze) trả về danh sách các ô đi theo đường ngắn nhất. Hàm void Dijkstra(const Maze &maze) để in kết quả ra màn hình, gọi hàm dijkstra.

```
#ifndef DIJKSTRA_H
#define DIJKSTRA_H
#include <iostream>
#include <vector>
```

```
#include <queue>
#include <fstream>
#include <algorithm>
using namespace std;
class Maze;
struct Point;
struct HD
{
    int row, col, dist;
    bool operator>(const HD &other) const { return dist > other.dist; }
};

std::vector<std::pair<int, int>> dijkstra(const Maze &maze);
void Dijkstra(const Maze &maze);
#endif
```

Dijkstra.cpp: là nơi cài đặt toàn bộ quá trình xử lý của thuật toán Dijkstra để tìm đường ngắn nhất trong mê cung. Chương trình bắt đầu bằng cách lấy số hàng, số cột, điểm bắt đầu và điểm kết thúc từ lớp Maze. Sau khi thu thập dữ liệu ban đầu, chương trình khởi tạo một mảng dist dùng để lưu khoảng cách nhỏ nhất từ điểm bắt đầu đến mọi ô trong mê cung. Sau đó, mảng parent được khởi tạo nhằm lưu lại ô trước đó của mỗi vị trí, phục vụ cho việc truy vết đường đi sau khi thuật toán hoàn thành. Tiếp tục sử dụng một hàng đợi ưu tiên priority queue với cấu trúc HD để lấy ra ô có khoảng cách nhỏ nhất tại mỗi bước. Mỗi ô được xét duyệt bốn hướng: lên, xuống, trái, phải. Khi thuật toán

kết thúc, chương trình tiến hành truy vết đường đi bằng cách bắt đầu từ điểm kết thúc và lần theo các giá trị trong mảng parent cho đến khi quay về điểm bắt đầu. Sau đó đường đi được đảo ngược để có kết quả theo đúng thứ tự từ điểm bắt đầu đến điểm kết thúc. Toàn bộ đường đi tìm được sẽ được ghi file path.txt.

```
#include "dijkstra.h"
#include "maze.h"
#include <fstream>

vector<pair<int, int>> dijkstra(const Maze &maze)
{
    int rows = maze.getRows(), cols = maze.getCols();
    Point start = maze.getStart();
    Point end = maze.getEnd();
    const int INF = 1e9;

    vector<vector<int>> dist(rows + 1, vector<int>(cols + 1, INF));
    vector<vector<pair<int, int>>> parent(rows + 1, vector<pair<int, int>>(cols + 1, {-1,
-1}));

    priority_queue<HD, vector<HD>, greater<HD>> pq;
    dist[start.x][start.y] = 0;
    pq.push({start.x, start.y, 0});
```

```
int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};

while (!pq.empty())
{
    HD curr = pq.top();
    pq.pop();

    int r = curr.row;
    int c = curr.col;
    int d = curr.dist;

    if (r == end.x && c == end.y)
        break;
    if (d > dist[r][c])
        continue;

    for (int i = 0; i < 4; i++)
    {
        int num_row = r + dx[i];
```

```
            int num_col = c + dy[i];
            if (!maze.isWall(num_row, num_col))
            {
                int num_dist = dist[r][c] + 1;
                if (num_dist < dist[num_row][num_col])
                {
                    dist[num_row][num_col] = num_dist;
                    parent[num_row][num_col] = {r, c};
                    pq.push({num_row, num_col, num_dist});
                }
            }
        }
    }
    ofstream out("path.txt");
    if (dist[end.x][end.y] == INF)
    {
        out << "Khong ton tai duong di!\n";
        out.close();
        return {};
    }
```

```
    vector<pair<int, int>> path;
    pair<int, int> cur = {end.x, end.y};


    while (!(cur.first == -1 && cur.second == -1))
    {
        path.push_back(cur);
        cur = parent[cur.first][cur.second];
    }
    reverse(path.begin(), path.end());
    out << "Duong di ngan nhat:\n";
    for (int i = 0; i < path.size(); i++)
    {
        pair<int, int> p = path[i];
        out << "(" << p.first << "," << p.second << ")\n";
    }
    out.close();
    return path;
}
```

3.2.3. Module Main & Kiểm thử tích hợp

main.cpp: Điểm khởi đầu chương trình

File **main.cpp** giữ vai trò là điểm khởi đầu của toàn bộ chương trình, đảm nhiệm việc điều phối luồng xử lý giữa các module chính, bao gồm:

- **Module Maze**: chịu trách nhiệm tải và kiểm tra dữ liệu mê cung.
- **Module Dijkstra**: triển khai thuật toán tìm đường đi ngắn nhất.

File main.cpp giúp kết nối các module này với nhau, đảm bảo chương trình vận hành theo đúng thứ tự xử lý và hỗ trợ kiểm thử tích hợp giữa các thành phần.

Luồng xử lý chính:

Luồng xử lý được kiểm soát hoàn toàn trong hàm main(), bảo đảm rằng mỗi bước xử lý đều được kiểm tra lỗi và phản hồi rõ ràng cho người dùng.

Mã nguồn main.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <utility>
#include <fstream>
#include "maze.h"
#include "dijkstra.h"
using namespace std;

int main()
{
	Maze maze;
	string filename;
	cin >> filename;

	if (!maze.loadFromFile(filename))
	{
		cout << "Không thể mở file " << filename << endl;
		return 0;
```

```
}

// Kiểm tra lối vào/lối ra
if (!maze.findStartEnd())
{
        cout << "Mê cung không hợp lệ (không tồn tại lối vào hoặc lối ra)!";
        return 0;
}

// Chạy Dijkstra
vector<pair<int, int>> path = dijkstra(maze);

if (path.empty())
{
        cout << "Mê cung không có không có đường thoát!" << endl;
        return 0;
}

ofstream fout("path.txt");
if (fout.is_open())
{
```

```
            for (auto &p : path)
            {
                fout << "(" << p.first << "," << p.second << ")\n";
            }
            fout.close();
        }
        else
        {
            cout << "Không thể tạo file path.txt" << endl;
        }

        return 0;
}
```

Chi tiết xử lý trong main.cpp

a. Nhận tên file đầu vào

Chương trình yêu cầu người dùng nhập tên file chứa ma trận mê cung:

– string filename;

– cin >> filename;

File đầu vào bao gồm:

– Số hàng, số cột,

– Dữ liệu từng ô của mê cung (0: đường đi, 1: tường, 2: Start, 3: End).

b. Tải dữ liệu mê cung

Gọi maze.loadFromFile(filename) để đọc dữ liệu từ file.

Quá trình này sẽ:

- Xác định kích thước mê cung (số hàng, số cột).
- Lưu ma trận mê cung dưới dạng 2D.
- Xác định vị trí ô lối vào (giá trị 2) và lối ra (giá trị 3).

```
if (!maze.loadFromFile(filename))
{
    cout << "Không thể mở file " << filename << endl;
    return 0;
}
```

Hàm loadFromFile() trong module Maze chịu trách nhiệm:

- Mở file.
- Đọc kích thước mê cung.
- Lưu ma trận vào cấu trúc 2D.
- Quét tìm vị trí Start và End (nếu có).

Nếu không mở được file → thông báo lỗi và dừng chương trình.

c. Kiểm tra tính hợp lệ của mê cung

Sau khi tải dữ liệu, chương trình kiểm tra tính hợp lệ của mê cung:

- Hàm maze.findStartEnd() sẽ kiểm tra:
  + Có tồn tại điểm bắt đầu (Start) không.

+ Có tồn tại điểm kết thúc (End) không.

– Nếu thiếu một trong hai, thông báo lỗi và kết thúc chương trình.

```
if (!maze.findStartEnd())
{
        cout << "Mê cung không hợp lệ (không tồn tại lối vào hoặc lối ra)!";
        return 0;
}
```

d. Gọi thuật toán Dijkstra để tìm đường đi

Gọi hàm dijkstra(maze) để tìm đường đi ngắn nhất từ Start đến End.

– Kết quả trả về là một vector<pair<int, int>> chứa danh sách các ô theo thứ tự đi của đường đi.

– Nếu không tìm được đường đi, vector trả về rỗng và báo "Mê cung không có đường thoát".

```
vector<pair<int, int>> path = dijkstra(maze);
if (path.empty())
{
     cout << "Mê cung không có đường thoát!" << endl;
      return 0;}
```

e. Xử lý kết quả

– Nếu tìm thấy đường đi:

+ Ghi tất cả tọa độ của các ô trong đường đi vào file path.txt.

+ Format mỗi ô: (hàng,cột), mỗi ô trên một dòng.

– Nếu không thể mở file path.txt để ghi, chương trình thông báo lỗi.

