

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN - NGÀNH TRÍ TUỆ NHÂN TẠO

BÁO CÁO ĐỒ ÁN CUỐI KÌ

Đề tài: **Xây dựng trò chơi Mummy Maze**

Môn học: Cơ sở lập trình cho Trí tuệ nhân tạo – CSC10013

Nhóm sinh viên thực hiện:

Nguyễn Châu Tuấn Kiệt (25122026)

Võ Duy Phát (25122013)

Phan Phước Quang Minh (25122029)

Nguyễn Minh Quang (25122038)

Giáo viên hướng dẫn:

Thầy Trần Hoàng Quân

Ngày 11 tháng 1 năm 2026



Mục lục

1	Giới thiệu đề tài và thông tin nhóm	1
1.1	Giới thiệu đề tài	1
1.2	Thành viên nhóm và Phân công nhiệm vụ	1
2	Các thuật toán đã tìm hiểu và triển khai trong game	2
2.1	Thuật toán Greedy (Tham lam)	2
2.1.1	Ý tưởng thuật toán	2
2.1.2	Quy ước Hệ tọa độ trong Game	3
2.1.3	Các bước thực hiện	3
2.1.4	Mã giả thuật toán Di chuyển (Greedy)	4
2.1.5	Ví dụ và chạy từng bước	5
2.1.6	Lưu ý mở rộng	6
2.2	Thuật toán BFS Tìm đường (Pathfinding BFS).	6
2.2.1	Ý tưởng thuật toán	6
2.2.2	Phạm vi áp dụng trong Game	7
2.2.3	Các bước thực hiện	7
2.2.4	Mã giả thuật toán (BFS)	8
2.2.5	Ví dụ và Chạy từng bước	9
2.2.6	Dánh giá thuật toán	10
2.3	Thuật toán BFS Kiểm chứng (Validation BFS)	10
2.3.1	Ý tưởng thuật toán	10
2.3.2	Phạm vi áp dụng trong Game	11
2.3.3	Các bước thực hiện	11
2.3.4	Mã giả thuật toán (BFS)	12
2.3.5	Ví dụ và Chạy từng bước	13
2.3.6	Dánh giá thuật toán	14
2.4	Thuật toán Sinh Mê cung (DFS Backtracking)	14
2.4.1	Ý tưởng thuật toán	14
2.4.2	Mã giả thuật toán (Maze Gen)	16
2.4.3	Ví dụ chạy mô phỏng	17

2.4.4	Dánh giá thuật toán	18
3	Kiến trúc chi tiết của game	19
3.1	Kiến trúc Tổng quan	19
3.2	Sơ lược Giao diện Game	19
3.2.1	Giao diện Menu và Hệ thống người dùng	19
3.2.2	Giao diện Thiết lập màn chơi	21
3.2.3	Giao diện Gameplay chính	22
3.2.4	Hệ thống Phản hồi trạng thái	22
3.2.5	Các màn hình chức năng phụ trợ	24
3.3	Các chức năng chi tiết của Game	26
3.3.1	Về Mê cung	26
3.3.2	Về Hệ thống Nhân vật	27
3.3.3	Các chức năng Hỗ trợ người chơi	29
3.4	Các Kịch bản của trò chơi	30
3.5	Vòng lặp Trò chơi (Game Loop)	31
4	Quá trình phát triển game	33
4.1	Giai đoạn 1: Phân tích và Thiết kế	33
4.2	Giai đoạn 2: Phát triển Core Game	33
4.3	Giai đoạn 3: Phát triển Thuật toán và Tính năng nâng cao	33
4.4	Giai đoạn 4: Kiểm thử và Tối ưu hóa	34
5	Những hạn chế và hướng phát triển	35
5.1	Các hạn chế hiện tại	35
5.2	Hướng phát triển trong tương lai	35
6	Tài nguyên và Trích dẫn	37
6.1	Tài nguyên Đồ họa và Âm thanh	37
6.2	Tham khảo Mã nguồn và Thuật toán	37
7	Tài liệu tham khảo	38

1 Giới thiệu đề tài và thông tin nhóm

1.1 Giới thiệu đề tài

- **Tên đề tài:** Xây dựng trò chơi *Mummy Maze*.
- **Mô tả:** Đồ án tái hiện tựa game giải đố kinh điển Mummy Maze của PopCap Games. Người chơi điều khiển nhà thám hiểm tìm đường thoát khỏi mê cung, tránh né xác ướp, quái vật và bẫy.
- **Công nghệ sử dụng:**
 - Ngôn ngữ: Python 3.14
 - Thư viện đồ họa: Pygame.
 - Lưu trữ dữ liệu: JSON (JavaScript Object Notation).

