BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HỒ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO CUỐI KỲ
ĐỀ TÀI: GIẢI QUYẾT BÀI TOÁN TÌM ĐƯỜNG ĐI TRONG MÊ CUNG BẰNG CÁC THUẬT TOÁN TÌM KIẾM

Giảng viên hương dẫn:	Th.s Trần Tiến Đức
Sinh viên thực hiện:	Huỳnh Thanh Duy
MSSV:	22110118
Lớp:	22110CL1B
Khóa:	2022
Mã học phần:	241ARIN330585

Thành phố Hồ Chí Minh, tháng 12 năm 2024

DANH SÁCH THAM GIA ĐỀ TÀI

HỌC KỲ I, NĂM HỌC 2024 – 2025

Tên đề tài: Giải quyết bài toán tìm đường đi trong mê cung bằng các thuật toán tìm đường đi

Họ và tên sinh viên	Mã số sinh viên
Huỳnh Thanh Duy	22110118

Nhận xét của giảng viên

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Ngày 01, tháng 12, năm 2024

Giảng viên chấm điểm

MỤC LỤC

PHẦN MỞ ĐẦU **1**
1. LÝ DO CHỌN ĐỀ TÀI: 1
2. KẾT CẤU ĐỀ TÀI: 1
Chương 1: Các thuật toán tìm đường đi: **2**
1. Giới thiệu chung về các thuật toán tìm đường đi: 2
2. Thuật toán tìm kiếm không thông tin: 2
2.1. Thuật toán tìm kiếm theo chiều rộng (Breadth-First Search – BFS): 2
2.2. Thuật toán tìm kiếm theo chiều sâu (Depth-First Search - DFS): 3
3. Thuật toán tìm kiếm có thông tin: 4
3.1. Thuật toán A*: 4
3.2. Thuật toán tìm kiếm tham lam theo hướng dẫn tốt nhất (Greedy Best-First Search): 6
3.3. Thuật toán tìm kiếm chi phí đồng nhất (Uniform Cost Search - UCS): 6
Chương 2: Ứng dụng thuật toán vào bài toán: **7**
1. Tổng quan về chương trình: 7
2. Tạo mê cung ngẫu nhiên: 8
3. Triển khai thuật toán giải mê cung: 8
4. Xây dựng giao diện người dùng: 9
5. Thực nghiệm và đánh giá: 14

PHẦN MỞ ĐẦU

1. LÝ DO CHỌN ĐỀ TÀI:

Bài toán tìm đường đi trong mê cung là một bài toán yêu cầu xác định một lộ trình từ điểm xuất phát đến đích trong một không gian lưới hai chiều, nơi có các ô trống (đường đi được phép) và các ô cản (tường). Bài toán không chỉ đơn giản là tìm kiếm một đường đi, mà còn có thể yêu cầu tối ưu hóa lộ trình, chẳng hạn như tìm đường ngắn nhất hoặc tiết kiệm chi phí nhất. Đây chính là lý do bài toán luôn thu hút sự chú ý trong cả giảng dạy và nghiên cứu.

Để giải quyết bài toán này, các thuật toán tìm kiếm đường đi đóng vai trò quan trọng. Các thuật toán có thể được phân loại thành hai nhóm chính: **thuật toán tìm kiếm không có thông tin** và **thuật toán tìm kiếm có thông tin**. Trong nhóm đầu tiên, các thuật toán như BFS (Breadth-First Search) và DFS (Depth-First Search) được sử dụng rộng rãi. BFS hoạt động theo nguyên tắc mở rộng đồng đều tất cả các trạng thái, đảm bảo tìm được đường đi ngắn nhất nếu tồn tại. Tuy nhiên, nó đòi hỏi nhiều bộ nhớ khi giải các bài toán lớn. Trong khi đó, DFS tìm kiếm theo chiều sâu, tiết kiệm bộ nhớ hơn nhưng không đảm bảo kết quả tối ưu. Với nhóm thuật toán tìm kiếm có thông tin, các phương pháp như A*, Greedy Best-First Search và Uniform Cost Search (UCS) mang lại sự linh hoạt và hiệu quả hơn. A* kết hợp giữa chi phí thực tế và ước lượng chi phí còn lại, đảm bảo tìm được đường đi ngắn nhất trong không gian có trọng số. Greedy Best-First Search tập trung vào trạng thái gần đích nhất, giúp tìm kiếm nhanh hơn nhưng có thể không tìm được đường tối ưu. UCS, một biến thể của BFS, mở rộng trạng thái có chi phí thấp nhất trước, rất phù hợp cho các bài toán với trọng số khác nhau.

Việc lựa chọn đề tài này không chỉ là một cơ hội để tôi rèn luyện tư duy thuật toán và kỹ năng lập trình mà còn giúp tôi học cách giải quyết các bài toán thực tế từ góc độ khoa học và kỹ thuật. Qua việc nghiên cứu, tôi nhận ra rằng một bài toán tưởng chừng đơn giản như mê cung lại chứa đựng rất nhiều bài học bổ ích về cấu trúc dữ liệu, thuật toán và khả năng tối ưu hóa. Đây không chỉ là một bài tập lý thuyết mà còn là cầu nối giúp tôi áp dụng kiến thức vào các vấn đề lớn hơn trong tương lai như trí tuệ nhân tạo, điều hướng tự động và tối ưu hóa hệ thống. Chính vì vậy, tôi tin rằng việc nghiên cứu bài toán này là một lựa chọn đúng đắn và đầy ý nghĩa đối với hành trình học tập và phát triển của tôi trong lĩnh vực công nghệ thông tin.

2. KẾT CẤU ĐỀ TÀI:

Chương 1: Các thuật toán tìm kiếm đường đi.
Chương 2: Ứng dụng thuật toán vào bài toán.

PHẦN NỘI DUNG

Chương 1: Các thuật toán tìm đường đi:

1. Giới thiệu chung về các thuật toán tìm đường đi:

Bài toán tìm đường đi trong mê cung đặt ra yêu cầu tìm một lộ trình từ **điểm bắt đầu** (**start point**) đến **điểm kết thúc** (**goal point**) trong một môi trường có **các ô trống** (**đường đi được phép**) và **ô cản** (**tường**). Điểm hấp dẫn của bài toán này không chỉ nằm ở việc xác định một đường đi bất kỳ mà còn ở khả năng tối ưu hóa lộ trình theo các tiêu chí như ngắn nhất hoặc chi phí thấp nhất.

Để giải quyết bài toán này, các thuật toán tìm kiếm được xem là công cụ cốt lõi. Chúng đóng vai trò duyệt qua không gian trạng thái, nơi mỗi trạng thái là một vị trí trong mê cung, để tìm ra lộ trình đáp ứng yêu cầu. Các thuật toán tìm kiếm được phân thành hai nhóm chính:

Thuật toán tìm kiếm không thông tin (Uninformed Search): Không sử dụng bất kỳ dữ liệu bổ sung nào về cấu trúc của mê cung hay vị trí của đích. Chúng thực hiện việc duyệt qua các trạng thái một cách toàn diện, dựa trên logic duyệt trạng thái mà không "đoán trước" hướng đi.