```
ofstream fout("path.txt");
if (fout.is_open())
{
	for (auto &p : path)
	{
		fout << "(" << p.first << "," << p.second << ")\n";
	}
	fout.close();
}
else
{
	cout << "Không thể tạo file path.txt" << endl;
}
```

f. Kiểm tra tích hợp

Mục đích của kiểm thử tích hợp là đảm bảo các module chính của chương trình - bao gồm Maze, Dijkstra và Main - hoạt động đúng khi được kết hợp với nhau.

Hàm main() là vị trí lý tưởng để thực hiện kiểm thử tích hợp, bởi vì nó đóng vai trò điều phối toàn bộ quy trình xử lý: đọc và phân tích mê cung (Maze), kiểm tra tính hợp lệ (Validate), tìm đường đi ngắn nhất bằng thuật toán Dijkstra, và cuối cùng là xuất kết quả. Trong quá trình này, mỗi bước đều có cơ chế xử lý lỗi riêng, giúp dễ dàng xác định

và đánh giá xem toàn bộ hệ thống có vận hành chính xác hay không. Nhờ đó, chương trình có thể kiểm tra được nhiều tình huống khác nhau như: tệp đầu vào không tồn tại, mê cung thiếu vị trí Start/End, mê cung không có đường thoát, đường đi hợp lệ, hoặc quá trình ghi kết quả ra file thành công hay thất bại. Điều này đảm bảo tính toàn vẹn và độ tin cậy của toàn bộ hệ thống khi triển khai thực tế.

Bảng kịch bản kiểm thử tích hợp (Integration Test Scenarios Table):

Testcase	**Kết quả mong đợi**	**Kết quả thực tế**
1. input1.txt (mê cung 12×15, có đường đi hợp lệ)	Chương trình đọc file thành công, tìm được đường đi S → E bằng Dijkstra. Tạo file **path.txt** và ghi đầy đủ các toạ độ đường đi.	Đúng như mong đợi: Tìm được đường đi và tạo file **path.txt**.
2. Tên file không tồn tại (ví dụ: abc.txt)	Chương trình thông báo lỗi: **"Không thể mở file <tên file>"** và kết thúc chương trình.	Đúng như mong đợi: Hiển thị thông báo lỗi *"Không thể mở file <tên file>"*.
3. Mê cung thiếu S hoặc E (không có lối vào/lối ra)	Chương trình thông báo: **"Mê cung không hợp lệ (không tồn tại lối vào hoặc lối ra)!"**	Đúng như mong đợi: Hiển thị thông báo lỗi tương ứng.
4. Mê cung hợp lệ nhưng không tồn tại đường đi S → E	Chương trình in ra: **"Mê cung không có không có đường thoát!"**	Đúng như mong đợi: Thông báo không tìm được đường đi.

5. Lỗi khi ghi file path.txt (ví dụ: không có quyền ghi, thư mục bị khoá)	Chương trình hiển thị lỗi: **"Không thể tạo file path.txt"**	Đúng như mong đợi: Thông báo lỗi khi không tạo được file.

Bảng 2. Bảng kịch bản kiểm thử tích hợp

Phương pháp kiểm thử:

– Kiểm thử White-box: Kiểm tra từng luồng xử lý trong code.
– Kiểm thử Black-box: Kiểm tra input/output mà không quan tâm cài đặt bên trong.
– Kiểm thử Integration: Kiểm tra sự tương tác giữa các module.

g. Biên dịch chương trình

```
g++ -o main main.cpp maze.cpp dijkstra.cpp
```

Lệnh này biên dịch tất cả các file nguồn C++ và tạo ra file thực thi main.exe (Windows) hoặc main (Linux/Mac).

3.2.4. Module Giao diện người dùng (UI)

3.2.4.1. Màn hình chọn mê cung (Maze Selection Screen)

Màn hình này là màn hình đầu tiên xuất hiện khi người dùng khởi động chương trình. Chức năng chính của màn hình bao gồm: cho phép người dùng lựa chọn tệp mê cung định dạng .txt chọn từ danh sách sẵn có; hiển thị giao diện với màu sắc chủ đạo phù hợp theo chủ đề; và khi người dùng hoàn tất việc lựa chọn, giao diện sẽ tự động chuyển sang màn hình hiển thị mê cung. Thiết kế của màn hình nhằm mô phỏng cảm giác như một chú vịt con đang đứng trước "bản đồ mê cung", chuẩn bị bắt đầu cuộc hành trình khám phá, tạo sự thân thiện và sinh động cho người dùng ngay từ lần đầu tiên sử dụng.

***Tiêu đề chương trình:**

- Tiêu đề: **"THE BABY DUCKLING'S ADVENTURE!"**
- Font chữ Comic Sans MS đậm, tạo cảm giác vui nhộn đúng với chủ đề "chú vịt con".
- Nền xanh đậm (#2c3e50) và chữ vàng (#FFD700), đồng bộ với màu nhân vật vịt.

***Danh sách 6 level mê cung:**

Chương trình hiển thị 6 nút tương ứng với 6 độ khó khác nhau:

Level	**File**	**Kích thước**	**Màu sắc**
Level 1	input1.txt	12×15	#90EE90
Level 2	input2.txt	15×15	#FFD700
Level 3	input3.txt	15×15	#FFA500
Level 4	input4.txt	18×18	#FF69B4
Level 5	input5.txt	22×22	#9370DB
Level 6	input6.txt	25×25	#FF6347

Bảng 3. Danh sách 6 level mê cung

Ý nghĩa thiết kế: Mỗi cấp độ (level) trong giao diện được thiết kế với màu sắc riêng, giúp người dùng dễ dàng phân biệt mức độ khó. Các nút chức năng được bố trí lớn, bo tròn, thuận tiện cho việc thao tác. Đồng thời, tổng thể giao diện thể hiện rõ phong cách nhẹ nhàng, sinh động, mang đậm chủ đề hoạt hình, tạo cảm giác thân thiện và hấp dẫn cho người dùng.

***Nút điều khiển (Control Buttons):**

Giao diện cung cấp một nút điều khiển chính: Exit Game - dùng để thoát ứng dụng.

Nút thoát được thiết kế với màu đỏ để dễ nhận biết và có sự tách biệt với các nút chọn level. Chức năng này đảm bảo người dùng có thể rời chương trình một cách chủ động và nhanh chóng.

```
    def exit_app(self):
        """Exit the application"""
        if messagebox.askokcancel("Exit", "Are you sure you want to abandon the duckling? □"):
            self.root.quit()
```

Hàm exit_app() được sử dụng để xử lý việc thoát chương trình. Khi người dùng nhấn nút thoát, hệ thống sẽ hiển thị một hộp thoại xác nhận với thông điệp nhắc nhở:

"Are you sure you want to abandon the duckling?"

Thiết kế này không chỉ nhắc nhở người dùng trước khi rời khỏi ứng dụng mà còn tạo sự gắn kết, mang lại trải nghiệm gần gũi, sinh động. Nếu người dùng chọn **OK**, chương trình sẽ kết thúc bằng cách gọi self.root.quit(). Ngược lại, nếu người dùng chọn **Cancel**, ứng dụng sẽ tiếp tục chạy bình thường, đảm bảo không gây gián đoạn trải nghiệm.

*** Xử lý sự kiện chọn mê cung:**

Khi người dùng nhấn vào bất kỳ level nào, chương trình gọi:

```
    def select_maze(self, filename):
        """Handle maze selection"""
        if not os.path.exists(filename):
            messagebox.showerror(
                "⊘ Oops! File Not Found",
```