1.2 Thành viên nhóm và Phân công nhiệm vụ

Dưới đây là bảng phân công công việc chi tiết của nhóm:

STT	Họ và tên	Nhiệm vụ được giao	Tiến độ hoàn thành
1	Phan Phước Quang Minh	Maze & Gameplay Core: - Xây dựng hệ thống mê cung (logic bẫy, tường, khóa). - Quản lý trạng thái game. - Hệ thống tính điểm và Bảng xếp hạng.	100%
2	Võ Duy Phát	Nhân vật & Thuật toán: - Cài đặt Class Explorer và Mummy. - Triển khai thuật toán BFS, Greedy. - Logic AI nâng cao (Zone Defense).	100%
3	Nguyễn Minh Quang	Giao diện & Âm thanh: - Hiển thị chế độ Console (ASCII). - Hiệu ứng đồ họa (Animation). - Quản lý âm thanh và UI.	100%
4	Nguyễn Châu Tuấn Kiệt	Hệ thống & Hỗ trợ: - Chức năng Undo/Reset (Stack). - Hệ thống Save/Load (JSON). - Quản lý Level và Đăng nhập. - Tài liệu hướng dẫn.	100%

2 Các thuật toán đã tìm hiểu và triển khai trong game

Dựa trên yêu cầu đồ án và đặc thù của game Mummy Maze, nhóm tập trung triển khai **bốn** thuật toán cốt lõi để giải quyết các bài toán khác nhau trong game:

- **Greedy (Tham lam):** Xây dựng hành vi di chuyển cơ bản của quái vật (Mummy/Scorpion).
- **BFS Tìm đường (Pathfinding BFS):** Hỗ trợ quái vật tìm đường ngắn nhất đuổi bắt người chơi (AI nâng cao).
- **BFS Kiểm chứng (Validation BFS):** Duyệt toàn bộ không gian trạng thái để đảm bảo map sinh ra luôn có lời giải.
- **DFS Quay lui (Backtracking):** Thuật toán sinh cấu trúc mê cung ngẫu nhiên.

2.1 Thuật toán Greedy (Tham lam)

2.1.1 Ý tưởng thuật toán

Thuật toán Greedy (Tham lam) lựa chọn nước đi tối ưu cục bộ tại thời điểm hiện tại để giảm tối đa **khoảng cách Manhattan** ($|x_1 - x_2| + |y_1 - y_2|$) tới người chơi.

Tuy nhiên, để tăng tính chiến thuật và độ khó cho game, mã nguồn áp dụng cơ chế "**Ưu tiên Hướng**" (Directional Priority) khác nhau cho từng loại quái vật:

- **Quái vật Trắng (White Mummy/Scorpion):** Tuân theo quy tắc "**Ưu tiên Ngang**" (Horizontal First). Chúng luôn cố gắng di chuyển Trái/Phải để cùng cột (y) với người chơi trước. Chỉ khi đã cùng cột hoặc bị chặn tường, chúng mới di chuyển Lên/Xuống.
- **Quái vật Đỏ (Red Mummy/Scorpion):** Tuân theo quy tắc "**Ưu tiên Dọc**" (Vertical First). Ngược lại với loại Trắng, chúng ưu tiên di chuyển Lên/Xuống để cùng hàng (x) với người chơi trước, sau đó mới xét đến chiều ngang.

Sự kết hợp này khiến người chơi khó bắt bài hơn, vì khi bị truy đuổi bởi cả hai loại quái vật, người chơi sẽ bị dồn vào thế "gọng kìm" (một con chặn ngang, một con chặn dọc).