Thuật toán tìm kiếm có thông tin (Informed Search): Sử dụng thông tin bổ sung thông qua hàm heuristic để hướng dẫn quá trình tìm kiếm. Điều này giúp các thuật toán này hiệu quả hơn, đặc biệt trong các bài toán lớn.
Việc lựa chọn thuật toán phù hợp không chỉ phụ thuộc vào yêu cầu của bài toán mà còn vào cấu trúc của mê cung, như kích thước, độ phức tạp, và tiêu chí tối ưu hóa. Trong phần tiếp theo, chúng ta sẽ phân tích chi tiết cách các thuật toán hoạt động và áp dụng vào bài toán.

2. Thuật toán tìm kiếm không thông tin:

2.1. Thuật toán tìm kiếm theo chiều rộng (Breadth-First Search – BFS):

Nguyên tắc hoạt động:

BFS hoạt động dựa trên nguyên tắc **mở rộng theo chiều rộng**, nghĩa là duyệt qua tất cả các trạng thái có cùng khoảng cách từ điểm bắt đầu trước khi duyệt các trạng thái xa hơn. Điều này đảm bảo rằng BFS sẽ tìm thấy đường đi ngắn nhất (nếu tồn tại), nhưng cũng yêu cầu nhiều bộ nhớ để lưu trữ các trạng thái đã mở rộng.

BFS sử dụng hàng đợi (queue) để lưu trữ các trạng thái đang chờ được xử lý.

Hàng đợi hoạt động theo cơ chế “First in first out” (FIFO).

Lộ trình duyệt trạng thái:
BFS tìm kiếm theo chiến lược "duyệt toàn bộ". Thuật toán bắt đầu từ điểm xuất phát và duyệt tất cả các ô xung quanh trước khi tiến sang các ô xa hơn. Điều này được thực hiện bằng cách sử dụng **hàng đợi (queue)** để lưu trữ các trạng thái cần kiểm tra:

1. Ban đầu, trạng thái điểm bắt đầu được đưa vào hàng đợi.
2. Lấy trạng thái đầu tiên từ hàng đợi và kiểm tra xem đó có phải là đích không.
3. Nếu không, tất cả các trạng thái liền kề hợp lệ (ô trống) của trạng thái hiện tại được đưa vào cuối hàng đợi.
4. Lặp lại quá trình cho đến khi hàng đợi rỗng hoặc đích được tìm thấy.

⇨ BFS duyệt tất cả các trạng thái theo thứ tự gần đến xa, vì vậy nó đảm bảo tìm được đường đi ngắn nhất trong mê cung với chi phí bằng nhau.

Đánh giá:

Ưu điểm	Nhược điểm
1. Hiệu quả trong các mê cung đơn giản và có kích thước nhỏ. 2. Đảm bảo tìm được đường đi ngắn nhất.	1. Tốn bộ nhớ, vì phải lưu trữ toàn bộ các trạng thái đã duyệt trong hàng đợi. 2. Chậm khi không gian trạng thái lớn hoặc mê cung có nhiều ngõ cụt.

2.2. Thuật toán tìm kiếm theo chiều sâu (Depth-First Search - DFS):

Nguyên tắc hoạt động:
DFS hoạt động dựa trên nguyên tắc **mở rộng theo chiều sâu**, nghĩa là đi sâu vào một nhánh của không gian tìm kiếm trước khi quay lại và khám phá các nhánh khác. DFS sử dụng **ngăn xếp (stack)** để lưu trữ các trạng thái đang chờ xử lý, theo cơ chế "Last in first out" (LIFO).

Chiến lược khám phá chiều sâu:
DFS tập trung vào chiến lược **"đào sâu"**. Từ điểm xuất phát, thuật toán duyệt theo một nhánh sâu nhất trước khi quay lại để kiểm tra các nhánh khác. Thuật toán sử dụng **ngăn xếp (stack)** để quản lý các trạng thái:

1. Trạng thái điểm bắt đầu được đẩy vào ngăn xếp.
2. Lấy trạng thái trên đỉnh ngăn xếp và kiểm tra xem đó có phải là đích không.

3. Nếu không, các trạng thái lân cận hợp lệ của trạng thái hiện tại được đẩy vào ngăn xếp.
4. Tiếp tục cho đến khi ngăn xếp rỗng hoặc đích được tìm thấy.

⇨ DFS duyệt sâu nhất có thể trước khi quay lại các trạng thái khác, tiết kiệm bộ nhớ hơn so với BFS.

Đánh giá:

Ưu điểm	Nhược điểm
1. Tiêu tốn ít bộ nhớ hơn, phù hợp với không gian trạng thái lớn. 2. Dễ triển khai và hoạt động hiệu quả trong các mê cung nhỏ.	1. Không đảm bảo tìm được đường đi ngắn nhất. 2. Có thể rơi vào vòng lặp hoặc ngõ cụt nếu không xử lý tốt các trạng thái đã duyệt.

3. Thuật toán tìm kiếm có thông tin:

3.1. Thuật toán A*:

❖ **Nguyên tắc hoạt động:**

A* hoạt động dựa trên việc **đánh giá trạng thái** thông qua một hàm tổng hợp chi phí. Mỗi trạng thái được đánh giá bằng một hàm $f(n)$, trong đó:

$$\boldsymbol{f(n) = g(n) + h(n)}$$

$\boldsymbol{f(n)}$: Tổng chi phí ước lượng để đi từ điểm bắt đầu đến đích qua trạng thái n.

$\boldsymbol{g(n)}$: Chi phí thực tế từ điểm bắt đầu đến trạng thái hiện tại nnn.

$\boldsymbol{h(n)}$: Ước lượng chi phí từ trạng thái hiện tại nnn đến đích (goal point).

⇨ Trạng thái có giá trị $\boldsymbol{f(n)}$ nhỏ nhất sẽ được mở rộng trước.

❖ **Quy trình định hướng tìm kiếm:**

A* là một thuật toán tìm kiếm có định hướng, sử dụng cả chi phí thực tế và ước lượng chi phí còn lại để dẫn dắt quá trình tìm kiếm đến đích một cách hiệu quả. Quy trình này được thực hiện thông qua hàm đánh giá thông qua hàm $g(n)$ được định nghĩa ở trên.

Bước 1: Khởi tạo

Đưa điểm bắt đầu (start point) vào **hàng đợi ưu tiên (priority queue)**, với giá trị $\boldsymbol{f(n) = g(n) + h(n)}$**.**

Đặt $\boldsymbol{g(start) = 0}$ (chi phí từ điểm bắt đầu đến chính nó là 0).

Bước 2: Mở rộng trạng thái

Lấy trạng thái nnn có giá trị $\boldsymbol{f(n)}$ nhỏ nhất từ hàng đợi ra để xử lý.
Nếu trạng thái này là đích (goal point), dừng thuật toán và truy ngược lại đường đi từ điểm bắt đầu đến đích.
Nếu chưa, tiếp tục mở rộng trạng thái n:
Với mỗi trạng thái lân cận m, tính toán:
$\boldsymbol{g(m)} = \boldsymbol{g(n)} +$ chi phí di chuyển từ m đến n
$\boldsymbol{h(m)} =$ hàm heuristic ước lượng đến đích
$\boldsymbol{f(m)} = \boldsymbol{g(m)} + \boldsymbol{h(m)}$
Nếu trạng thái mmm chưa từng được xét hoặc giá trị $\boldsymbol{g(m)}$ mới *nhỏ hơn giá trị cũ,* cập nhật $\boldsymbol{f(m)}$ và thêm $\boldsymbol{m}$ vào hàng đợi ưu tiên.