```
                f"Oh no! The file '{filename}' flew away! \n\n"
                f"Please make sure all maze files are swimming\n"
                f"in the same pond (folder) as this program." )
            return
        self.on_select(filename)
```

Chức năng:

Trước tiên, chương trình kiểm tra tính tồn tại của tệp mê cung. Trong trường hợp tệp bị xóa hoặc được đặt sai thư mục, hệ thống sẽ hiển thị thông báo lỗi mang tính hài hước:
" Oh no! The file 'input_.txt' flew away!

Please make sure all maze files are swimming

in the same pond (folder) as this program."
Điều này vừa nhắc nhở người dùng, vừa tạo trải nghiệm thân thiện và sinh động.

Nếu tệp hợp lệ, chương trình sẽ gọi phương thức self.on_select(filename) để tiến hành xử lý, đồng thời giao diện sẽ tự động chuyển sang màn hình giải mê cung, cho phép người dùng tiếp tục cuộc hành trình khám phá mê cung.

3.2.4.2. Lớp MazeUI - Hiển thị mê cung và đường đi

Lớp MazeUI được xây dựng với mục tiêu cung cấp một giao diện trực quan và sinh động cho quá trình hiển thị mê cung. Lớp này chịu trách nhiệm biểu diễn mê cung dưới dạng lưới đồ họa, trong đó mỗi ô được mô phỏng bằng các hiệu ứng như màu nước, bờ cỏ, hoa súng cũng như các đối tượng đặc trưng gồm vịt con (điểm xuất phát) và vịt mẹ (điểm đích). Bên cạnh đó, lớp cũng đảm nhiệm việc gán màu sắc và hiệu ứng phù hợp cho từng loại địa hình dựa trên dữ liệu đọc từ tệp đầu vào, bảo đảm sự nhất quán giữa dữ liệu và phần hiển thị.

Ngoài chức năng mô phỏng trực quan, MazeUI còn hỗ trợ các thao tác chính của người dùng như khởi chạy quá trình giải mê cung, quay trở về màn hình lựa chọn ban đầu hoặc xoá đường đi để quan sát lại kết quả. Toàn bộ bố cục giao diện được tổ chức rõ ràng, bao gồm thanh tiêu đề, vùng canvas hiển thị mê cung, bảng chú thích màu sắc và cụm các nút điều hướng. Nhờ thiết kế này, giao diện mang lại trải nghiệm thân thiện, dễ quan sát và phù hợp với định hướng xây dựng ứng dụng trực quan, sinh động theo phong cách hoạt hình.

Chức năng tổng quan:

Lớp MazeUI chịu trách nhiệm xây dựng và điều khiển toàn bộ giao diện hiển thị mê cung. Đây là module trung tâm trong việc trực quan hóa dữ liệu mê cung, vẽ các thành phần đồ họa, xử lý thao tác của người dùng và thể hiện kết quả của thuật toán tìm đường. Cụ thể, lớp này đảm nhiệm bốn nhóm chức năng chính:

- Đọc dữ liệu mê cung từ tệp đầu vào.
- Vẽ mê cung lên Canvas với các lớp trang trí đặc trưng.
- Hiển thị đường đi sau khi thuật toán tìm đường được thực thi.
- Cung cấp các nút chức năng như: *Giải, Quay lại, Làm mới*.

Mô tả chi tiết:

a. Thanh tiêu đề

Thanh tiêu đề là thành phần được đặt ở vị trí đầu trang giao diện, giữ vai trò quan trọng trong việc định hướng chủ đề của mô phỏng. Nội dung được hiển thị là câu mô tả hành trình: **"THE BABY DUCKLING IS LOST IN THE REED FIELD"**, giúp người dùng nhanh chóng nhận diện bối cảnh: chú vịt con đang bị lạc trong đồng lau và cần được dẫn đường trở về với vịt mẹ. Đây là điểm nhấn đầu tiên khi người dùng mở giao diện, góp phần tạo cảm giác sinh động và cuốn hút.

Bên cạnh nội dung, thanh tiêu đề còn được thiết kế với tông màu lam đậm chủ đạo. Việc lựa chọn màu sắc này nhằm đồng bộ hóa giao diện với chủ đề "mặt nước – đầm cỏ", tạo sự hài hòa với tổng thể màu nền và các họa tiết trang trí của chương trình. Kiểu chữ và bố cục cũng được lựa chọn kỹ, sử dụng font cỡ lớn, đậm và có tính trang trí để tăng tính trực quan và làm nổi bật thông điệp chính.

```
def create_widgets(self):
        # Title bar with pond theme
        title_frame = tk.Frame(self.frame, bg="#2E86AB")
        title_frame.pack(fill=tk.X)
        title_label = tk.Label(
            title_frame,
            text="THE BABY DUCKLING IS LOST IN THE REED FIELD",
            font=("Comic Sans MS", 22, "bold"),
            bg="#2E86AB",
            fg="white",
            pady=12
        )
        title_label.pack()

        subtitle_label = tk.Label(
            title_frame,
            text="💚 Help the baby duckling find its mother! 💚",
```

```
            font=("Arial", 13, "italic"),
            bg="#2E86AB",
            fg="#E8F4F8",
            pady=5
        )

        subtitle_label.pack()
```

Thanh tiêu đề giúp định hướng chủ đề mô phỏng, tạo điểm nhấn thẩm mỹ và mang lại trải nghiệm trực quan ngay từ trang mở đầu. Đồng thời, phần tiêu đề và phụ đề hỗ trợ người dùng nhận biết bước thao tác cần thực hiện tiếp theo (chọn "Help the Duckling!"), đảm bảo dòng thao tác được rõ ràng và liền mạch.

b. Các nút chức năng

Lớp **MazeUI** cung cấp ba nút điều khiển chính, giúp người dùng tương tác trực tiếp với hệ thống giải mê cung và quản lý việc hiển thị đường đi:

- **Help the Duckling**: Khi người dùng nhấn nút này, thuật toán giải mê cung sẽ được khởi chạy, tìm đường đi từ vị trí bắt đầu (vịt con) đến đích (vịt mẹ). Đây là hành động chính để quan sát quá trình giải mê cung.
- **Back to Pond**: Nút này cho phép người dùng quay trở lại màn hình chọn cấp độ, từ đó có thể lựa chọn mê cung khác để thử nghiệm.
- **Clear Path**: Dùng để làm mới canvas, xóa tất cả các đường đi đã vẽ trước đó, trả canvas về trạng thái ban đầu. Nút này giúp người dùng dễ dàng thử nghiệm lại mà không cần thoát khỏi màn hình hiện tại.

Các nút được thiết kế với kích thước lớn, phông chữ rõ ràng, phong cách bo tròn và màu sắc tươi sáng, nhằm tạo cảm giác thân thiện và dễ thao tác, phù hợp với mọi độ tuổi. Việc phân chia các nút thành một khung riêng giúp giao diện trực quan và dễ sử dụng.

```
# Button Frame
button_frame = tk.Frame(self.frame, bg="#E8F4F8")
button_frame.pack(pady=15)

self.solve_btn = tk.Button(
    button_frame,
    text="Help the Duckling!",
    font=("Comic Sans MS", 11, "bold"),
    bg="#27ae60",
    fg="black",
    activebackground="#229954",
    padx=20,
    pady=10,
    command=self.solve_maze,
    cursor="hand2",
    state=tk.NORMAL
)
self.solve_btn.grid(row=0, column=0, padx=8)
```

```
        self.back_btn = tk.Button(
            button_frame,
            text="Back to Pond",
            font=("Comic Sans MS", 11, "bold"),
            bg="#3498db",
            fg="black",
            activebackground="#2980b9",
            padx=20,
            pady=10,
            command=self.back_to_selection,
            cursor="hand2"
        )
        self.back_btn.grid(row=0, column=1, padx=8)

        self.restart_btn = tk.Button(
            button_frame,
            text="Clear Path",
            font=("Comic Sans MS", 11, "bold"),
            bg="#f39c12",
            fg="black",
```

```
            activebackground="#e67e22",
            padx=20,
            pady=10,
            command=self.restart,
            cursor="hand2"
        )
        self.restart_btn.grid(row=0, column=2, padx=8)
```

Các nút chức năng này cung cấp phương tiện để người dùng điều khiển quá trình mô phỏng: từ việc khởi động thuật toán giải mê cung, quay lại màn hình lựa chọn cấp độ, đến làm mới đường đi. Nhờ vậy, giao diện trở nên trực quan, thân thiện và dễ thao tác, đồng thời hỗ trợ tốt cho việc quan sát và trải nghiệm quá trình giải mê cung.

c. Khu vực Canvas

Canvas là khu vực đồ họa chính, nơi toàn bộ mê cung và đường đi được vẽ. Lớp MazeUI bổ sung thêm hiệu ứng trang trí như:

- Viền đầm cỏ (*pond bank*)
- Hiệu ứng mặt nước
- Lá sen
- Vịt con và vịt mẹ
- Đường đi của thuật toán

Canvas còn được trang bị thanh cuộn ngang và dọc giúp người dùng quan sát những mê cung có kích thước lớn.

Code minh họa tạo Canvas và thanh cuộn:

```
# Canvas container with pond border
canvas_container = tk.Frame(self.frame, bg="#E8F4F8")
canvas_container.pack(pady=10, padx=30, fill=tk.BOTH, expand=True)