2.1.2 Quy ước Hệ tọa độ trong Game

Để đồng bộ với cách xử lý mảng 2 chiều trong Python và thư viện Pygame, nhóm sử dụng hệ tọa độ màn hình (Computer Graphics Coordinate System):

- **Gốc tọa độ (0, 0):** Nằm ở góc trên cùng bên trái.
- **Trục Dọc (x hoặc row):** Tăng dần từ trên xuống dưới.
- **Trục Ngang (y hoặc col):** Tăng dần từ trái sang phải.

Hệ quả: Để di chuyển về phía người chơi tại (0,0), quái vật cần giảm tọa độ, dẫn đến giá trị hiệu số Δ thường là số âm (Ví dụ: đi lên là giảm x , sang trái là giảm y).

2.1.3 Các bước thực hiện

Quy trình sau dựa trên hàm `move_greedy` trong file `characters.py`:

1. **Input:** Tọa độ quái vật (x_m, y_m), tọa độ người chơi (x_p, y_p), bản đồ (*Maze*).
2. **Tính khoảng cách:** Xác định chênh lệch tọa độ $\Delta x = x_p - x_m$ (Dọc) và $\Delta y = y_p - y_m$ (Ngang).

3. Xác định hướng ưu tiên:

- *Mummy Trắng (White):* Ưu tiên đi Ngang (Δy) trước. Nếu $\Delta y \neq 0$, thử đi sang Trái/Phải.
- *Mummy Đỏ (Red):* Ưu tiên đi Dọc (Δx) trước. Nếu $\Delta x \neq 0$, thử đi Lên/Xuống.

4. **Kiểm tra tính hợp lệ:** Trước khi di chuyển, kiểm tra xem hướng đi đó có bị chặn bởi Tường (*Wall*) hoặc Cổng đóng (*Closed Gate*) hay không.

5. Quyết định di chuyển:

- Nếu hướng ưu tiên hợp lệ \rightarrow Di chuyển và kết thúc lượt.
- Nếu hướng ưu tiên bị chặn \rightarrow Thử hướng còn lại (ví dụ: đang ưu tiên Ngang bị chặn thì chuyển sang thử Dọc).
- Nếu cả hai hướng đều bị chặn \rightarrow Dừng yên.

2.1.4 Mã giả thuật toán Di chuyển (Greedy)

Thuật toán tìm đường được cài đặt trong file `character.py`, thuộc lớp `enemy`, phương thức `move_greedy`.

1. Giải thích các thành phần:

- **Enemy:** Đối tượng quái vật (Mummy).
- **Explorer:** Đối tượng người chơi (mục tiêu cần đuổi theo).
- **Maze:** Bản đồ mê cung (để biết đâu là đường trống).
- **Gate:** Cánh cổng (đối tượng đặc biệt mà quái vật không thể đi xuyên qua khi đang đóng).
- **Uu_Tien_Ngang:** Đây là biến quyết định "tính cách" di chuyển của quái vật:
 - Nếu là **ĐÚNG**: Quái vật sẽ luôn nhìn sang trái/phải để tìm cách áp sát người chơi trước. Nếu bị chặn mới chịu đi dọc (Ví dụ: Mummy trắng).
 - Nếu là **SAI**: Quái vật ưu tiên tiến lên/lùi xuống trước. Nếu không đi dọc được mới chịu đi ngang (Ví dụ: Mummy đỏ).

2. Mã giả thuật toán:

HÀM `move_greedy(Enemy, Maze, Gate, Explorer, Uu_Tien_Ngang)`

{

// TRƯỜNG HỢP 1: Nếu là loại ưu tiên hướng NGANG (Mummy Trắng)

NẾU (`Uu_Tien_Ngang == ĐÚNG`) THÌ:

NẾU (Enemy và Explorer chưa cùng cột) THÌ:

Enemy thử bước sang Ngang một ô

NẾU (Đã bước đi thành công) THÌ KẾT_THÚC_LƯỢT

// Nếu bị chặn hoặc đã thẳng cột, quái vật sẽ chuyển sang đi Dọc

Enemy thử bước theo chiều Dọc một ô

// TRƯỜNG HỢP 2: Nếu là loại ưu tiên hướng DỌC (Mummy Đỏ)