Bước 3: Lặp lại

Lặp lại quá trình trên cho đến khi tìm thấy đích hoặc hàng đợi trống:
Nếu tìm thấy đích, thuật toán kết thúc và trả về đường đi ngắn nhất.
Nếu hàng đợi trống mà không tìm thấy đích, kết luận rằng không có đường đi từ điểm bắt đầu đến đích.

❖ **Đánh giá:**

⇨ **Kết hợp giữa thực tế và dự đoán**:
- $\boldsymbol{g(n)}$giúp đảm bảo rằng chi phí thực tế được tính toán chính xác.
- $\boldsymbol{h(n)}$cung cấp một hướng dẫn trực quan, giảm thiểu thời gian duyệt các trạng thái không cần thiết.

⇨ **Hàng đợi ưu tiên**: A* sử dụng hàng đợi ưu tiên để luôn mở rộng trạng thái có giá trị $\boldsymbol{f(n)}$ nhỏ nhất trước, giúp tập trung vào những trạng thái có triển vọng đạt được đích nhanh nhất.

⇨ **Khả năng tối ưu**: Nếu hàm $\boldsymbol{h(n)}$là **admissible** (không đánh giá vượt quá chi phí thực tế) và **consistent** (tuân thủ bất đẳng thức tam giác), A* đảm bảo tìm được đường đi ngắn nhất.

Ưu điểm	**Nhược điểm**
1. Tìm được đường đi tối ưu (nếu $h(n)$ phù hợp). 2. Hiệu quả hơn các thuật toán như BFS trong không gian tìm kiếm lớn nhờ khả năng định hướng.	3. Cần nhiều bộ nhớ để lưu trữ các trạng thái đã mở rộng. 4. Hiệu quả phụ thuộc mạnh mẽ vào chất lượng của hàm heuristic.

3.2. Thuật toán tìm kiếm tham lam theo hướng dẫn tốt nhất (Greedy Best-First Search):

❖ **Nguyên tắc hoạt động:**

Greedy Best-First Search mở rộng trạng thái dựa trên giá trị ước lượng chi phí $\boldsymbol{h(n)}$ (chỉ tập trung vào khoảng cách từ trạng thái hiện tại đến đích). Nó ưu tiên các trạng thái có $\boldsymbol{h(n)}$ nhỏ nhất mà không quan tâm đến chi phí thực tế $\boldsymbol{g(n)}$.

❖ **Cơ chế tìm kiếm hướng mục tiêu:**

Thuật toán ưu tiên mở rộng trạng thái gần đích nhất, chỉ sử dụng ước lượng chi phí $\boldsymbol{h(n)}$.

Giống A*, Greedy cũng sử dụng hàng đợi ưu tiên, nhưng không quan tâm đến chi phí thực tế $\boldsymbol{g(n)}$.

1. Tính $\boldsymbol{h(n)}$ cho trạng thái ban đầu và thêm vào hàng đợi ưu tiên.
2. Lấy trạng thái có $\boldsymbol{h(n)}$ nhỏ nhất từ hàng đợi ra để xử lý:
   a) Nếu đó là trạng thái đích, thuật toán kết thúc.
   b) Nếu không, các trạng thái lân cận được tính $\boldsymbol{h(n)}$ và thêm vào hàng đợi.
3. Tiếp tục lặp lại cho đến khi tìm thấy đích hoặc hàng đợi rỗng.

❖ **Đánh giá:**

Ưu điểm	**Nhược điểm**
Tìm kiếm nhanh, đặc biệt trong mê cung lớn.	Không đảm bảo tìm được đường đi ngắn nhất.

3.3. Thuật toán tìm kiếm chi phí đồng nhất (Uniform Cost Search - UCS):

❖ **Nguyên tắc hoạt động:**

UCS chỉ dựa trên chi phí thực tế $\boldsymbol{g(n)}$ để quyết định trạng thái nào sẽ được mở rộng tiếp theo. Nó mở rộng trạng thái có $\boldsymbol{g(n)}$ nhỏ nhất, đảm bảo rằng trạng thái được mở rộng là trạng thái rẻ nhất tại thời điểm đó.

❖ **Tiến trình tìm kiếm dựa trên chi phí thực tế:**

UCS tập trung vào mở rộng trạng thái có chi phí thực tế thấp nhất $\boldsymbol{g(n)}$ bỏ qua ước lượng $\boldsymbol{g(n)}$

Dùng hàng đợi ưu tiên để quản lý các trạng thái.

1. Thêm trạng thái ban đầu vào hàng đợi ưu tiên với
2. $\boldsymbol{g(n) = 0}$.

3. Lấy trạng thái có $\boldsymbol{g(n)}$ nhỏ nhất từ hàng đợi ra để xử lý:
    a. Nếu đó là trạng thái đích, thuật toán kết thúc.
    b. Nếu không, các trạng thái lân cận được tính toán lại giá trị $\boldsymbol{g(n)}$ và thêm vào hàng đợi.
4. Tiếp tục lặp lại cho đến khi tìm thấy đích hoặc hàng đợi rỗng.

❖ **Đánh giá:**

Ưu điểm	**Nhược điểm**
Đảm bảo tìm được đường đi tối ưu nếu tất cả chi phí là dương.	Chậm hơn A* khi không sử dụng heuristic.

Chương 2: Ứng dụng thuật toán vào bài toán:

1. Tổng quan về chương trình:

❖ **Mục tiêu của chương trình:**

Chương trình được xây dựng nhằm giải quyết bài toán tìm đường đi trong mê cung bằng cách áp dụng các thuật toán tìm kiếm như **BFS, DFS, A*, Greey, UCS**.

Chương trình không những hướng đến mục tiêu hiểu rõ về **nguyên tắ hoạt động** của các thuật toán mà còn hướng đến việc ứng dụng các thuật toán tìm đường vào một bài toán thực tế cụ thể. Đồng thời giúp đưa ra đánh giá khách quan về hiệu suất tìm đường của các thuật toán đối với các trường hợp cụ thể

❖ **Các tính năng của chương trình:**

Tạo mê cung ngẫu nhiên với kích thước truyền vào có thể thay đổi nhưng để đảm bảo trực qua hóa nên chương trình vẫn gắn kích thước cố định cho mê cung là một ma trận có kích thước 50 * 20 ô.

Triển khai các thuật toán để tìm đường đi : Chương trình cho phép người dùng lựa chọn 1 trong 5 thuật toán đã nêu ở phần 1 để tìm đường đi trong mê cung.

Hiển thị kết quả tìm được là đường đi ngắn nhất và quá trình loang tìm đường của các thuật toán. Đồng thời đưa ra chiều dài đường đi và tổng số bước di chuyển để tiện cho việc so sánh hiệu suất giữa các thuật toán.

❖ **Các thư viện và công cụ sử dụng:**

Ngôn ngữ lập trình: Python.

Thư viện hỗ trợ:

simple.search của simpleai: cung cấp các thuật toán tìm kiếm (BFS, DFS, A*, Greedy, Uniform Cost).

Tkinter: Xây dựng giao diện người dùng trên desktop.