# Decorative frame - grass border
deco_frame = tk.Frame(canvas_container, bg="#90C695", bd=10, relief=tk.RIDGE)
deco_frame.pack(fill=tk.BOTH, expand=True)

canvas_frame = tk.Frame(deco_frame, bg="#5DADE2")
canvas_frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)



# Scrollbars
v_scrollbar = tk.Scrollbar(canvas_frame, orient=tk.VERTICAL)
v_scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

h_scrollbar = tk.Scrollbar(canvas_frame, orient=tk.HORIZONTAL)
h_scrollbar.pack(side=tk.BOTTOM, fill=tk.X)

# Canvas with water background
```

```
self.canvas = tk.Canvas(
        canvas_frame,
        bg="#5DADE2",
        highlightthickness=0,
        yscrollcommand=v_scrollbar.set,
        xscrollcommand=h_scrollbar.set
)
self.canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)


# Liên kết scrollbars với canvas
v_scrollbar.config(command=self.canvas.yview)
h_scrollbar.config(command=self.canvas.xview)
```

Chức năng: Hiển thị toàn bộ mê cung và cho phép cuộn tối ưu với mê cung kích thước lớn.

d. Tải mê cung từ tệp (load_maze)

Hàm load_maze() chịu trách nhiệm đọc dữ liệu, chuyển đổi về ma trận và kiểm tra các điều kiện hợp lệ:

- Số dòng và số cột của mê cung.
- Tồn tại điểm vào (2) và điểm ra (3).
- Tự động thêm tường (1) nếu dữ liệu thiếu.

Code minh họa:

```
def load_maze(self, filename):
        try:
            with open(filename, 'r') as f:
                lines = f.readlines()
                if not lines:
                    raise ValueError("Empty maze file")

                first_line = lines[0].strip().split()
                if len(first_line) < 2:
                    raise ValueError("Invalid maze format")

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

                self.maze_data = []
                self.has_entrance = False
                self.has_exit = False

                for i in range(1, min(self.rows + 1, len(lines))):
                    row_values = lines[i].strip().split()
```

```
                row = []
                for j in range(self.cols):
                    if j < len(row_values):
                        val = int(row_values[j])
                        row.append(val)
                        if val == 2:
                            self.has_entrance = True
                        elif val == 3:
                            self.has_exit = True
                    else:
                        row.append(1)  # Fill with walls if data missing
                self.maze_data.append(row)

            # Fill remaining rows if needed
            while len(self.maze_data) < self.rows:
                self.maze_data.append([1] * self.cols)
```

Chức năng: Bảo đảm dữ liệu mê cung hợp lệ trước khi vẽ và xử lý.

e. Vẽ mê cung

Mỗi ô mê cung được vẽ với màu sắc đại diện:

Mã	Ý nghĩa	Màu sắc

1	Tường/cỏ	Xanh lá
0	Đường đi	Xanh nước
2	Vịt con	Vàng
3	Vịt mẹ	Trắng

Bảng 4. Màu sắc đại diện của mỗi ô mê cung

Ngoài ra, lớp còn có các hàm trang trí giúp tăng tính sinh động.

Hàm vẽ cỏ bờ đầm:

```
    def draw_pond_bank(self, x1, y1, x2, y2):
        """Draw pond banks with grass/reed texture"""
        # Base green color
        self.canvas.create_rectangle(x1, y1, x2, y2, fill="#7CB342",
outline="#689F38", width=2)

        # Add texture with darker green streaks
        for i in range(3):
            offset = i * (self.cell_size // 4)
            self.canvas.create_line(x1 + offset, y1, x1 + offset, y2,
                                    fill="#558B2F", width=2)
```

```
            # Add small grass details
            for i in range(2, self.cell_size - 2, 6):
                self.canvas.create_line(x1 + i, y2 - 5, x1 + i + 2, y1 + 5,
                                        fill="#8BC34A", width=1)
```

Hàm vẽ mặt nước và lá sen:

```
    def draw_water_with_lily(self, x1, y1, x2, y2):
        """Draw water with subtle wave pattern"""
        # Base water color
        self.canvas.create_rectangle(x1, y1, x2, y2, fill="#5DADE2",
outline="#4A9BC7", width=1)

        # Add wave pattern
        mid_y = (y1 + y2) // 2
        self.canvas.create_line(x1, mid_y, x2, mid_y, fill="#81C9F0", width=1,
smooth=True)

        # Occasionally add a lily pad
        import random
        if random.random() < 0.15:  # 15% chance
            self.draw_lily_pad(x1, y1, x2, y2)
```

```
    def draw_lily_pad(self, x1, y1, x2, y2):
        """Draw a small lily pad"""
        center_x = (x1 + x2) // 2
        center_y = (y1 + y2) // 2
        size = self.cell_size // 3

        # Lily pad
        self.canvas.create_oval(center_x - size//2, center_y - size//2,
                                center_x + size//2, center_y + size//2,
                                fill="#7CB342", outline="#689F38", width=1)

        # Small notch
        self.canvas.create_line(center_x + size//2, center_y,
                                center_x + size//3, center_y,
                                fill="#689F38", width=2)
```

Hàm vẽ vịt con (Start):

```
def draw_baby_duck(self, x1, y1, x2, y2):
        """Draw a cute baby duck"""
        center_x = (x1 + x2) // 2
```

```
        center_y = (y1 + y2) // 2
        size = self.cell_size // 2

        # Duck body
        self.canvas.create_oval(center_x - size//2, center_y - size//3,
                                center_x + size//2, center_y + size//3,
                                fill="#FFD700", outline="#FFA500", width=2)

        # Duck head
        head_size = size // 2.5
        self.canvas.create_oval(center_x - head_size//2, center_y - size//2,
                                center_x + head_size//2, center_y - size//6,
                                fill="#FFD700", outline="#FFA500", width=2)

        # Duck beak
        self.canvas.create_polygon(
            center_x + head_size//2, center_y - size//3,
            center_x + head_size, center_y - size//3,
            center_x + head_size//2, center_y - size//4,
            fill="#FF8C00", outline="#FF6347"
        )
```

```
        # Eye
        self.canvas.create_oval(center_x - 2, center_y - size//3,
                                center_x + 2, center_y - size//3 + 4,
                                fill="black")
```

Hàm vẽ vịt mẹ (Goal):

```
def draw_mother_swan(self, x1, y1, x2, y2):
        """Draw a mother swan/duck"""
        center_x = (x1 + x2) // 2
        center_y = (y1 + y2) // 2
        size = self.cell_size // 1.8

        # Swan body
        self.canvas.create_oval(center_x - size//2, center_y - size//4,
                                center_x + size//2, center_y + size//3,
                                fill="white", outline="#CCCCCC", width=2)

        # Swan neck (curved)
        neck_points = [
```

```
            center_x - size//4, center_y - size//6,
            center_x - size//3, center_y - size//2,
            center_x - size//4, center_y - size//1.5
        ]
        self.canvas.create_line(neck_points, fill="white", width=6, smooth=True)
        self.canvas.create_line(neck_points, fill="#E0E0E0", width=5, smooth=True)

        # Swan head
        head_size = size // 3
        self.canvas.create_oval(center_x - size//3, center_y - size//1.4,
                                center_x - size//6, center_y - size//2,
                                fill="white", outline="#CCCCCC", width=2)