NGƯỢC_LẠI (`Uu_Tien_Ngang == SAI`):

NẾU (Enemy và Explorer chưa cùng hàng) THÌ:

 Enemy thủ bước theo chiều Dọc một ô

NẾU (Đã bước đi thành công) THÌ KẾT_THÚC_LƯỢT

// Nếu bị chặn hoặc đã thắng hàng, quái vật sẽ chuyển sang đi Ngang

 Enemy thủ bước sang Ngang một ô

TRẢ_VỀ Trạng thái mới của Enemy

}

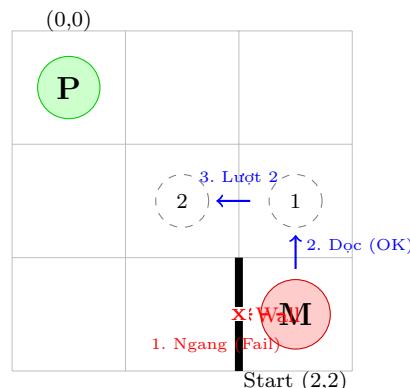
* **Ghi chú logic:** Thuật toán này thể hiện tính "tham lam" ở chỗ quái vật luôn thử hướng nó ưu tiên trước để thu hẹp khoảng cách nhanh nhất. Nó chỉ chấp nhận đi hướng còn lại nếu hướng ưu tiên bị vật cản chặn lại hoặc nó đã đứng đúng hàng/cột với người chơi.

* **Ghi chú:** Các hàm phụ trợ move_Horizontal và move_Vertical sẽ trực tiếp thay đổi tọa độ (x, y) bên trong đối tượng Enemy nếu nước đi hợp lệ.

Đánh giá thuật toán:

- **Độ phức tạp thời gian:** $O(1)$. Thuật toán chỉ thực hiện các phép so sánh tọa độ và phép toán cộng trừ cơ bản, không sử dụng vòng lặp tìm kiếm đường đi (như BFS/DFS).
- **Ưu điểm:** Tốc độ xử lý cực nhanh, phù hợp cho mức độ khó "Dễ" (Easy).
- **Nhược điểm:** Dễ bị người chơi "bẫy" vào góc tường vì không biết tìm đường vòng.

2.1.5 Ví dụ và chạy từng bước



Hình 1: Minh họa Mummy xử lý khi gặp tường (Cơ chế Fallback)

Tình huống: Mummy Trắng tại vị trí **(2, 2)**. Người chơi tại **(0, 0)**. Có một bức tường chấn ngang giữa ô **(2, 2)** và **(2, 1)**.

- **Lượt 1:**

- *Tính toán:* $\Delta y = 0 - 2 = -2$ (Cần sang Trái), $\Delta x = 0 - 2 = -2$ (Cần đi Lên).
- *Ưu tiên Ngang:* Mummy muốn sang Trái về ô **(2, 1)**.
- *Kiểm tra:* Gặp Tường chấn → **Thất bại** (Hàm `move_Horizontal` trả về vị trí cũ).
- *Fallback (Thủ hướng phụ):* Mummy chuyển sang thủ đi Lên (Đọc) về ô **(1, 2)**. Không có vật cản.
- *Hành động:* Mummy di chuyển đến **(1, 2)**.

- **Lượt 2:** (Vị trí mới: **(1, 2)**; Người chơi: **(0, 0)**)

- *Tính toán:* $\Delta y = -2$ (Vẫn cần sang Trái).
- *Ưu tiên Ngang:* Mummy thủ sang Trái về ô **(1, 1)**. Không có tường ngăn.
- *Hành động:* Mummy di chuyển đến **(1, 1)** và kết thúc lượt ngay (không cần xét đọc nữa).