Streamlit: Xây dựng giao diện người dùng trên trình duyệt web.
PIL, random: Hỗ trợ vẽ mê cung và xử lý hình ảnh.

2. **Tạo mê cung ngẫu nhiên:**
Nguyên tắc tạo mê cung: Sử dụng thuật toán **Depth-First Search (DFS)** áp dụng kĩ thuật đệ quy (**Backtracking**) để sinh mê cung ngẫu nhiên.

Mê cung được biểu diễn dưới dạng ma trận 2D, trong đó:
"#": Tường (không thể đi qua).
" ": Đường đi.
"o": Điểm bắt đầu.
"x": Điểm kết thúc.

Quá trình tạo mê cung:
- **a.** Khởi tạo mê cung dưới dạng một ma trận toàn bộ là tường.
- **b.** Chọn một ô ngẫu nhiên làm điểm bắt đầu, sau đó mở rộng đường đi bằng cách duyệt DFS:
  - ⇨ Tạo các ngã rẽ và kết nối các ô xung quanh để tạo đường đi.
  - ⇨ Tránh các vòng lặp và đảm bảo đường đi không quay lại điểm đã duyệt.
- **c.** Kiểm tra độ phức tạp của mê cung:
  - ⇨ Xác định số lượng đường đi tối thiểu.
  - ⇨ Kiểm tra kết nối giữa điểm bắt đầu và điểm kết thúc.

Triển khai trong mã nguồn:

Chức năng chính: ***generate_map_with_paths()*** trong file ***random_map_generator.py***.

Hàm sử dụng phương pháp lặp đệ quy để mở rộng các đường đi và tạo các ngã rẽ trong mê cung.

3. **Triển khai thuật toán giải mê cung:**
Mục tiêu: Áp dụng các thuật toán tìm kiếm từ chương 1 (BFS, DFS, A*, Greedy, Uniform Cost) để tìm đường từ điểm bắt đầu đến điểm kết thúc trong mê cung.

Các bước triển khai:
1. Nhập mê cung được tạo từ module tạo mê cung.
2. Xác định vị trí điểm bắt đầu và điểm kết thúc.
3. Người dùng chọn thuật toán tìm kiếm:
   - **a. BFS (Tìm kiếm theo chiều rộng):** Mở rộng các trạng thái gần nhất trước để tìm đường ngắn nhất.
   - **b. DFS (Tìm kiếm theo chiều sâu):** Ưu tiên mở rộng sâu vào một

nhánh trước khi quay lại các nhánh khác.

c. **A*:** Sử dụng hàm heuristic để hướng dẫn tìm kiếm, đảm bảo tìm đường tối ưu.
d. **Greedy (Tìm kiếm tham lam):** Mở rộng trạng thái gần đích nhất theo hàm ước lượng $h(n)$.
e. **Uniform Cost (Tìm kiếm chi phí đồng nhất):** Tập trung vào chi phí thực tế từ điểm bắt đầu đến trạng thái hiện tại.

Kết quả:

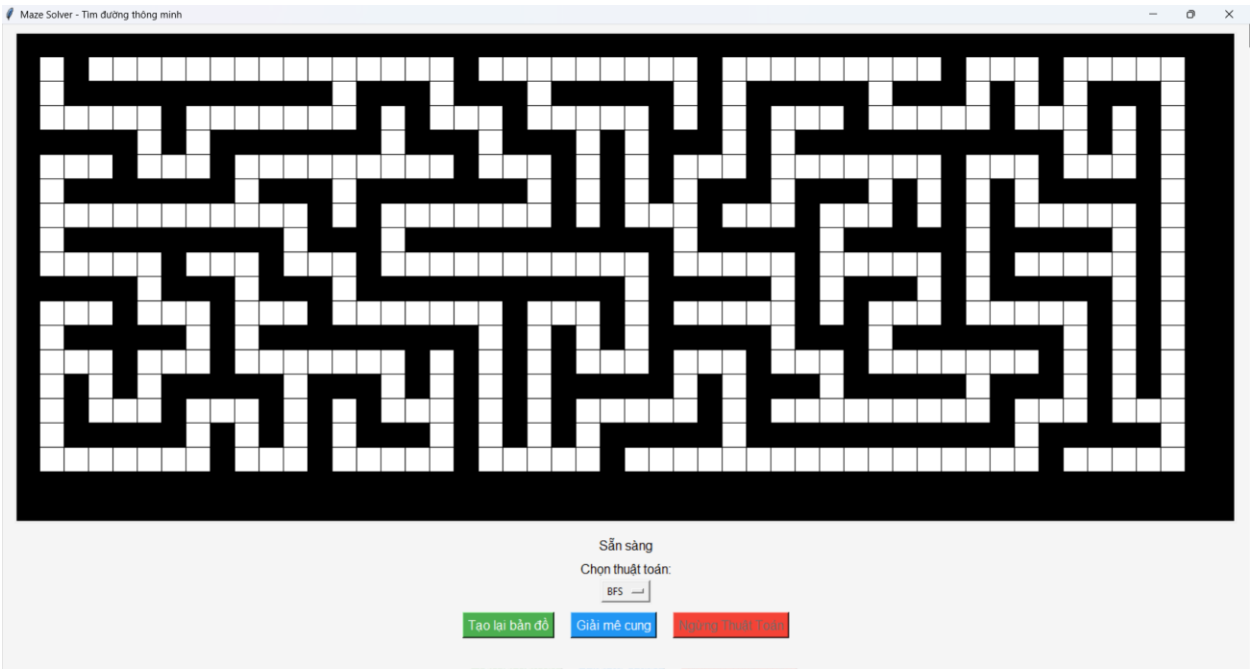
Hiển thị đường đi tìm được trên mê cung.

Tính toán và hiển thị các chỉ số như thời gian thực thi, độ dài đường đi.

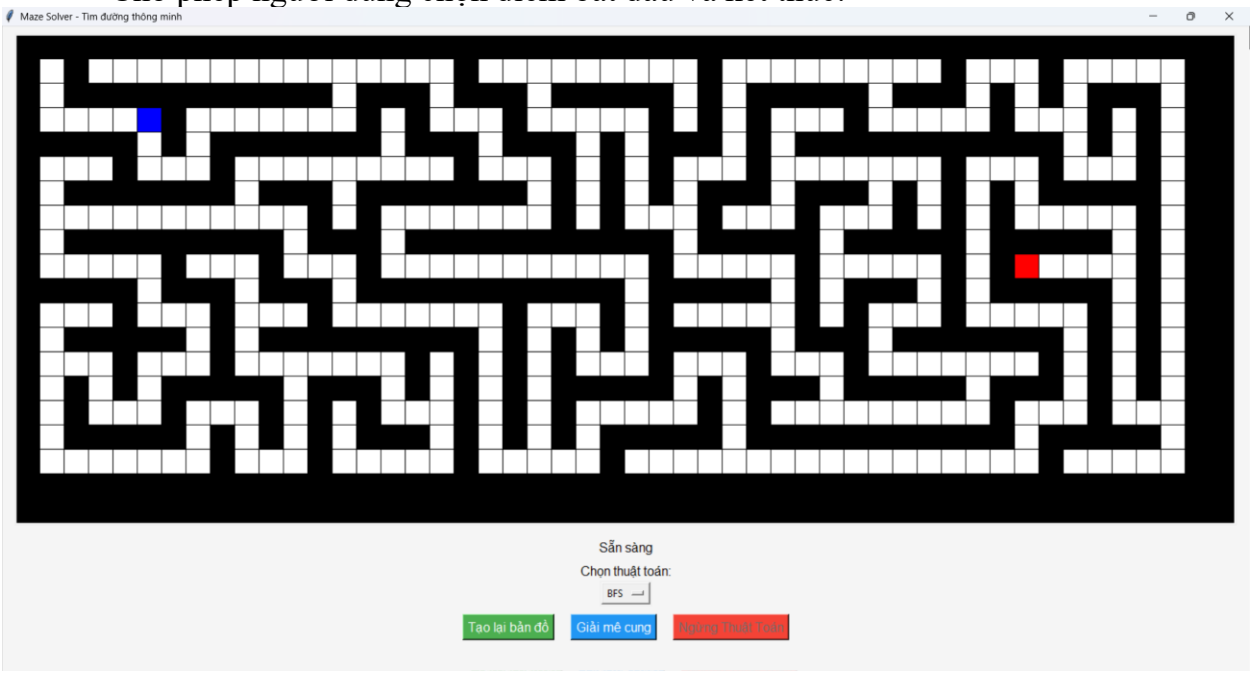
4. Xây dựng giao diện người dùng:

Giao diện Tkinter:

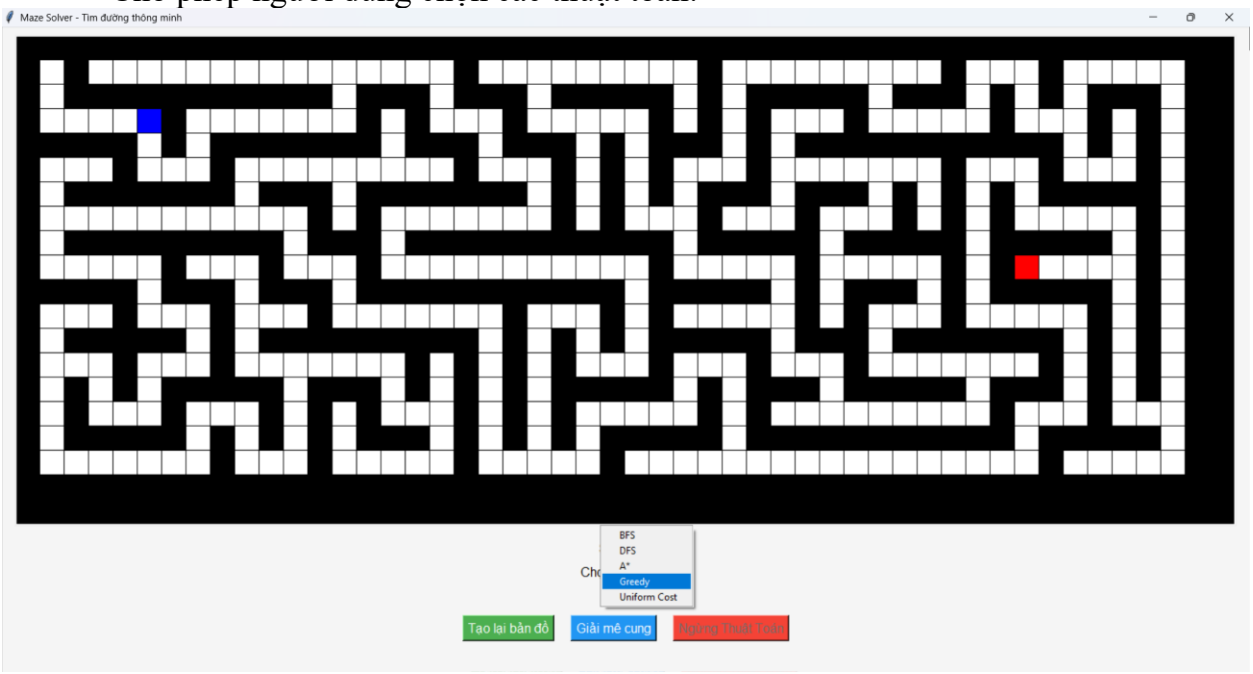
Hiển thị mê cung trên canvas:



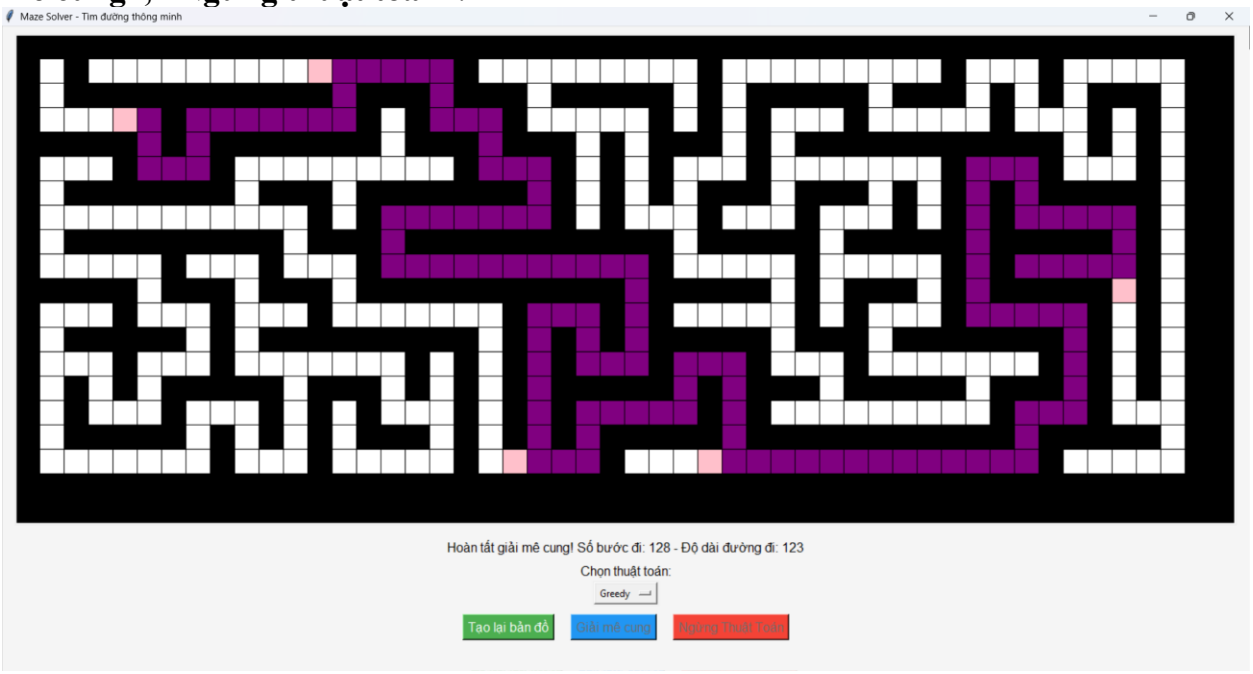
Cho phép người dùng chọn điểm bắt đầu và kết thúc:



Cho phép người dùng chọn các thuật toán:



Cho phép người dùng thao tác với các nút **“Tạo lại bản đồ”, “Giải mê cung”, “Ngưng thuật toán”**:



⇨ **“Ngưng thuật toán”:** Ngưng chạy qua trình loang tìm đường đi mà hiện hằn đường đi cuối cùng và kết thúc thuật toán.