        # Swan beak

        self.canvas.create_polygon(
            center_x - size//3, center_y - size//1.5,
            center_x - size//2.2, center_y - size//1.5,
            center_x - size//3, center_y - size//1.7,
            fill="#FF8C00", outline="#FF6347"
        )
```

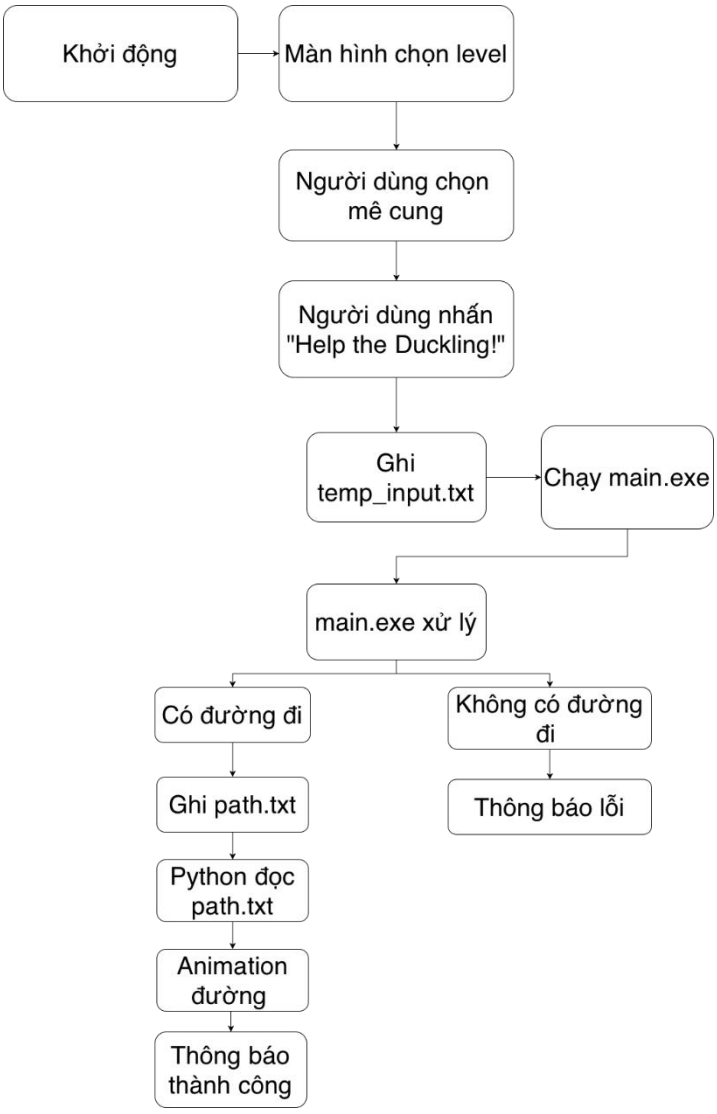
```
# Eye
self.canvas.create_oval(center_x - size//4, center_y - size//1.5,
                        center_x - size//5, center_y - size//1.7,
                        fill="black")
```

Chức năng: Trực quan hóa mê cung sinh động, hỗ trợ việc quan sát đường đi tìm được.

Tổng thể, lớp **MazeUI** giữ vai trò trọng tâm trong hệ thống giao diện của ứng dụng. Thông qua việc kết hợp hài hòa giữa cấu trúc dữ liệu, hiệu ứng đồ họa và các chức năng điều khiển, lớp không chỉ giúp quá trình giải mê cung trở nên trực quan và dễ hiểu mà còn nâng cao trải nghiệm của người dùng, đáp ứng tốt yêu cầu của cả bài toán học thuật và chương trình mô phỏng tương tác.

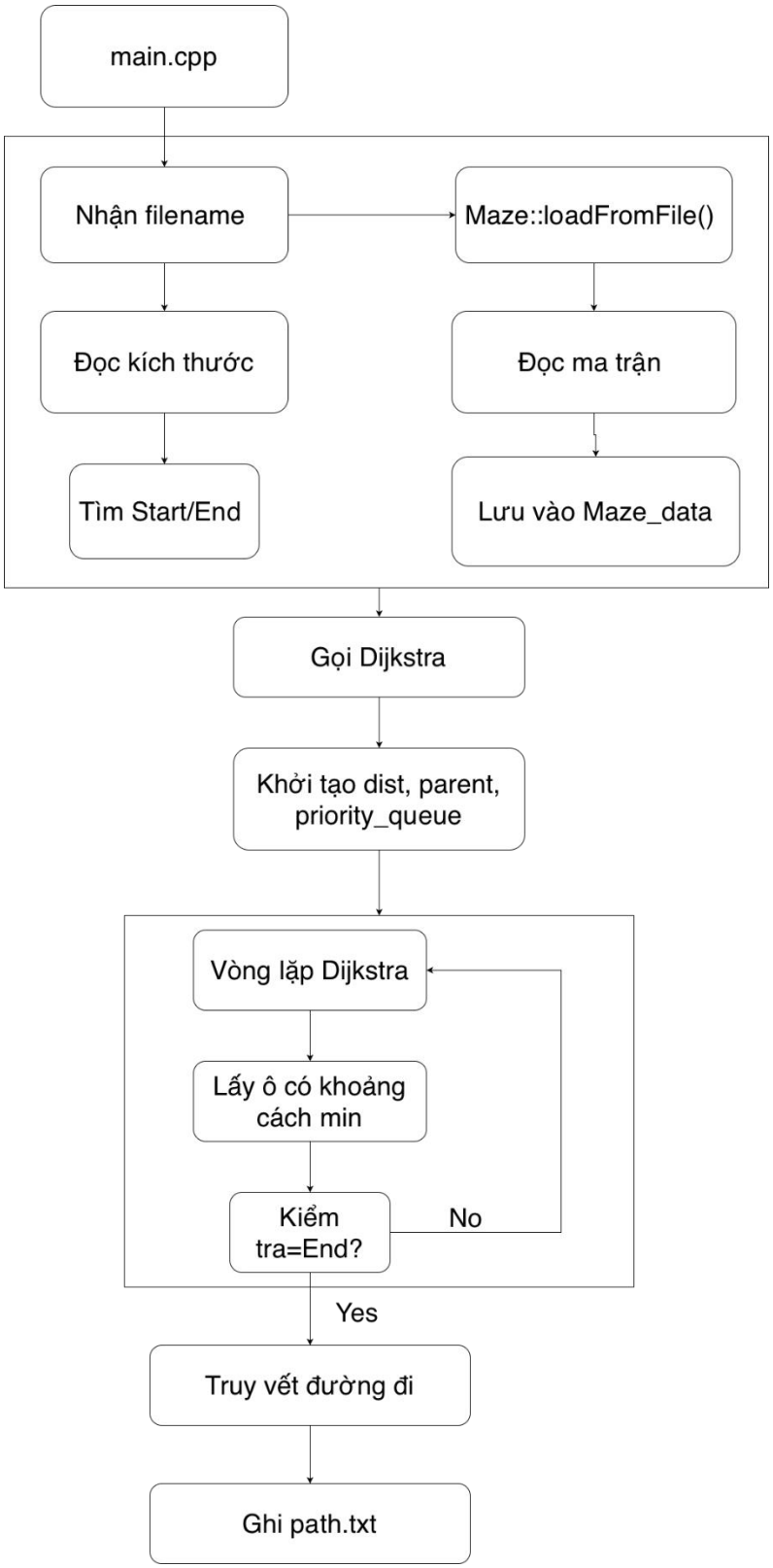
3.3. Sơ đồ luồng xử lý chương trình

Luồng xử lý tổng thể:



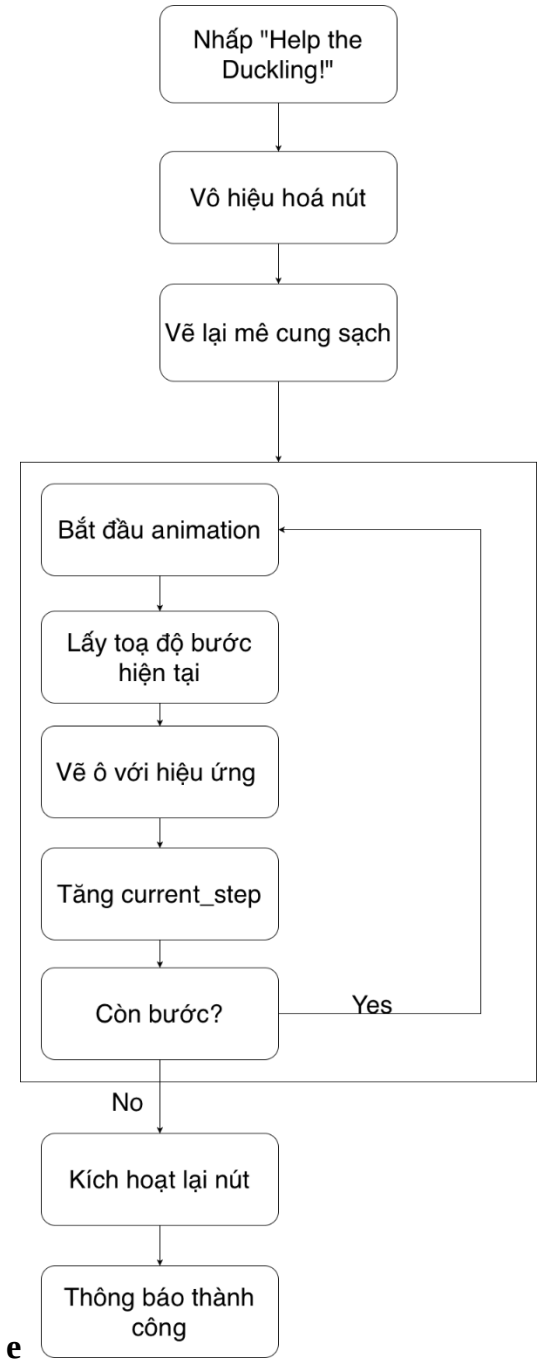
Hình 2. Luồng xử lý tổng thể

Luồng xử lý chi tiết trong C++:


main.cpp
Nhận filename
Maze::loadFromFile()
Đọc kích thước