2.1.6 Lưu ý mở rộng

Do đặc thù thiết kế hướng đối tượng (OOP) và kế thừa trong code:

- **Mummy Đỏ:** Sử dụng cùng thuật toán nhưng đảo ngược độ ưu tiên trong Mã giả (Thủ Dọc trước, Ngang sau).
- **Bọ cạp (Scorpion):** Lớp `scorpion_white` và `scorpion_red` kế thừa trực tiếp từ lớp Mummy, do đó chúng có hành vi tìm đường y hệt, chỉ khác biệt về tốc độ di chuyển trong vòng lặp chính (Game Loop).

2.2 Thuật toán BFS Tìm đường (Pathfinding BFS).

2.2.1 Ý tưởng thuật toán

Khác với thuật toán Greedy (chỉ lựa chọn tối ưu cục bộ), thuật toán **BFS (Breadth-First Search - Duyệt theo chiều rộng)** hoạt động dựa trên nguyên tắc "vết dầu loang". Thuật toán sử dụng

cấu trúc dữ liệu **Hàng đợi (Queue)** để duyệt tất cả các ô lân cận của vị trí hiện tại trước khi mở rộng ra xa hơn.

Mục tiêu chính của BFS trong đồ án này là tìm ra **đường đi ngắn nhất** từ Quái vật đến Người chơi, giúp AI giải quyết được các tình huống bị tường chắn mà Greedy không thể vượt qua.

2.2.2 Phạm vi áp dụng trong Game

Dựa trên phân tích mã nguồn của hàm `ai_move` thuộc class `enemy` trong file `character.py`, thuật toán BFS được áp dụng linh hoạt tùy theo độ khó:

- **Cấp độ Trung bình (Difficulty 2):** Quái vật sử dụng BFS hoàn toàn để truy đuổi người chơi (phương thức `move_smart_bfs`). AI sẽ luôn tìm được đường đến đích nếu có đường đi.
- **Cấp độ Khó (Difficulty 3):** Quái vật sử dụng chiến thuật kết hợp trong hàm `ai_move`:
 - *Chế độ Tấn công:* Khi khoảng cách đến người chơi ≤ 6 ô, AI chuyển sang dùng BFS để tối ưu hóa việc bắt người chơi.
 - *Chế độ Tuần tra:* Nếu người chơi ở xa, AI dùng BFS để tìm đường quay về vị trí phòng thủ (Cổng) hoặc đi tuần ngẫu nhiên.

2.2.3 Các bước thực hiện

Quy trình xử lý dựa trên hàm `bfs_find_next_step` trong file `character.py`:

1. **Input:** Tọa độ bắt đầu (Start), Tọa độ đích (Target), Bản đồ (Maze), Trạng thái Cổng (Gate).
2. **Khởi tạo:**
 - **Queue:** Chứa các đường đi đang xét (bắt đầu từ [Start]).
 - **Visited:** Tập hợp lưu các ô đã đi qua để tránh vòng lặp.
3. **Vòng lặp (Loop):** Trong khi Queue không rỗng:
 - Lấy đường đi đầu tiên ra khỏi Queue.
 - Xác định đỉnh cuối cùng (Current) của đường đi đó.

- **Kiểm tra đích:** Nếu Current trùng Target → Trả về bước đi thứ 2 trong đường dẫn (Next Step).

4. **Mở rộng (Expand):** Duyệt 4 ô xung quanh (Lên, Xuống, Trái, Phải).

5. **Kiểm tra hợp lệ:** Chỉ thêm vào Queue nếu ô đó:

- Nằm trong phạm vi bản đồ.
- Không phải là Tường (%).
- Không phải là Cổng đang đóng (Gate).
- Chưa nằm trong tập Visited.

2.2.4 Mã giả thuật toán (BFS)

Mã giả dựa trên hàm bfs_find_next_step trong file character.py:

```
HÀM BFS_Find_Next_Step(Start_Pos, Target_Pos, Maze, Gate)
```

```
{
```

```
// 1. Khởi tạo cấu trúc dữ liệu
```

```
Queue = [ [Start_Pos] ] // Hàng đợi chứa các đường dẫn
```

```
Visited = { Start_Pos } // Đánh dấu ô bắt đầu
```

```
// 2. Vòng lặp chính
```