Triển khai ở file tkinter_maze.py:

Chức năng **start_tkinter_gui()** khởi tạo giao diện.

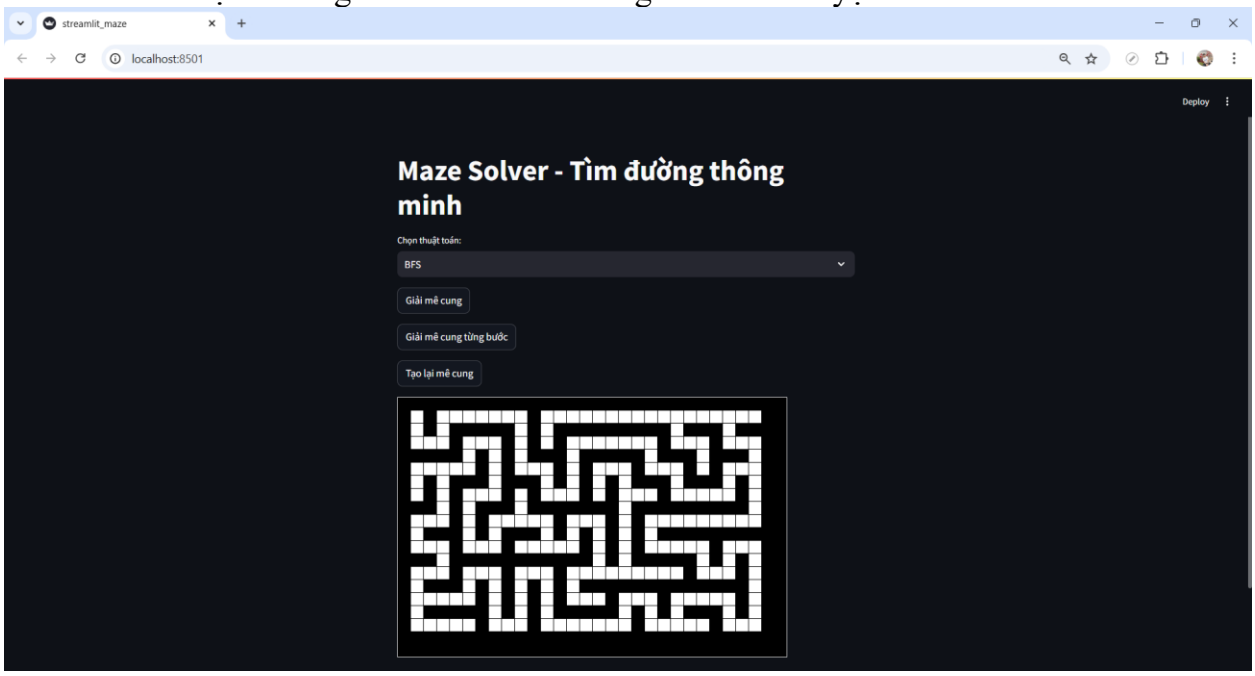
Canvas được sử dụng để vẽ mê cung và hiển thị kết quả đường đi.

Các sự kiện click chuột để chọn điểm bắt đầu/kết thúc.

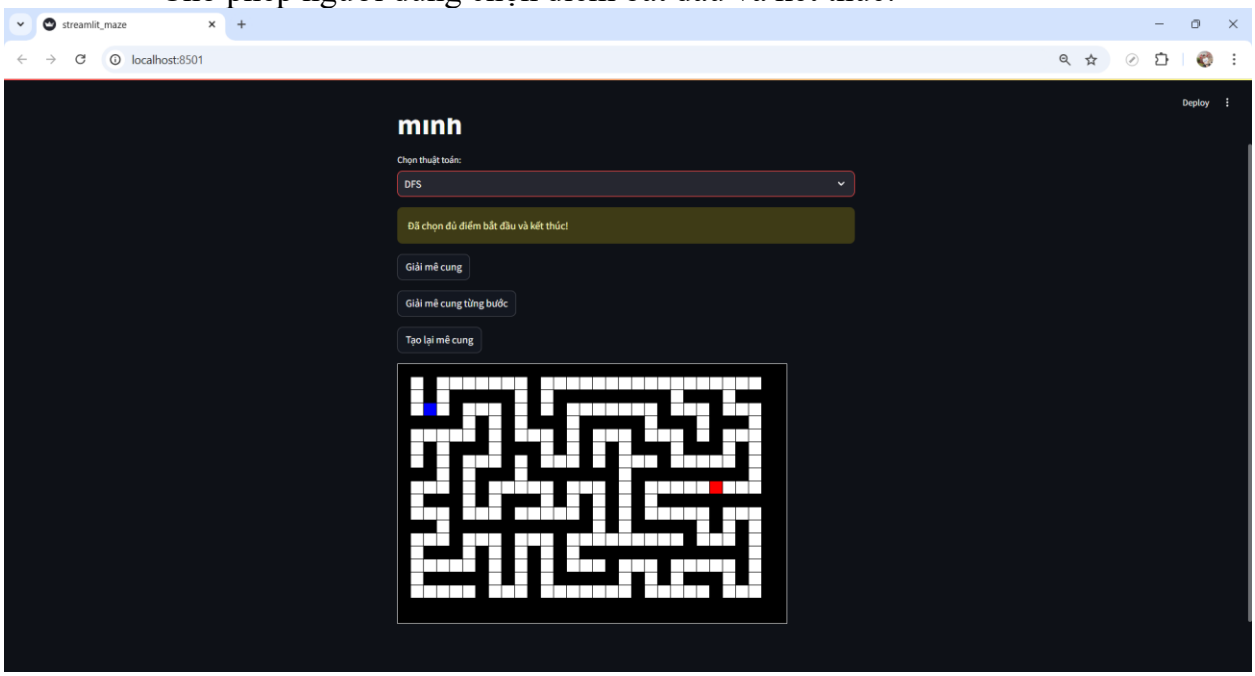
Giao diện Streamlit:

Tương tự với Tkinter nhưng mà trên trình duyệt web:

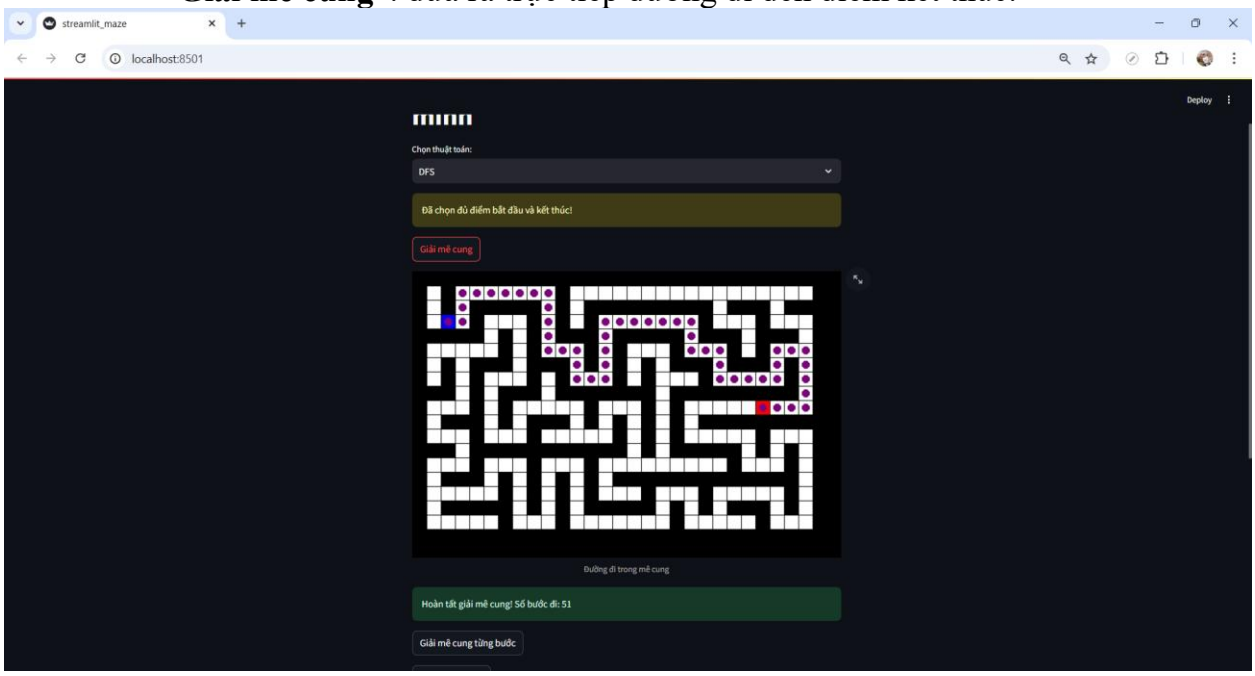
Hiển thị mê cung và các nút chức năng trên trình duyệt web:



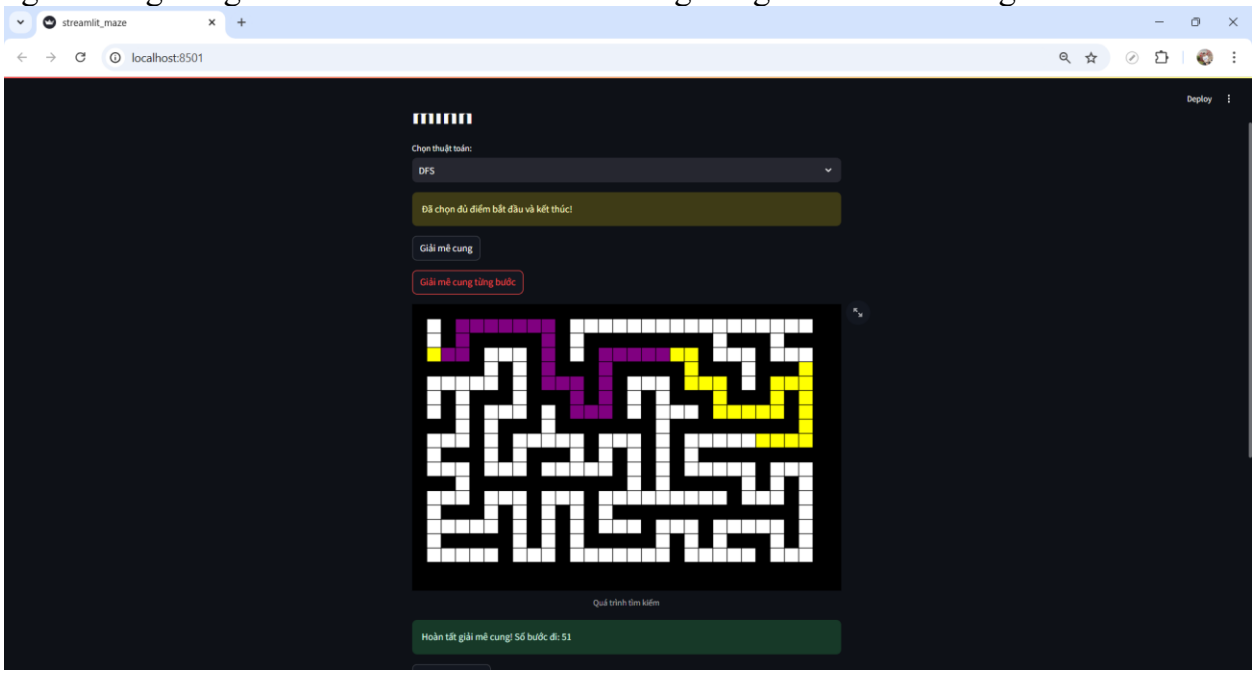
Cho phép người dùng chọn điểm bắt đầu và kết thúc:



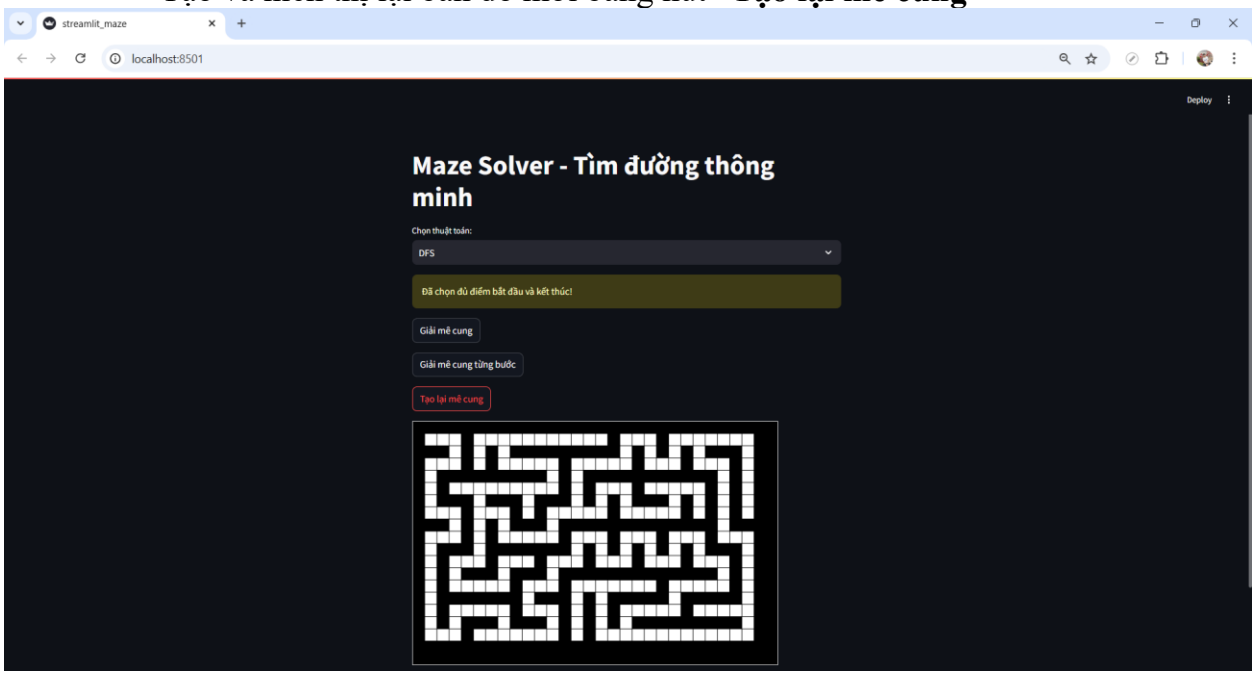
“Giải mê cung”: đưa ra trực tiếp đường đi đến điểm kết thúc.



