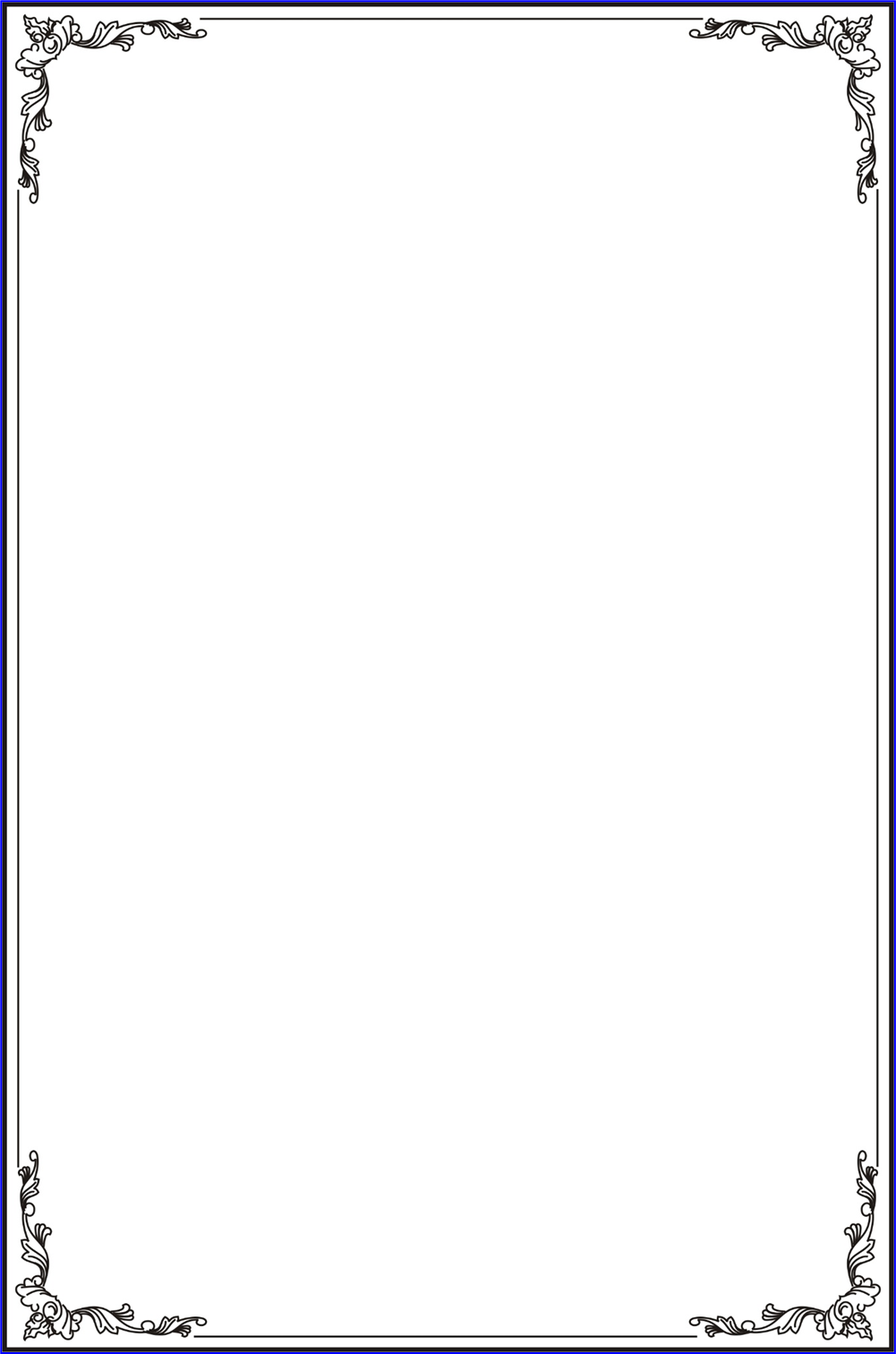
****

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

**KHOA CÔNG NGHỆ THÔNG TIN**

**🕯✡🕮🕮✡🕯**

****

**Giảng viên :PGS.TS Hoàng Văn Dũng**

|  |  |  |
| --- | --- | --- |
| **MSSV** | **Họ và tên** | **Mức độ đóng góp (%)** |
| 21110304 | Phan Vũ Thắng | 100% |
| 21110312 | Ngô Ngọc Thông | 100% |
| 21110301 | Phùng Hửu Thành | 100% |

|  |
| --- |
| ***Nhóm sinh viên thực hiện :*** |

**Tên tiểu luận: Xây dựng game giải mã mê cung dùng các thuật toán trong việc xây dựng bản đồ và tìm đường đi tối ưu**

**Báo cáo cuối kỳ:**

**TRÍ TUỆ NHÂN TẠO**

TP. Hồ Chí Minh, tháng 12 năm 2023

# DANH MỤC CÁC CỤM TỪ VIẾT TẮT

|  |  |
| --- | --- |
| AI | Artificial Intelligence |
| BFS | Breadth first search |
| DFS | Depth-First Search |
| USC | Uniform-Cost Search |
| TG | Thời gian |
| DD | Độ dài đường đi |
| SDD | Số đỉnh đã duyệt |

# LỜI NÓI ĐẦU

Trí tuệ nhân tạo đã xuất hiện được một thời gian dài, nhưng đến hiện tại thì vẫn chưa được phổ biến và xã hội hóa. Vậy nên dù có khá nhiều thông tin nhưng tìm hiểu về AI vẫn không phải là dễ. Qua bản báo cáo này, nếu có gì sai sót mong thầy thông cảm và góp ý để chúng em hoàn thiện hơn.

Để hoàn thành được báo cáo về môn học Trí tuệ nhân tạo, chúng em không thể quên gửi lời cảm ơn đến Giảng viên. Thầy Hoàng Văn Dũng đã cung cấp nguồn kiến thức và tài liệu để chúng em học tập, tìm hiểu.

Chúng em xin chân thành cảm ơn!