Đọc ma trận
Tìm Start/End
Lưu vào Maze_data
Gọi Dijkstra
Khởi tạo dist, parent, priority_queue
Vòng lặp Dijkstra
Lấy ô có khoảng cách min
Kiểm tra=End?
No
Yes
Truy vết đường đi
Ghi path.txt

Hình 3. Luồng xử lý chi tiết trong C++

Luồng animation trong Python:



e

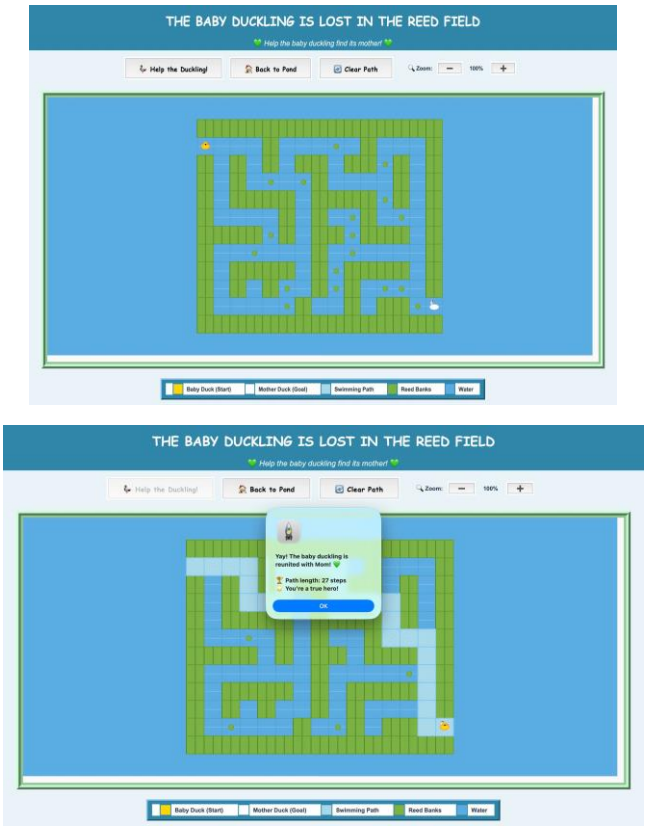
Hình 4. Luồng animation trong Python

Với cấu trúc module rõ ràng và các thành phần tách biệt, chương trình đảm bảo tính dễ bảo trì, mở rộng và debug. Giao diện đồ họa trực quan giúp người dùng dễ dàng thao tác và quan sát kết quả một cách sinh động.

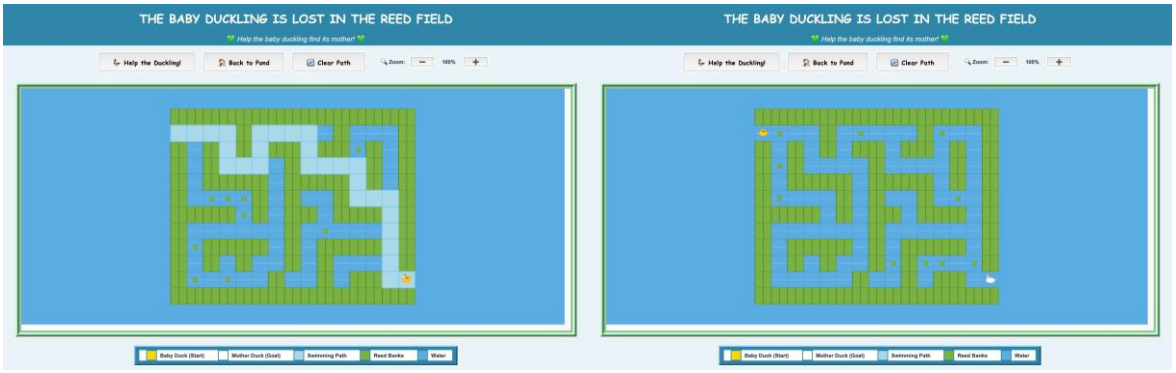
Chương 4. DEMO VÀ KẾT QUẢ THỰC NGHIỆM

4.1. Giao diện và minh họa kết quả

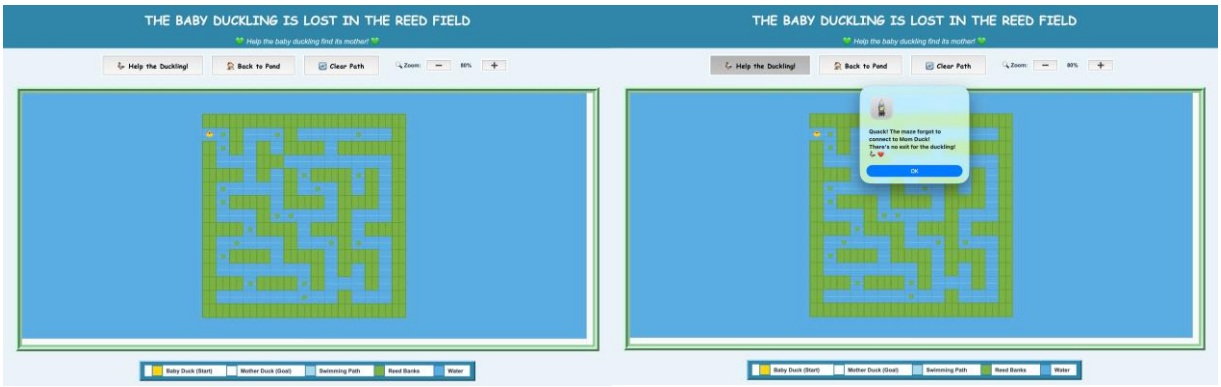
Hình 5. Màn hình chọn level



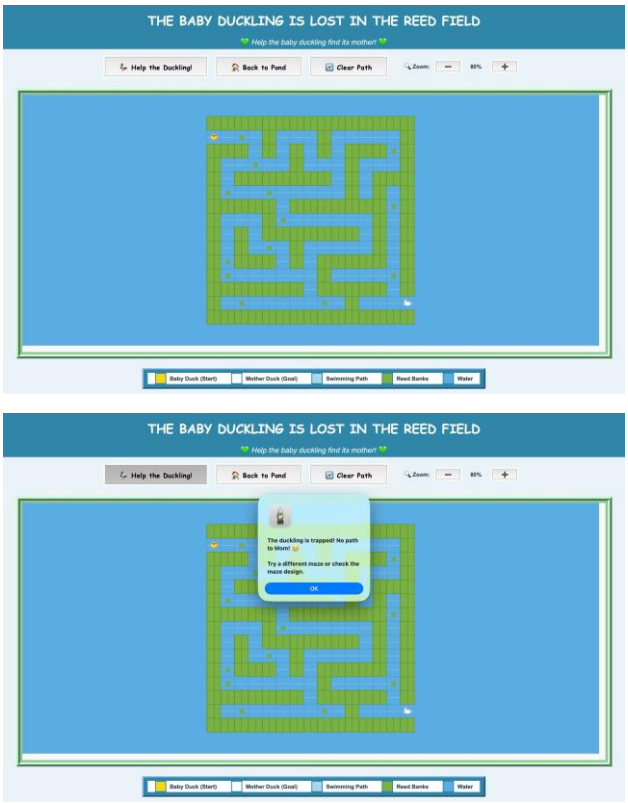
Hình 6. Giao diện mê cung level 1



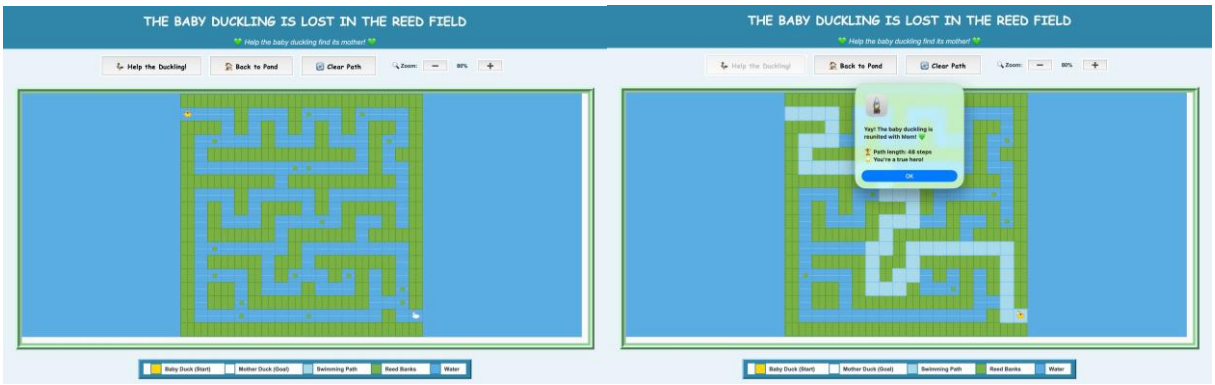
Hình 7. Bấm Clear Path



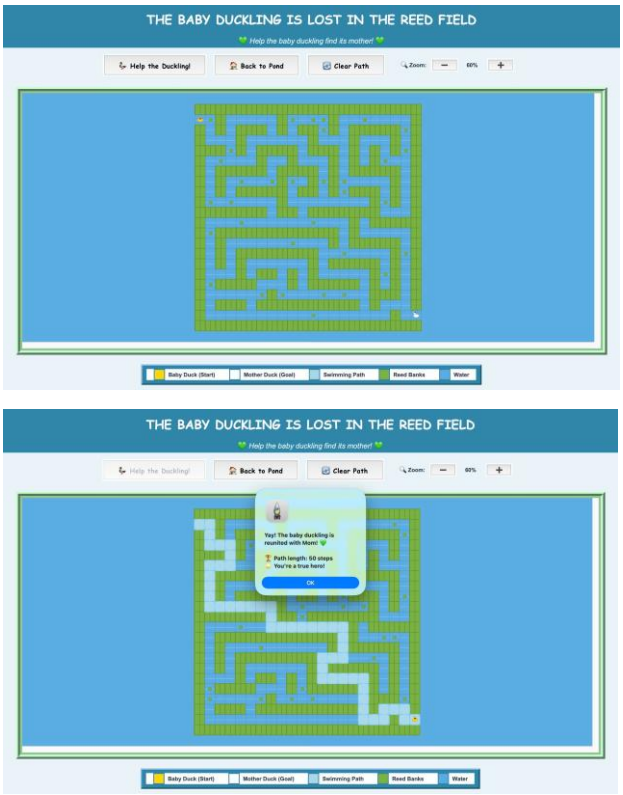
Hình 8. Giao diện mê cung level 2



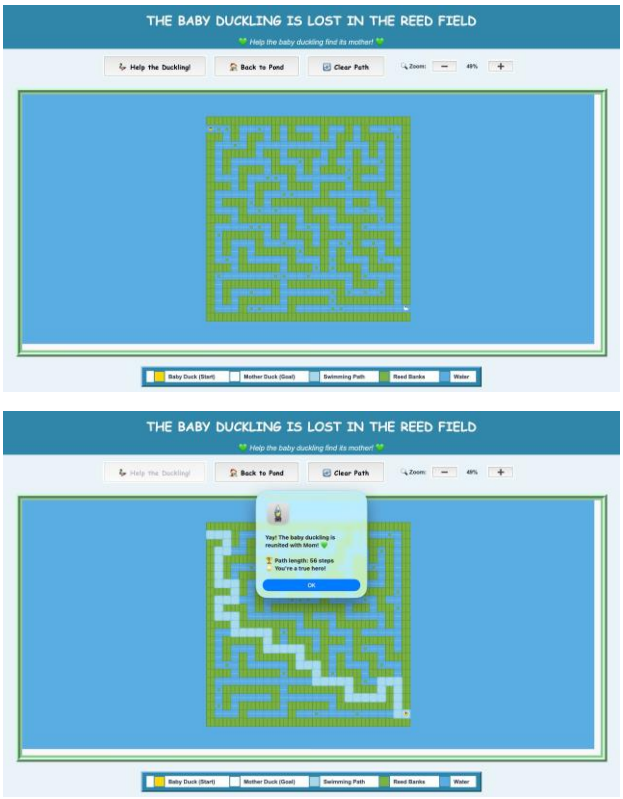
Hình 9. Giao diện mê cung level 3



Hình 10. Giao diện mê cung level 4



Hình 11. Giao diện mê cung level 5



Hình 12. Giao diện mê cung level 6

4.2. Các bộ kiểm thử (Test Case)

4.2.1. Maze đơn giản có đường đi

input1.txt