“Giải mê cung từng bước”: Đưa ra file ảnh gif và trình chiếu cho người dùng đồng thời đưa ra thêm ảnh của đường đi ngắn nhất cuối cùng.

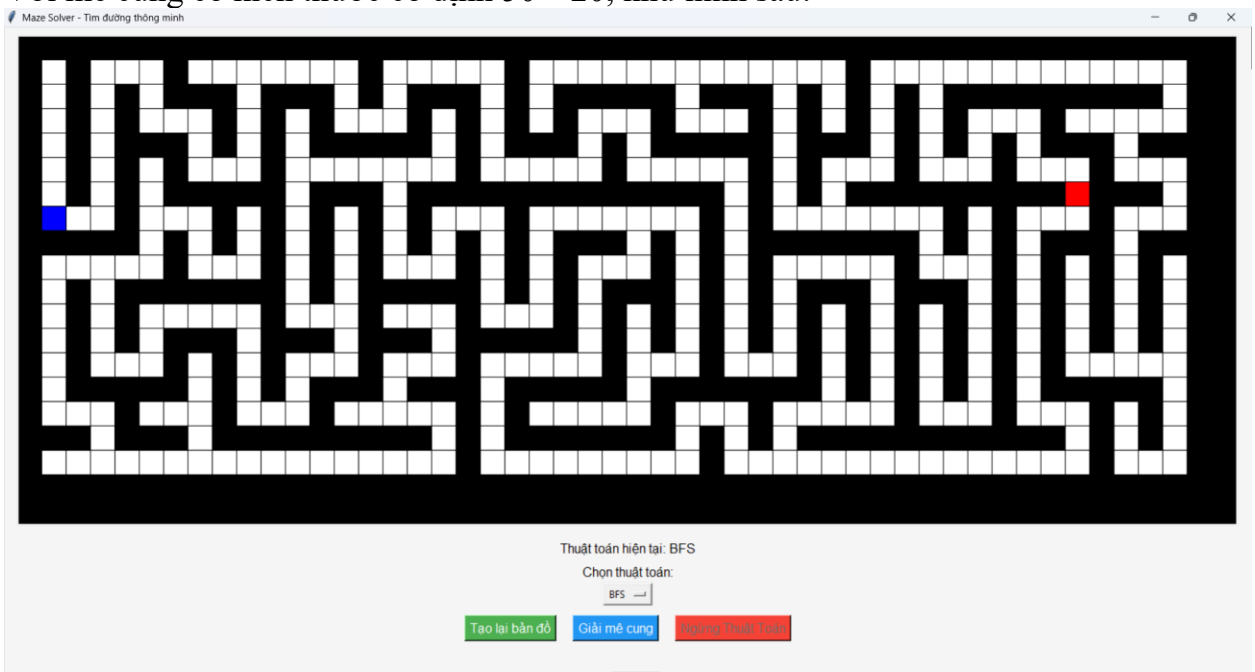


Tạo và hiển thị lại bản đồ mới bằng nút “**Tạo lại mê cung**”



5. Thực nghiệm và đánh giá:

Với mê cung có kích thước cố định 50 * 20, như hình sau:



Điểm bắt đầu là ô màu xanh dương
Điểm kết thúc là ô màu đỏ.

Kết quả các thuật toán, theo bảng sau:

Thuật toán	Số bước duyệt (ô được mở rộng)	Độ dài đường đi
BFS	152	112
DFS	116	112
A*	132	112
Greedy	119	112
Uniform Cost	152	112

Phân tích kết quả:

Breadth-First Search (BFS):

Số bước duyệt: BFS duyệt tất cả các trạng thái có cùng khoảng cách từ điểm bắt đầu trước khi tiến sang các trạng thái xa hơn, dẫn đến số bước duyệt cao nhất (152).

Độ dài đường đi: Vì BFS đảm bảo tìm được đường đi ngắn nhất, kết quả là 112.

Depth-First Search (DFS):

Số bước duyệt: DFS ưu tiên tìm kiếm theo chiều sâu, giảm số lượng trạng thái cần mở rộng so với BFS (116 bước duyệt).

Độ dài đường đi: May mắn trong trường hợp này, DFS tìm được đường đi ngắn nhất (112). Tuy nhiên, không phải lúc nào DFS cũng đảm bảo tối ưu.

A*:

Số bước duyệt: A* sử dụng heuristic để hướng dẫn tìm kiếm, kết hợp giữa chi phí thực tế và ước lượng chi phí còn lại, giúp giảm số bước duyệt so với BFS (132 bước).

Độ dài đường đi: A* đảm bảo tối ưu, tìm được đường đi ngắn nhất (112).

Greedy Best-First Search:

Số bước duyệt: Greedy tập trung mở rộng các trạng thái gần đích nhất, vì vậy số bước duyệt ít hơn BFS và A* (119 bước).

Độ dài đường đi: Greedy không đảm bảo tối ưu, nhưng trong trường hợp này, kết quả trùng với đường đi ngắn nhất (112).

Uniform Cost Search (UCS):

Số bước duyệt: UCS có cơ chế duyệt tương tự BFS, ưu tiên các trạng thái có chi phí thực tế thấp nhất. Kết quả là số bước duyệt bằng với BFS (152 bước).

Độ dài đường đi: UCS đảm bảo tối ưu, đường đi tìm được là 112.

Nhận xét tổng thể

BFS và UCS:

Mặc dù đảm bảo đường đi tối ưu, cả hai thuật toán phải mở rộng nhiều trạng thái nhất, dẫn đến số bước duyệt lớn.

Phù hợp với bài toán yêu cầu độ chính xác tuyệt đối nhưng không yêu cầu tối ưu về thời gian hoặc bộ nhớ.

DFS:

DFS nhanh hơn trong trường hợp này nhờ số bước duyệt ít hơn. Tuy nhiên, việc tìm được đường đi ngắn nhất chỉ là may mắn trong cấu trúc mê cung cụ thể này.

DFS không phải lựa chọn tốt khi yêu cầu đảm bảo tối ưu.

A*:

Là thuật toán cân bằng tốt nhất giữa tốc độ và độ chính xác. Nhờ heuristic hiệu quả, A* giảm đáng kể số bước duyệt so với BFS và UCS mà vẫn đảm bảo đường đi tối ưu.

Greedy:

Greedy tìm kiếm nhanh hơn A* do không tính chi phí thực tế mà chỉ tập trung vào ước lượng. Tuy nhiên, trong nhiều trường hợp phức tạp, Greedy có thể không tìm được đường đi ngắn nhất.

Đánh giá và ứng dụng thực tiễn:

BFS và UCS: Phù hợp với các bài toán cần kết quả chính xác, nhưng không yêu cầu tối ưu tốc độ hoặc bộ nhớ.

DFS: Phù hợp cho các mê cung nhỏ hoặc khi tốc độ quan trọng hơn độ tối ưu.

A*: Lựa chọn tốt nhất cho các bài toán lớn, yêu cầu cân bằng giữa tốc độ và độ chính xác.

Greedy: Phù hợp cho các bài toán yêu cầu tốc độ cao, nhưng không cần độ chính xác tuyệt đối.