Tập thể nhóm

|  |  |
| --- | --- |
| **Điểm** | **Đánh giá của giảng viên** |
|  |  |

**MỤC LỤC**

[DANH MỤC CÁC CỤM TỪ VIẾT TẮT 2](#_Toc152758615)

[LỜI NÓI ĐẦU 3](#_Toc152758617)

[Chương 1: MÔ TẢ BÀI TOÁN 6](#_Toc152758618)

[1.1 Phát biểu bài toán 6](#_Toc152758619)

[1.2 Mục đích bài toán 6](#_Toc152758620)

[1.3 Yêu cầu 6](#_Toc152758621)

[Chương 2: CƠ SỞ LÝ THUYẾT 7](#_Toc152758622)

[2.1 Ngôn ngữ lập trình Python 7](#_Toc152758623)

[2.2 Môi trường lập trình Spyder 8](#_Toc152758624)

[2.3. Thư viện Pygame 8](#_Toc152758625)

[2.4 Các thuật toán 9](#_Toc152758626)

[2.4.1. Giải thuật BFS (Breadth-First Search) 9](#_Toc152758627)

[2.4.2. Giải thuật UCS 9](#_Toc152758628)

[2.4.3 Giải Thuật DFS 10](#_Toc152758629)

[2.4.4 Giải thuật Dijkstra 10](#_Toc152758630)

[2.4.5 Giải thuật A\* 10](#_Toc152758631)

[2.4.6 Giải thuật Greedy BFS 11](#_Toc152758632)

[Chương 3: PHÂN TÍCH ,THIẾT KẾ GIẢI PHÁP 12](#_Toc152758633)

[3.1. Phân tích bài toán 12](#_Toc152758634)

[3.2. Thiết kế giải pháp 12](#_Toc152758635)

[Chương 4: CÀI ĐẶT, XÂY DỰNG TRÒ CHƠI 20](#_Toc152758636)

[4.1. Cài đặt thuật toán 20](#_Toc152758637)

[4.1.1 Thuật toán BFS 20](#_Toc152758638)

[4.1.2. Thuật toán DFS 21](#_Toc152758639)

[4.1.3. Thuật toán USC 21](#_Toc152758640)

[4.1.4. Thuật toán DIJKSTRA 22](#_Toc152758641)

[4.1.5. Hàm Heuristic 23](#_Toc152758642)

[4.1.6 Thuật toán A\* 23](#_Toc152758643)

[4.1.7 Thuật toán Greedy 24](#_Toc152758644)

[4.2 Cài đặt giao diện và chức năng 25](#_Toc152758645)

[4.2.1. Tạo bản đồ 25](#_Toc152758646)

[4.2.2. Các hàm chức năng 27](#_Toc152758647)

[4.2.3. Tạo nút chức năng 29](#_Toc152758648)

[4.2.4 Giao diện trò chơi 31](#_Toc152758649)

[4.3 So sánh thuật toán 33](#_Toc152758650)

[Chương 5: KẾT LUẬN 34](#_Toc152758651)

[5.1. Đánh giá kết quả 34](#_Toc152758652)

[5.2. Hướng phát triển 35](#_Toc152758653)

[Tài Liệu Tham Khảo 36](#_Toc152758654)

# Chương 1: MÔ TẢ BÀI TOÁN

## 1.1 Phát biểu bài toán

Bài toán tìm đường là bài toán tìm kiếm đường đi giữa hai điểm đầu và cuối nếu có lời giải khả thi. Cụ thể với đề tài này, chúng em sẽ mô tả khu vực tìm kiếm bằng ma trận ngẫu nhiên với kích thước 50x25. Mỗi phần tử trong mảng đại diện cho một ô vuông trên bản đồ. Trên bản đồ xuất hiện các ô không thể đi được, gọi là các nút vật cản, là những ô không thể đi vào.

Bài toán sử dụng ba phương pháp tìm kiếm bao gồm tìm kiếm theo chiều rộng, tìm kiếm, theo chiều sâu và giải thuật A\*. Biết rằng ở một nút bất kỳ, nó chỉ có thể đi theo 4 hướng lên, xuống, trái, phải.

## 1.2 Mục đích bài toán

Áp dụng tìm đường đi từ 1 điểm đến 1 điểm trên mặt phẳng bằng cách sử dụng các phương pháp tìm kiếm BFS,DFS,USC,A\*.

## 1.3 Yêu cầu

Tìm hiểu chi tiết các thuật toán được sử dụng.

Thiết kế giao diện để mô phỏng thực hiện các thuật toán.

Thực thi chương trình, so sánh và đánh giá các thuật toán dựa trên thời gian thực hiện và chi phí đường đi từ điểm đầu đến điểm cuối.

# Chương 2: CƠ SỞ LÝ THUYẾT

## 2.1 Ngôn ngữ lập trình Python

**\* Giới thiệu**

Python là ngôn ngữ lập trình máy tính bậc cao thường được sử dụng để xây dựng trang web và phần mềm, tự động hóa các tác vụ và tiến hành phân tích dữ liệu. Python là ngôn ngữ có mục đích chung, nghĩa là nó có thể được sử dụng để tạo nhiều chương trình khác nhau và không chuyên biệt cho bất kỳ vấn đề cụ thể nào. Python nỗ lực hướng đến một cú pháp đơn giản hơn, gọn gàng hơn trong khi vẫn cho các nhà phát triển lựa chọn phương pháp viết code của họ. Đối lập với khẩu hiệu "có nhiều hơn một cách để làm việc này", triết lý thiết kế của Python lại nằm trong châm ngôn "chỉ nên có một— và tốt nhất là chỉ một—cách rõ ràng để làm việc này”. Thay vì tích hợp hết tất cả các tính năng vào phần cốt lõi, Python được thiết kế để dễ dàng mở rộng (bằng các mô đun). Tính mô đun nhỏ gọn này đã làm cho Python trở nên phổ biến như là một cách thêm các giao diện lập trình được vào các ứng dụng hiện có.

**\*** **Đặc điểm**

- Python được thông dịch: Python được trình thông dịch xử lý trong thời gian chạy. Bạn không cần phải biên dịch chương trình của mình trước khi thực hiện nó.

- Python có tính tương tác (Interactive): Tại một dấu nhắc Python (command line) bạn có thể tương tác trực tiếp với trình thông dịch để viết chương trình Python.

- Python hỗ trợ kỹ thuật lập trình hướng đối tượng hoặc kỹ thuật lập trình đóng gói mã trong các đối tượng.

- Python là ngôn ngữ linh hoạt và dễ tiếp cận: Python là ngôn ngữ tuyệt vời cho các lập trình viên và hỗ trợ phát triển một loạt các ứng dụng từ xử lý văn bản đơn giản, lập trình web, cho đến lập trình game.

## 2.2 Môi trường lập trình Spyder

Spyder Là một môi trường phát triển tương tác mạnh mẽ cho ngôn ngữ Python. Tôi có các tính năng chỉnh sửa nâng cao, kiểm tra tương tác, gỡ lỗi và xem xét nội tâm cũng như môi trường máy tính số. Nhờ sự hỗ trợ của IPython (trình thông dịch Python tương tác được cải thiện) và các thư viện Python phổ biến như NumPy, SciPy hoặc matplotlib (Vẽ đồ thị tương tác 2D / 3D).