```
12 15
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 0 0 0 1 0 0 0 0 0 1 0 0 0 1
1 0 1 0 1 0 1 1 0 1 1 0 1 0 1
1 0 1 0 0 0 0 1 0 0 0 0 1 0 1
1 0 1 1 1 1 0 1 1 1 1 0 1 0 1
1 0 0 0 0 1 0 1 0 0 1 0 0 0 1
1 1 1 1 0 1 0 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 1
1 0 1 1 1 1 0 1 0 1 1 1 1 0 1
1 0 1 0 1 0 0 1 0 1 0 0 0 0 1
1 0 0 0 0 0 1 0 0 1 0 1 1 0 3
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

input4.txt

```
18 18
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1
1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1
1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1
1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1
1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1
1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 1 0 1
1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1
1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 0 1
1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 1 1
1 0 1 1 0 1 0 0 0 0 1 0 0 0 1 0 1 1
1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 3
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

input5.txt

```
22 22
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 1 0 1 1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 0 1
1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1
1 1 1 0 1 0 1 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1
1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 1 1
1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 1
1 0 1 0 1 1 1 1 1 0 1 1 0 1 0 1 1 1 0 1 0 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1
1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1
1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 1
1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1
1 0 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 1
1 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 0 1
1 0 0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1 0 1 0 1
1 0 1 1 1 1 0 0 0 1 0 1 1 1 1 1 0 1 0 1 0 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 0 1
1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 3
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

input6.txt

```
25 25
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 0 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 1
1 1 0 1 0 1 1 0 1 0 1 1 1 0 1 0 1 1 0 1 1 1 1 0 1
1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 1
1 0 1 1 1 1 0 1 1 0 1 1 1 1 0 1 1 0 1 0 1 0 1 0 1
1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 1
1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1
1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1
1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1
1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1
1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 1 0 1 1 0 1
1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1
1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 0 1
1 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1
1 0 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 1
1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1
1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1
1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1
1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1
1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 3
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Hình 13. Maze đơn giản có đường đi

4.2.2. Maze không có lối vào/ thoát

input2.txt

```
15 15
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 0 1 0 0 0 1 0 0 0 0 0 0 0 1
1 0 1 0 1 0 1 1 1 0 1 1 1 1 1
1 0 0 0 1 1 0 0 0 0 0 0 0 0 1
1 1 1 0 1 0 1 0 1 1 1 0 1 0 1
1 0 0 0 0 0 1 0 0 0 1 0 1 0 1
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1
1 0 0 0 1 0 0 0 0 0 1 0 0 0 1
1 1 1 0 1 1 1 0 1 1 1 0 1 1 1
1 0 0 0 0 0 1 0 0 0 1 0 0 0 1
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 0 1 0 1 0 1 0 1
1 1 1 0 1 1 1 0 1 0 0 0 1 0 1
1 0 0 0 0 0 0 0 0 0 1 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Hình 14. Maze không có lối vào/ thoát