Spyder cũng có thể được sử dụng như một thư viện cung cấp các tiện ích mạnh mẽ liên quan đến bảng điều khiển cho các ứng dụng dựa trên PyQt của chúng tôi. Nó có thể được sử dụng để tích hợp bảng điều khiển gỡ lỗi trực tiếp vào thiết kế giao diện người dùng đồ họa của bạn.

## 2.3. Thư viện Pygame

Pygame là một bộ mô-đun Python đa nền tảng được thiết kế để viết trò chơi điện tử. Nó bao gồm đồ họa máy tính và thư viện âm thanh được thiết kế để sử dụng với ngôn ngữ lập trình Python.

Pygame sử dụng thư viện Simple DirectMedia Layer (SDL), với mục đích cho phép phát triển trò chơi máy tính trong thời gian thực mà không cần cơ chế bậc thấp của ngôn ngữ lập trình C và các dẫn xuất của nó. Điều này dựa trên giả định rằng các chức năng đắt tiền nhất bên trong trò chơi có thể được trừu tượng hóa khỏi logic trò chơi, do đó có thể sử dụng ngôn ngữ lập trình bậc cao, chẳng hạn như Python, để cấu trúc trò chơi. 11 Các tính năng khác mà SDL không có bao gồm toán học vectơ, phát hiện va chạm, quản lý đồ họa 2D, hỗ trợ MIDI, camera, thao tác mảng pixel, chuyển đổi, lọc, hỗ trợ phông chữ freetype nâng cao và vẽ.

Trong đề tài này, nhóm chúng em sử dụng Pygame để xây dựng giao diện và mô phỏng chuyển động thuật toán.

## 2.4 Các thuật toán

### 2.4.1. Giải thuật BFS (Breadth-First Search)

**Giới thiệu thuật toán BFS**

Thuật toán duyệt đồ thị ưu tiên chiều rộng (Breadth-First Search) là thuật toán tìm kiếm mù mà những đỉnh gần với đỉnh gốc sẽ được duyệt trước.

**Ý tưởng**

Với đồ thị không trọng số và đỉnh nguồn s. Đồ thị này có thể là đồ thị có hướng hoặc vô hướng, điều đó không quan trọng đối với thuật toán.

Có thể hiểu thuật toán như một ngọn lửa lan rộng trên đồ thị:

Ở bước thứ 0, chỉ có đỉnh nguồn s đang cháy.

Ở mỗi bước tiếp theo, ngọn lửa đang cháy ở mỗi đỉnh lại lan sang tất cả các đỉnh kề với nó.

Trong mỗi lần lặp của thuật toán, "vòng lửa" lại lan rộng ra theo chiều rộng. Những đỉnh nào gần s hơn sẽ bùng cháy trước.

### 2.4.2. Giải thuật UCS

**Giới thiệu thuật toán USC**

Thuật toán tìm kiếm chi phí đều (Uniform Cost Search) là một biến thể của thuật toán Dijkstra, được sử dụng hiệu quả trong những đồ thị vô hạn hoặc những đồ thị quá lớn để biểu diễn trong bộ nhớ. Thuật toán UCS được sử dụng chính trong ngành Trí tuệ Nhân tạo.

**Ý tưởng**

Việc tìm kiếm sẽ bắt đầu tại nút gốc và tiếp tục tìm bằng cách duyệt các nút tiếp theo với trọng số (chi phí) thấp nhất tính từ nút gốc. Các nút được duyệt tiếp tục cho đến khi đến được nút đích.

### 2.4.3 Giải Thuật DFS

**Giới thiệu thuật toán DFS**

Một phiên bản của tìm kiếm theo chiều sâu đã được nhà toán học người Pháp Charles Pierre Trémaux nghiên cứu vào thế kỷ 19 như một chiến lược để giải các mê cung.

**Ý Tưởng**

Từ một đỉnh, ta đi qua sâu vào từng nhánh. Khi đã duyệt hết nhánh của đỉnh, lùi lại để duyệt đỉnh tiếp theo.

Thuật toán dừng khi đi qua hết tất cả các đỉnh có thể đi qua.

### 2.4.4 Giải thuật Dijkstra

**Giới thiệu thuật toán Dijkstra**

Thuật toán Dijkstra có thể giải quyết bài toán tìm đường đi ngắn nhất trên đồ thị vô hướng lẫn có hướng miễn là trọng số không âm.

**Ý tưởng**

Thuật toán sẽ tìm khoảng cách từ đỉnh gốc 0 tới tất cả các đỉnh còn lại trong đồ thị

Bắt đầu bằng việc chọn điểm gốc, chọn điểm a kề điểm gốc có khoảng cách nhỏ nhất

Sau đó lấy các điểm kề điểm a, nếu khoảng cách từ gốc đến các điểm đó nhỏ hơn khoảng cách từ gốc sang a từ a sang điểm đó thì sẽ ghi nhận và lặp lại cho đến khi tìm ra đường đi ngắn nhất đến tất cả các điểm

### 2.4.5 Giải thuật A\*

**Giới thiệu thuật toán A\***

Thuật toán này sử dụng một "đánh giá heuristic" để xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó. Thuật toán này duyệt các nút theo thứ tự của đánh giá heuristic này. Do đó, thuật toán A\* là một ví dụ của tìm kiếm theo lựa chọn tốt nhất (BFS).

**Ý tưởng**

A\* xây dựng tăng dần tất cả các tuyến đường từ điểm xuất phát cho tới khi nó tìm thấy một đường đi chạm tới đích. Tuy nhiên, cũng như tất cả các thuật toán tìm kiếm có thông tin nó chỉ xây dựng các tuyến đường “có vẻ” như dẫn về phía đích. Để biết những tuyến đường nào có khả năng sẽ dẫn tới đích, A\* sử dụng một “đánh giá heuristic” về khoảng cách từ điểm bất kỳ cho tới đích. Trong trường hợp tìm đường đi ,đánh giá này có thể là khoảng cách đường chim bay

### 2.4.6 Giải thuật Greedy BFS

**Giới thiệu thuật toán Greedy BFS**

Greedy Best-First Search là một thuật toán tìm kiếm với tri thức bổ sung từ việc sử dụng các tri thức cụ thể của bài toán.

**Ý tưởng**

Thuật toán GBFS sẽ đánh giá các đỉnh bằng cách sử dụng hàm heuristic h(n), nghĩa là hàm đánh giá sẽ bằng với hàm heuristic f(n) = h(n). Điều này khiến cho thuật toán trở nên ‘tham lam’.

# Chương 3: PHÂN TÍCH ,THIẾT KẾ GIẢI PHÁP

## 3.1. Phân tích bài toán

Đây là bài toán về việc sử dụng các thuật toán tìm kiếm AI không học. Vì vậy để kiểm chứng được độ chính xác của thuật toán thì cần phải có được những bản đồ khác nhau để cho ra được sự so sánh chính xác về hướng đi, cũng như là cách giải của các thuật toán. Do đó chúng ta cần phải tạo được những bản đồ ngẫu nhiên để giải quyết được vấn đề trên

Công việc giải mê cung thì chúng ta có thể hiểu nó như là việc tìm đường đi giữa 2 điểm mà giữa 2 điểm đó có nhiều chướng ngại vật ở giữa vì vậy chúng ta có thể sử dụng những thuật toán tìm kiếm đường đi phổ biến hiện nay để có thể tìm được những đường đi thích hợp.

Các thuật toán được sử dụng trong dự án lần này bào gồm các thuật toán DFS, BFS, UCS, Dijkstra, A\*, greedy. Đây là những thuật toán tìm kiếm đường đi phổ thông hiện nay, để có thể áp dụng được những giải thuật này vào trong dự án, thì cần phải hiểu được nguyên lí hoạt động ý tưởng cách triển khai và cài đặt

Qua đó có thể so sánh kiểm chứng cũng như đánh giá được độ hiệu quả, sự tiện lợi cũng như là chi phải bỏ ra cho các kết quả qua từng thuật toán

## 3.2. Thiết kế giải pháp

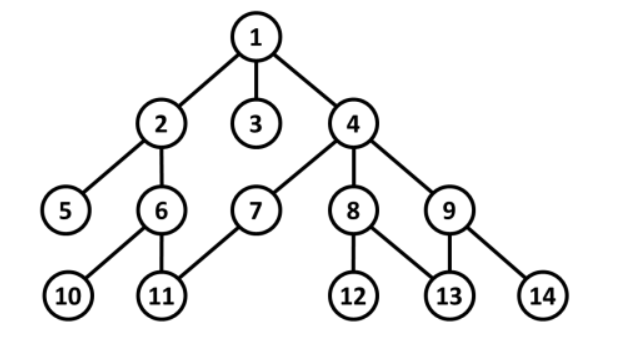
Để xử lí được các vấn đề ở trên, đầu tiên chúng ta cần tạo ra được bản đồ ngẫu nhiên để kiểm chứng được các thuật toán. Ở đây em đã dùng thuật toán Recursive - Backtracking để tạo ra được các bản đồ đường đi ngẫu nhiên. Đồng thời cũng có map trắng để có thể thấy được các đường đi trong trường hợp đặc biệt

Sau khi đã có được bản đồ thì sẽ thực hiện việc giải mê cung bằng các thuật toán. Mấu chốt ở đây là việc có thể hiểu và cài đặt được các thuật toán khác thì sẽ cho ra được các kết quả khác nhau và thêm nữa là chi phí để sửa dụng các thuật toán cũng khác nhau

Vì lẽ đó ngoài việc tìm hiểu và cài đặt các thuật toán chúng ta cần có được những hàm có thể đếm được thời gian và chi phí cho từng thuật toán khác nhau, sau đó lưu lại các kết quả đã thu được để có thể dựa vào mà đánh giá được thời gian, chi phí và độ phức tạp của bài toán qua đó có thể đánh giá và sử dụng được thuật toán trong các trường hợp khác nhau

**Tìm hiểu các thuật toán**

**Thuật toán BFS:**

****

**Mô tả**

1. Chọn điểm bắt đầu, chọn ra các đỉnh có thể đi được từ điểm bắt đầu đưa vào hàng đợi (đang hướng tới)

2. Lấy ra đỉnh đầu tiên trong hàng đợi và quan sát nó:

- Nếu đỉnh này chính là đỉnh đích, dừng quá trình tìm kiếm và trả về kết quả.

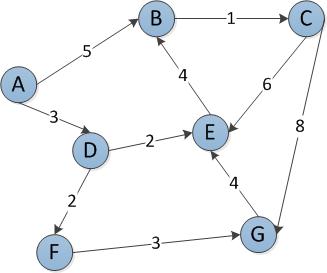
- Nếu không phải thì chèn tất cả các đỉnh kề với đỉnh vừa thăm nhưng chưa được quan sát trước đó vào hàng đợi.

3. Nếu hàng đợi là rỗng, thì tất cả các đỉnh có thể đến được đều đã được quan sát

–> dừng việc tìm kiếm và trả về "không thấy".

4. Nếu hàng đợi không rỗng thì quay về bước 2.

**Thuật toán UCS:**



Khởi tạo: PQ rỗng, CLOSE rỗng.

Đưa trạng thái ban đầu START vào PQ, độ ưu tiên g(START) = 0

Lặp đến khi PQ rỗng

Lấy một trạng thái n (có g thấp nhất) ra khỏi PQ. Đưa n vào CLOSE.

Nếu n là trạng thái đích GOAL thì “đã tìm thấy”.

Dừng thuật toán.

Nếu không, với mỗi trạng thái con n’chưa xét (n’ không thuộc CLOSE) của n

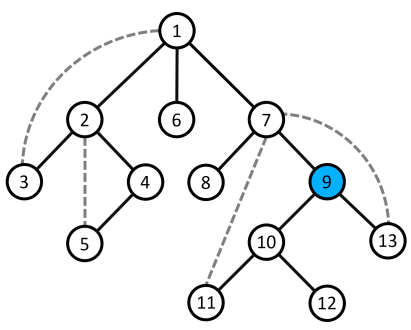
Tính độ ưu tiên: g(n’) = g(n) + cost(n, n’)

đưa (n’, g(n’)) vào PQ

Kết thúc:

Thông báo không có đường đi từ START đến GOAL.

**Thuật toán DFS:**



- Đây là thuật toán tìm các đỉnh bằng cách duyệt theo chiều sâu.

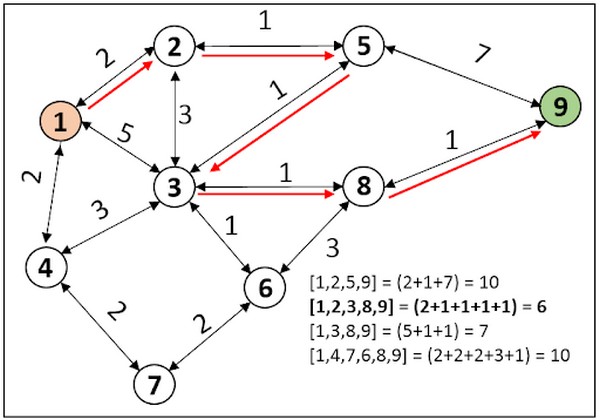
- Xuất phát từ 1 đỉnh và đi cho đến khi không thể đi tiếp, sau đó đi về lại đỉnh đầu. Trong quá trình quay lại:

+ Nếu gặp đường đi khác thì đi cho đến khi không đi tiếp được nữa

+ Nếu không tìm ra đường đi nào khác thì ngừng việc tìm kiếm.