4.2.3. Maze không có đường thoát

input3.txt

```
15 15
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 0 0 0 1 0 0 0 1 0 0 0 0 0 1
1 1 1 0 1 0 1 0 1 0 1 1 1 0 1
1 0 0 0 0 0 1 0 0 0 1 0 0 0 1
1 0 1 1 1 1 1 1 1 0 1 0 1 1 1
1 0 0 0 0 0 0 0 0 0 1 0 1 0 1
1 1 1 1 1 0 1 1 1 1 1 1 1 0 1
1 0 0 0 1 0 0 0 0 0 0 0 1 0 1
1 0 1 0 1 1 1 1 1 1 1 0 1 0 1
1 0 1 0 0 0 1 0 0 0 0 0 1 0 1
1 0 1 1 1 0 1 0 1 1 1 1 1 0 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 1
1 1 1 1 1 1 1 0 1 1 1 1 1 0 1
1 0 0 0 0 0 0 0 0 0 1 0 0 0 3
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Hình 15. Maze không đường

Chương 5. PHÂN TÍCH VÀ THẢO LUẬN

5.1. Ưu điểm và hạn chế của chương trình

5.1.1. Ưu điểm

Tính chính xác của thuật toán: Chương trình đã triển khai thành công thuật toán Dijkstra, đảm bảo tìm được đường đi ngắn nhất tuyệt đối từ điểm xuất phát đến lối thoát trong mô hình mê cung có trọng số không âm.

Hiệu quả xử lý: Việc sử dụng cấu trúc dữ liệu tối ưu là **Hàng đợi Ưu tiên (Priority Queue)** giúp giảm độ phức tạp thời gian của thuật toán xuống còn O((V + E).log(V)) (hoặc O(E log V)), đảm bảo hiệu suất tốt ngay cả với các mê cung có kích thước lớn.

Mô hình hóa rõ ràng: Đề tài đã thực hiện việc ánh xạ mê cung dạng lưới 2D thành đồ thị vô hướng có trọng số dương một cách chính xác, làm nền tảng cho việc áp dụng các thuật toán lý thuyết đồ thị.

Trực quan hóa hiệu quả: Giao diện người dùng được xây dựng bằng Python cung cấp khả năng trực quan hóa quá trình tìm đường đi. Điều này giúp người dùng và người học dễ dàng quan sát và hiểu rõ nguyên lý hoạt động của thuật toán Dijkstra qua từng bước di chuyển.

Cấu trúc module rõ ràng: Hệ thống được chia thành hai phần độc lập là Backend (C++) xử lý logic tính toán và Frontend (Python) xử lý giao diện, giúp chương trình dễ bảo trì, nâng cấp và gỡ lỗi.

Xử lý các trường hợp: Chương trình xử lý được cả hai trường hợp: tìm thấy đường đi ngắn nhất và thông báo khi không tồn tại lối thoát.

5.1.2. Hạn chế

Giới hạn của thuật toán: Chương trình chỉ sử dụng thuật toán Dijkstra và chưa triển khai so sánh hiệu suất thực nghiệm với các thuật toán tối ưu hơn cho bài toán tìm đường như A* (dù đã có so sánh lý thuyết).

Mô hình trọng số đơn giản: Hiện tại, mô hình mê cung chỉ xét trọng số đồng nhất (w(u,v) = 1) cho mọi bước di chuyển giữa hai ô liền kề, chưa mở rộng cho mê cung có trọng số khác nhau (chi phí di chuyển khác nhau).

Giới hạn về di chuyển: Chương trình chỉ cho phép di chuyển theo 4 hướng cơ bản (lên, xuống, trái, phải) và không xét di chuyển chéo.

Tính linh hoạt của đầu vào: Chương trình phụ thuộc vào việc đọc dữ liệu mê cung từ file .txt có định dạng số cố định.

5.2. Khó khăn trong quá trình triển khai

Thách thức trong việc tích hợp: Khó khăn lớn nhất là xây dựng luồng dữ liệu thông suốt và chính xác giữa hai ngôn ngữ lập trình khác nhau (C++ cho xử lý và Python cho hiển thị) thông qua cơ chế I/O file và gọi hệ thống (subprocess.run()).

Xử lý Mô hình hóa: Đảm bảo việc chuyển đổi chính xác từ ma trận ký tự trong file input sang ma trận số (đỉnh, tường) trong cấu trúc dữ liệu C++.

Trực quan hóa đường đi: Việc tạo ra animation để mô phỏng từng bước của thuật toán trên giao diện Python đòi hỏi sự đồng bộ giữa tốc độ xử lý của C++ và tốc độ vẽ của Python.

Kiểm thử các trường hợp đặc biệt: Việc tạo và kiểm tra đầy đủ các bộ dữ liệu (Test Case), bao gồm cả các trường hợp không có đường đi hoặc file input không đúng định dạng, đòi hỏi sự tỉ mỉ trong quá trình kiểm thử.

5.3. Hướng phát triển mở rộng

Triển khai và so sánh thuật toán A*: Tiến hành triển khai thuật toán A* (sử dụng Heuristic) và thực hiện các bài kiểm thử thực nghiệm để so sánh rõ ràng hiệu suất (thời gian chạy) giữa Dijkstra và A* trên các mê cung lớn.

Mô hình trọng số đa dạng: Mở rộng mô hình mê cung để gán trọng số khác nhau cho các ô, mô phỏng môi trường có chi phí di chuyển không đồng nhất (ví dụ: khu vực bùn lầy, khu vực băng giá,...).

Nâng cấp giao diện người dùng: Bổ sung chức năng cho phép người dùng tự tạo/vẽ mê cung (ví dụ: click chuột để thêm/bỏ tường, đặt Start/End) trực tiếp trên giao diện.

Mở rộng phạm vi di chuyển: Mở rộng khả năng tìm đường đi với 8 hướng di chuyển, bao gồm cả di chuyển chéo.

Chương 6. KẾT LUẬN

6.1. Tóm tắt nội dung và kết quả đạt được

Đề tài **"Triển khai thuật toán Dijkstra trong việc giải bài toán tìm lối thoát mê cung"** đã hoàn thành các mục tiêu đề ra.

Tóm tắt nội dung: Đề tài đã tập trung nghiên cứu lý thuyết đồ thị và thuật toán tìm đường đi ngắn nhất Dijkstra. Chúng tôi đã mô hình hóa mê cung dạng lưới 2D thành đồ thị có trọng số dương và xây dựng chương trình giải mê cung với kiến trúc module hóa rõ ràng (C++ cho xử lý logic, Python cho trực quan hóa).

Kết quả đạt được:

- Chương trình đã được cài đặt thành công, sử dụng thuật toán Dijkstra kết hợp với Hàng đợi Ưu tiên (Priority Queue) để đảm bảo tìm ra đường đi ngắn nhất với hiệu suất tối ưu.
- Chương trình thể hiện được kết quả trực quan, minh họa rõ ràng đường đi ngắn nhất tìm được dưới dạng hoạt ảnh (animation).
- Các bộ kiểm thử đã chứng minh tính đúng đắn và khả năng xử lý các trường hợp ngoại lệ (có đường đi, không có đường đi, lỗi đầu vào) của chương trình.

Đồ án đã giúp củng cố kiến thức nền tảng về Lý thuyết đồ thị, rèn luyện kỹ năng tư duy trừu tượng, mô hình hóa bài toán và áp dụng thuật toán vào giải quyết vấn đề thực tế.

6.2. Lời cảm ơn và đề nghị góp ý

Chúng em xin bày tỏ lòng biết ơn chân thành đến TS. Nguyễn Viết Hưng, giảng viên hướng dẫn đã tận tình chỉ bảo, hỗ trợ và cung cấp những góp ý quý báu trong suốt quá trình thực hiện đề tài.

Chúng em cũng xin cảm ơn Khoa Công nghệ Thông tin, Trường Đại học Sư phạm Thành phố Hồ Chí Minh đã tạo điều kiện để chúng em có thể nghiên cứu và hoàn thành tiểu luận này.

Do giới hạn về thời gian và kiến thức, chương trình không thể tránh khỏi những thiếu sót và hạn chế. Chúng em kính mong nhận được những ý kiến đóng góp, phê bình từ Thầy/Cô và Hội đồng chấm thi để đề tài có thể được cải thiện và phát triển mở rộng hơn trong tương lai.

THAM KHẢO

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, 1989 .

[2] John Adrian Bondy and Uppaluri Siva Ramachandra Murty, *Graph Theory with Applications*, Macmillan, 1976.

[3]VuSon_tk_8907, *thuat_toan_tim_duong_di_ngan_nhat_trong_ly_thuyet_do_thi_5688*, n.d.

[4] Rdsic, *Thuật Toán Giải Mê Cung: Hướng Dẫn Chi Tiết Từng Bước*, n.d.

[5] BaoNguyen113, *Bài tập tổng hợp về thuật toán đường đi ngắn nhất*, n.d.

[6] Đoàn Văn Thắng, *de-tai-cac-thuat-toan-tim-duong-di-ngan-nhat-trong-do-thi-6171*, n.d.

[7] Lê Minh Hoàng, *Chuyên đề lý thuyết đồ thị*, n.d.

[8] CodeGym Vietnam, *Mỗi ngày 1 thuật toán – Bài toán Vượt qua mê cung*, n.d.

[9] Dương Nguyễn Phú Cường, *Lab 1.5 – Chuyển danh sách kề sang ma trận kề*, 2023.