Trong quá trình đi đến đỉnh khác, thuật toán sẽ lưu lại đỉnh cha vừa đi qua để khi đi ngược lại từ đỉnh Kết thúc đến đỉnh Xuất phát, ta có thể xem được đường đi từ đỉnh Kết thúc đến đỉnh Bắt Đầu

**Thuật toán Dijkstra:**



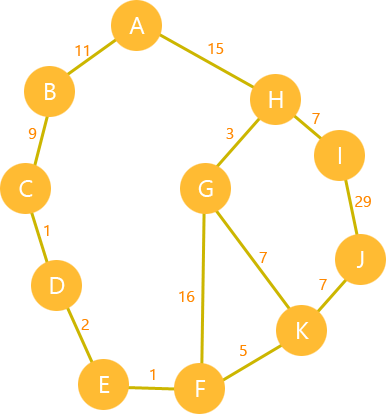
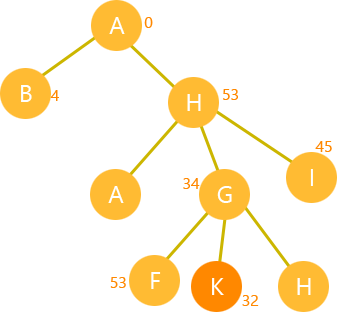
+ Từ đỉnh gốc, khởi tạo khoảng cách tới chính nó là 0, khởi tạo khoảng cách nhỏ nhất ban đầu tới các đỉnh khác là +∞. Ta được danh sách các khoảng cách tới đỉnh.

+ Chọn đỉnh a có khoảng cách nhỏ nhất trong danh sách này và ghi nhận. Các lần sau sẽ không xét tới đỉnh này nữa.

+ Lần lượt xét các đỉnh kề b của đỉnh a. Nếu khoảng cách từ đỉnh gốc tới đỉnh b nhỏ hơn khoảng cách hiện tại đang được ghi nhận thì cập nhật giá trị và đỉnh kề a vào khoảng cách hiện tại của b.

+ Sau khi xét tất cả đỉnh kề b của đỉnh a. Lúc này ta được danh sách khoảng cách tới các điểm đã được cập nhật. Quay lại Bước 2 với danh sách này. Thuật toán kết thúc khi chọn được khoảng cách nhỏ nhất từ tất cả các điểm.

**Thuật toán A\*:**



Thuật toán lưu giữ một tập các lời giải chưa hoàn chỉnh, nghĩa là các đường đi qua đồ thị, bắt đầu từ nút xuất phát.

Tập lời giải này được lưu trong một hàng đợi ưu tiên (priority queue).

Thứ tự ưu tiên gán cho một đường đi x được quyết định bởi hàm f(x) = g(x) + h(x). 13 Trong đó:

g(x) là chi phí của đường đi cho đến thời điểm hiện tại, nghĩa là tổng trọng số của các cạnh đã đi qua.

h(x) là hàm đánh giá heuristic về chi phí nhỏ nhất để đến đích từ x.

Ví dụ, nếu "chi phí" được tính là khoảng cách đã đi qua, khoảng cách đường chim bay giữa hai điểm trên một bản đồ là một đánh giá heuristic cho khoảng cách còn phải đi tiếp.

Hàm f(x) có giá trị càng thấp thì độ ưu tiên của x càng cao (do đó có thể sử dụng một cấu trúc heap tối thiểu để cài đặt hàng đợi ưu tiên này).

**A diagram of a diagram of a number of squares

Description automatically generated with medium confidenceThuật toán Greedy:**

Ta xây dựng tập S dần từng bước, bắt đầu từ tập S =∅

Tại mỗi bước, ta sẽ chọn một phần tử “tốt nhất” trong các phần tử còn lại của A để đưa vào tập S.

Việc chọn lựa một phần tử như thế ở mỗi bước sẽ được hướng dẫn bởi hàm chọn.

Phần tử được chọn bị loại khỏi tập A  
Nếu khi thêm phần tử được chọn vào tập S mà S vẫn thoả mãn các điều kiện của bài toán, ta tiếp tục mở rộng S bằng cách thêm vào phần tử được chọn

# Chương 4:CÀI ĐẶT, XÂY DỰNG TRÒ CHƠI

## 4.1. Cài đặt thuật toán

### 4.1.1 Thuật toán BFS

# Hàm để thực hiện một bước BFS

def add\_neighbors\_to\_queue(maze, queue, current, prev, visited, end):

    global vertices\_visited

    x, y = current

    for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

        next\_x, next\_y = x + direction[0], y + direction[1]

        if 0 <= next\_x < len(maze[0]) and 0 <= next\_y < len(maze) and \

           maze[next\_y][next\_x] == 0 and (next\_x, next\_y) not in visited:

            queue.append((next\_x, next\_y))

            prev[(next\_x, next\_y)] = current

            visited.add((next\_x, next\_y))

            vertices\_visited += 1

            if (next\_x, next\_y) != end:  # Không vẽ lại ô đích

                draw\_cell(BLUE,next\_x, next\_y)  # Vẽ màu cho ô mới

                pygame.display.flip()  # Cập nhật màn hình

                time.sleep(0.01)  # Điều chỉnh tốc độ hiển thị

def bfs\_step(maze, queue, prev, visited, end):

    global start\_time, end\_time

    if not queue:

        return True, False  # Queue rỗng, không tìm thấy lời giải

    if queue[0] == start:  # Bắt đầu tính thời gian khi bắt đầu từ điểm xuất phát

        start\_time = time.time()

    current = queue.popleft()

    if current == end:

        end\_time = time.time()  # Dừng đo thời gian khi tìm thấy lối ra

        return True, True  # Tìm thấy lời giải

    add\_neighbors\_to\_queue(maze, queue, current, prev, visited, end)

    return False, False  # BFS vẫn đang tiếp diễn

### 4.1.2. Thuật toán DFS

def dfs\_step(maze, stack, prev, visited, end):

    global vertices\_visited, start\_time, end\_time

    if not stack:

        return True, False

    if stack[-1] == start:  # Bắt đầu tính thời gian từ điểm xuất phát

        start\_time = time.time()

    current = stack.pop()

    if current == end:

        end\_time = time.time()  # Kết thúc tính thời gian khi tìm thấy lời giải

        return True, True  # Tìm thấy lời giải

    vertices\_visited += 1

    x, y = current

    for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

        next\_x, next\_y = x + direction[0], y + direction[1]

        if 0 <= next\_x < len(maze[0]) and 0 <= next\_y < len(maze) and \

           maze[next\_y][next\_x] == 0 and (next\_x, next\_y) not in visited:

            stack.append((next\_x, next\_y))

            prev[(next\_x, next\_y)] = current

            visited.add((next\_x, next\_y))

            if (next\_x, next\_y) != end:  # Kiểm tra để không vẽ lại ô đích

                draw\_cell(BLUE, next\_x, next\_y)  # Vẽ màu cho ô mới

                pygame.display.flip()  # Cập nhật màn hình

                time.sleep(0.01)  # Điều chỉnh tốc độ hiển thị

    return False, False  # DFS vẫn đang tiếp diễn

### 4.1.3. Thuật toán USC

def ucs\_step(maze, priority\_queue, prev, visited, cost, end):

    global vertices\_visited, start\_time, end\_time

    if not priority\_queue:

        return True, False

    if priority\_queue[0][1] == start:

        start\_time = time.time()

    current\_cost, current = heapq.heappop(priority\_queue)

    if current == end:

        end\_time = time.time()

        return True, True

    x, y = current

    for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

        next\_x, next\_y = x + direction[0], y + direction[1]

        if 0 <= next\_x < len(maze[0]) and 0 <= next\_y < len(maze) and \

           maze[next\_y][next\_x] == 0 and (next\_x, next\_y) not in visited:

            next\_cost = current\_cost + 1

            heapq.heappush(priority\_queue, (next\_cost, (next\_x, next\_y)))

            prev[(next\_x, next\_y)] = current

            cost[(next\_x, next\_y)] = next\_cost

            visited.add((next\_x, next\_y))

            vertices\_visited += 1

            if (next\_x, next\_y) != end:

                draw\_cell(BLUE, next\_x, next\_y)

                pygame.display.flip()

                time.sleep(0.01)

    return False, False

### 4.1.4. Thuật toán DIJKSTRA

def dijkstra\_step(maze, priority\_queue, prev, visited, cost, end):

    global vertices\_visited, start\_time, end\_time

    if not priority\_queue:

        return True, False

    if priority\_queue[0][1] == start:

        start\_time = time.time()

    current\_cost, current = heapq.heappop(priority\_queue)

    if current == end:

        end\_time = time.time()

        return True, True

    x, y = current

    for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

        next\_x, next\_y = x + direction[0], y + direction[1]

        if 0 <= next\_x < len(maze[0]) and 0 <= next\_y < len(maze) and \

           maze[next\_y][next\_x] == 0 and (next\_x, next\_y) not in visited:

            next\_cost = current\_cost + 1

            if next\_cost < cost.get((next\_x, next\_y), float('inf')):

                heapq.heappush(priority\_queue, (next\_cost, (next\_x, next\_y)))

                prev[(next\_x, next\_y)] = current

                cost[(next\_x, next\_y)] = next\_cost

                visited.add((next\_x, next\_y))

                vertices\_visited += 1

                if (next\_x, next\_y) != end:

                    draw\_cell(BLUE, next\_x, next\_y)

                    pygame.display.flip()

                    time.sleep(0.01)

    return False, False

### 4.1.5. Hàm Heuristic

Hàm Heuristic có tác dụng ước lượng khoảng cách từ một nút tới nút đích, giúp thuật toán xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó.

# Hàm heuristic (khoảng cách Manhattan)

def heuristic(cell, end):

    return abs(cell[0] - end[0]) + abs(cell[1] - end[1])

# Define a cutoff for the heuristic cost.

HEURISTIC\_CUTOFF = 10  # This value is arbitrary and should be chosen based on your specific use case.

### 4.1.6 Thuật toán A\*

# Hàm thực hiện bước tiếp theo của A\*

def astar\_step(maze, priority\_queue, prev, visited, cost, end):

    global vertices\_visited, start\_time, end\_time

    if not priority\_queue:

        return True, False

    if priority\_queue[0][1] == start:

        start\_time = time.time()

    current\_cost, current = heapq.heappop(priority\_queue)

    if current == end:

        end\_time = time.time()

        return True, True

    vertices\_visited += 1

    x, y = current

    for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

        next\_x, next\_y = x + direction[0], y + direction[1]

        if 0 <= next\_x < len(maze[0]) and 0 <= next\_y < len(maze) and \

           maze[next\_y][next\_x] == 0 and (next\_x, next\_y) not in visited:

            next\_cost = current\_cost + 1

            h\_cost = heuristic((next\_x, next\_y), end)

            total\_cost = next\_cost + h\_cost

            if (next\_x, next\_y) not in cost or total\_cost < cost.get((next\_x, next\_y), float('inf')):

                heapq.heappush(priority\_queue, (total\_cost, (next\_x, next\_y)))

                prev[(next\_x, next\_y)] = current

                cost[(next\_x, next\_y)] = next\_cost

                if (next\_x, next\_y) != end:

                    draw\_cell(BLUE, next\_x, next\_y)

                    pygame.display.flip()

                    time.sleep(0.01)

    return False, False

### 4.1.7 Thuật toán Greedy

def greedy\_step(maze, priority\_queue, prev, visited, end):

    global vertices\_visited, start\_time, end\_time

    if not priority\_queue:

        return True, False

    if priority\_queue[0][1] == start:

        start\_time = time.time()

    \_, current = heapq.heappop(priority\_queue)

    if current == end:

        end\_time = time.time()

        return True, True

    vertices\_visited += 1

    x, y = current

    for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

        next\_x, next\_y = x + direction[0], y + direction[1]

        if 0 <= next\_x < len(maze[0]) and 0 <= next\_y < len(maze) and \

           maze[next\_y][next\_x] == 0 and (next\_x, next\_y) not in visited:

            h\_cost = heuristic((next\_x, next\_y), end)

            heapq.heappush(priority\_queue, (h\_cost, (next\_x, next\_y)))

            prev[(next\_x, next\_y)] = current

            visited.add((next\_x, next\_y))

            if (next\_x, next\_y) != end:

                draw\_cell(BLUE, next\_x, next\_y)

                pygame.display.flip()

                time.sleep(0.01)

    return False, False # Greedy vẫn đang tiếp diễn

## 4.2 Cài đặt giao diện và chức năng

### 4.2.1. Tạo bản đồ

Tạo bản đồ ngẫu nhiên

def generate\_maze(width, height, open\_wall\_rate=0.08):

    global start, end

    def get\_neighbors(x, y):

        neighbors = []

        for dx, dy in [(0, 2), (2, 0), (0, -2), (-2, 0)]:  # Tăng khoảng cách giữa các ô

            nx, ny = x + dx, y + dy

            if 0 <= nx < width and 0 <= ny < height:

                neighbors.append((nx, ny))

        return neighbors

    def can\_be\_extended(x, y):

        # Kiểm tra xem ô có thể mở rộng mà không tạo ra nhánh cụt ngắn không

        count = 0

        for dx, dy in [(0, 2), (2, 0), (0, -2), (-2, 0)]:

            nx, ny = x + dx\*2, y + dy\*2

            if 0 <= nx < width and 0 <= ny < height and maze[ny][nx] == 0:

                count += 1

        return count <= 1  # Chỉ mở rộng nếu có 1 hoặc không có ô đã mở rộng xung quanh

    def carve\_passage\_from(x, y, maze, visited, width, height):

        directions = [(0, 2), (2, 0), (0, -2), (-2, 0)]

        random.shuffle(directions)

        for dx, dy in directions:

            nx, ny = x + dx, y + dy

            if (0 <= nx < width) and (0 <= ny < height) and (maze[ny][nx] == 1):

                between\_x, between\_y = (x + nx) // 2, (y + ny) // 2

                if not visited.get((nx, ny), False):

                    maze[between\_y][between\_x] = 0

                    maze[ny][nx] = 0

                    visited[(nx, ny)] = True

                    update\_display(maze, width, height, start, end)

                    carve\_passage\_from(nx, ny, maze, visited, width, height)

    maze = [[1 for \_ in range(width)] for \_ in range(height)]

    start = get\_random\_edge\_cell(width, height)

    end = opposite\_edge(start, width, height)

    visited = {(start[0], start[1]): True}

    maze[start[1]][start[0]] = 0

    carve\_passage\_from(start[0], start[1], maze, visited, width, height)

    maze[end[1]][end[0]] = 0

    for y in range(height):

        for x in range(width):

            if maze[y][x] == 1 and random.random() < open\_wall\_rate:

                maze[y][x] = 0

    update\_display(maze, width, height, start, end)

    return maze

### 4.2.2. Các hàm chức năng

**Hàm reset\_maze:** Đặt lại mê cung về trạng thái chưa giải

def reset\_maze(maze):

    for y in range(len(maze)):

        for x in range(len(maze[0])):

            if maze[y][x] == 0:  # Kiểm tra nếu ô hiện tại là đường đi

                if (x, y) != start and (x, y) != end:  # Bỏ qua ô xuất phát và ô đích

                    draw\_cell(BLACK, x, y)  # Chuyển thành màu đen

    pygame.display.flip()

**Hàm draw\_solution\_path:** Vẽ lộ trình giải cho thuật toán

def draw\_solution\_path(prev, start, end):

    global path\_length

    current = prev[end]

    path\_length = 0

    while current != start:

        path\_length += 1  # Tăng độ dài đường đi

        x, y = current

        draw\_cell(YELLOW, x, y)

        current = prev[current]

        pygame.display.flip()

        time.sleep(0.01)

    draw\_cell(RED, start[0], start[1])   # Vẽ lại ô xuất phát

    draw\_cell(GREEN, end[0], end[1])     # Vẽ lại ô đích

### 4.2.3. Tạo nút chức năng

# Hàm vẽ các nút

def draw\_button(button, image):

    # Blit the image at the position of the button

    screen.blit(image, button.topleft)

# Vẽ menu

def draw\_menu():

    # Calculate the menu background size

    menu\_background\_width = 0

    menu\_background\_height = 0 \* 0

    # Scale the PINK image to the size of the menu background

    menu\_background\_image = pygame.transform.scale(PINK, (menu\_background\_width, menu\_background\_height))

    # Draw the menu background image

    menu\_img\_position = (total\_maze\_width + margin\_x, margin\_y)

    screen.blit(menu\_background\_image, menu\_img\_position)

    # Draw the buttons with their respective images

    draw\_button(button\_generate, BUTTON\_IMAGE\_1)

    draw\_button(button\_generate\_easy, BUTTON\_IMAGE\_2)

    draw\_button(button\_solve\_bfs, BUTTON\_IMAGE\_3)

    draw\_button(button\_solve\_dfs, BUTTON\_IMAGE\_4)

    draw\_button(button\_solve\_ucs, BUTTON\_IMAGE\_5)

    draw\_button(button\_solve\_dijkstra, BUTTON\_IMAGE\_6)

    draw\_button(button\_solve\_astar, BUTTON\_IMAGE\_7)

    draw\_button(button\_solve\_greedy, BUTTON\_IMAGE\_8)

#    draw\_button(button\_player, PLAYER\_BUTTON\_IMAGE)

    pygame.display.flip()

### 4.2.4 Giao diện trò chơi

A screenshot of a video game

Description automatically generated

Các bước thực thi trò chơi:

1. Khi nhấn Start Game giao diện menu sẽ hiện ra



1. Chọn 1 trong 2 nút CREATE MAZE 1 để tạo ra bản đồ ngẫu nhiên.

Bản đồ của trò chơi:

A maze game with grass and trees

Description automatically generated

1. A close-up of a sign

   Description automatically generatedNhấn nút CREATE MAZE 2 để tạo một bản đồ trắng. Tại đây người chơi có thể tự tùy chỉnh map .
2. Chọn 1 trong các SOLVE ví dụ như để giải mê cung:
3. Nhấn nút Back để trở về giao diện chính
4. Nhấn nút Exit Game để thoát trò chơi

## 4.3 So sánh thuật toán

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Map*** | ***BFS*** | | | ***DFS*** | | | ***USC*** | | | ***DIJKSTRA*** | | | ***A\**** | | | | ***Greedy*** | | |
| **TG** | **DD** | **SDD** | **TG** | **DD** | **SDD** | **TG** | **DD** | **SDD** | **TG** | **DD** | **SDD** | **TG** | **DD** | **SDD** | **TG** | | **DD** | **SDD** |
| **1** | 9.66 | 106 | 572 | 6.18 | 218 | 397 | 9.46 | 106 | 564 | 9.49 | 106 | 564 | 6.55 | 106 | 386 | 4 | | 184 | 236 |
| **2** | 5.54 | 55 | 330 | 3,71 | 187 | 215 | 5.64 | 55 | 333 | 5.65 | 55 | 333 | 4.16 | 55 | 236 | 1.89 | | 55 | 107 |
| **3** | 10.01 | 95 | 586 | 9.06 | 275 | 522 | 10.14 | 95 | 591 | 10.15 | 95 | 591 | 8.78 | 95 | 509 | 4.82 | | 175 | 278 |
| **4** | 6.47 | 131 | 383 | 2.43 | 131 | 140 | 6.38 | 131 | 378 | 6.42 | 131 | 378 | 5.8 | 131 | 337 | 2.66 | | 131 | 154 |
| **5** | 2.85 | 58 | 176 | 6.94 | 272 | 401 | 2.98 | 58 | 184 | 2.96 | 58 | 184 | 1.76 | 58 | 104 | 1.14 | | 58 | 67 |
| **6** | 7.16 | 91 | 426 | 10.73 | 125 | 620 | 6.94 | 91 | 415 | 6.95 | 91 | 415 | 4.2 | 91 | 246 | 1.68 | | 91 | 96 |
| **7** | 11.17 | 118 | 670 | 3.92 | 138 | 228 | 11.49 | 118 | 671 | 11.45 | 118 | 671 | 10.4 | 130 | 605 | 3.74 | | 140 | 201 |

# Chương 5: KẾT LUẬN

## 5.1. Đánh giá kết quả

Chương trình mà nhóm chúng em phát triển, tập trung vào việc tạo ra một trò chơi giải mê cung, đã đạt được những thành tựu đáng kể. Chúng em đã thành công trong việc áp dụng các thuật toán phức tạp để máy có thể tự tìm ra lối thoát khỏi mê cung, một tính năng thú vị và thách thức.

**+ Ưu điểm:**

* Giao diện người dùng đơn giản, trực quan: Điều này giúp người chơi dễ dàng tiếp cận và thao tác với trò chơi, ngay cả với những người mới tiếp cận.
* Đa dạng thuật toán tìm đường trong mê cung: Sự đa dạng này không chỉ tạo thêm thách thức mà còn giúp người chơi hiểu rõ hơn về cách thức hoạt động của các thuật toán.
* Tạo map mê cung ngẫu nhiên: Điều này làm tăng độ đa dạng và tính mới mẻ cho mỗi lần chơi trò chơi
* Tính năng tùy chỉnh map: Người chơi có thể tạo ra những bản đồ theo ý thích của mình, điều này mở ra khả năng sáng tạo không giới hạn.

**+ Nhược điểm:**

* Thiếu tính năng tạo map theo từng độ khó: Điều này giới hạn trải nghiệm của người chơi ở các cấp độ khác nhau, nhưng bù lại người chơi có thể tự tạo map tùy thích của riêng mình.
* Chưa có chế độ chơi dành cho người chơi: Điều này làm giảm đi sự tương tác và hấp dẫn của trò chơi.
* Mã lập trình cần được tối ưu hơn: Điều này sẽ giúp trò chơi hoạt động mượt mà hơn và tăng trải nghiệm người dùng.

## 5.2. Hướng phát triển

Dựa trên những đánh giá trên, chúng em đề xuất một số hướng phát triển cho dự án trong tương lai:

* Nâng cấp giao diện người dùng: Mục tiêu là tạo ra một giao diện hấp dẫn, thân thiện và dễ sử dụng hơn.
* Tính năng chọn độ khó: Điều này sẽ làm trò chơi phù hợp với mọi đối tượng người chơi, từ người mới bắt đầu đến người chơi có kinh nghiệm.
* Cho phép nhập và chia sẻ map từ người chơi khác: Tính năng này khuyến khích sự tương tác cộng đồng và sáng tạo.
* Tối ưu hóa thuật toán: Mục đích là cải thiện hiệu suất và tăng độ khó, đồng thời giảm thiểu các lỗi trong quá trình chơi.
* Thêm chế độ chơi mới: Cung cấp các chế độ chơi mới để làm phong phú trải nghiệm của người chơi, ví dụ như chế độ player mode cho phép người chơi di chuyển trong mê cung và tự tìm lời giải.
* Tích hợp âm thanh: Âm thanh sẽ làm tăng độ chân thực và hấp dẫn cho trò chơi.

Nhìn chung, dự án của chúng em đã đạt được những thành tựu cơ bản, nhưng vẫn còn nhiều cơ hội để phát triển và cải thiện trải nghiệm người chơi. Chúng em mong muốn tiếp tục nghiên cứu và phát triển để mang đến một sản phẩm chất lượng và thú vị hơn nữa.

# Tài Liệu Tham Khảo

1. TonyPDavis. (2018). Python Breadth First Search Maze solving program. [Video]. Youtube. <https://youtu.be/ZuHW4fS60pc?si=TRe9BXlMW4oLqxIW>
2. "zit-software." (2023). zit-pathfinding [Giao diện, Mã nguồn]. GitHub. <https://github.com/zit-software/zit-pathfinding>
3. Vnoi\_Lê Minh Hoàng.(2021).Giải thuật BFS.<https://vnoi.info/wiki/algo/graph-theory/breadth-first-search.md>
4. Shinichi Lê.(2020).Giải Thuật A\*[. https://www.iostream.co/article/thuat-giai-a-DVnHj](.%20https:/www.iostream.co/article/thuat-giai-a-DVnHj%20)
5. Trần, Thơ. (2019). Thuật giải UCS. viblo. <https://viblo.asia/p/cac-thuat-toan-co-ban-trong-ai-phan-biet-best-first-search-va-uniform-cost-search-ucs-Eb85omLWZ2G>
6. Trần, Thơ. (2019). Thuật giải UCS. viblo. <https://viblo.asia/p/cac-thuat-toan-co-ban-trong-ai-phan-biet-best-first-search-va-uniform-cost-search-ucs-Eb85omLWZ2G>
7. Supernova. (2018). Giải thuật greedy <https://vnoi.info/wiki/translate/topcoder/Greedy-is-Good.md>
8. FPTEDU. (2020).Thuật toán dijkstra <https://aptech.fpt.edu.vn/thuat-toan-dijkstra.html>
9. Minh, Bui. (2020). Các Yếu Tố Lập Trình Game Cơ Bản Với Pygame. CodeLearn <https://codelearn.io/sharing/lap-trinh-game-co-ban-voi-pygame\>
10. Damian, A. (2018). Môi trường Spyder <https://ubunlog.com/vi/m%C3%B4i-tr%C6%B0%E1%BB%9Dng-ph%C3%A1t-tri%E1%BB%83n-tr%C4%83n-spyder/#comments>