TRONG KHI (Queue không rỗng) THỰC HIỆN:

```
Path = Queue.pop_left() // Lấy đường đi đầu tiên
```

```
Current = Path.last_node()
```

```
// Kiểm tra đích đến
```

NẾU (Current == Target_Pos) THÌ:

```
TRẢ VỀ Path[1] // Trả về bước đi tiếp theo
```

```
// Duyệt 4 hướng: Lên, Xuống, Trái, Phải
```

VỚI MỌI Next_Pos TRONG Neighbors(Current):

NẾU (IsValid(Next_Pos) VÀ Next_Posnotin Visited) THÌ:

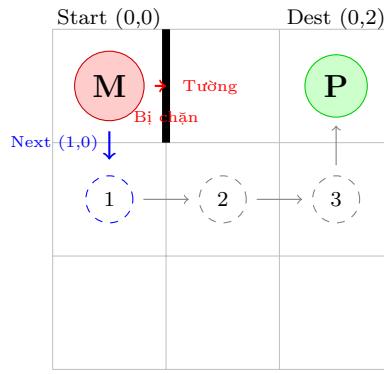
```

    Visited.add(Next_Pos)
    New_Path = Path + [Next_Pos]
    Queue.append(New_Path)

TRẦ_VỀ Start_Pos // Không tìm thấy đường, dừng yên
}

```

2.2.5 Ví dụ và Chạy từng bước



Hình 2: Minh họa đường đi BFS khi gặp tường chắn

Tình huống: Quái vật (M) tại **(0,0)**, Người chơi (P) tại **(0,2)**. Có một bức tường chẵn giữa ở **(0,1)**. Đây là tình huống mà Greedy sẽ thất bại (bị kẹt), nhưng BFS sẽ tìm ra đường vòng.

1. **Bước 1 (Khởi tạo):** Queue = {[$(0, 0)$]}. Visited = { $(0, 0)$ }.

2. **Bước 2 (Duyệt lần 1):**

- Lấy $(0, 0)$. Xét ô bên phải $(0, 1) \rightarrow$ Bị Tường chẵn.
- Xét ô bên dưới $(1, 0) \rightarrow$ Hợp lệ. Thêm vào Queue.

3. **Bước 3 (Duyệt lần 2):**

- Lấy đường dẫn đến $(1, 0)$. Từ đây xét ô bên phải $(1, 1) \rightarrow$ Hợp lệ.
- Queue lúc này: { $\dots, [(0, 0), (1, 0), (1, 1)]$ }.

4. **Bước 4 (Duyệt lần 3):**

- Từ $(1, 1)$ đi lên $(0, 1)$ bị chặn tường mặt kia.
- Di sang phải $(1, 2) \rightarrow$ Hợp lệ.

5. Bước 5 (Đến đích):

- Từ $(1, 2)$ đi lên $(0, 2) \rightarrow$ Trùng với vị trí Người chơi (P).
- **Kết luận:** Tìm thấy đường đi: $(0, 0) \rightarrow (1, 0) \rightarrow (1, 1) \rightarrow (1, 2) \rightarrow (0, 2)$.

6. Kết quả: Hàm trả về bước đi tiếp theo là **(1,0)** (Di xuống để tránh tường).

2.2.6 Đánh giá thuật toán

- **Độ phức tạp thời gian:** $O(R \times C)$ (với R, C là kích thước bản đồ). Trong trường hợp xấu nhất, thuật toán phải duyệt qua toàn bộ các ô trống.
- **Ưu điểm:**
 - Luôn tìm ra đường đi ngắn nhất (Optimal Path).
 - Giải quyết triệt để vấn đề "kẹt góc tường" của thuật toán Greedy.
- **Nhược điểm:** Tốn tài nguyên bộ nhớ (lưu Queue và Visited) và thời gian tính toán hơn so với Greedy, đặc biệt trên các bản đồ kích thước lớn.

2.3 Thuật toán BFS Kiểm chứng (Validation BFS)

2.3.1 Ý tưởng thuật toán

Khác với thuật toán BFS tìm đường cho AI (Mục 2.2) chỉ hoạt động trên bản đồ 2D tĩnh, thuật toán BFS trong module `search.py` thực hiện tìm kiếm trên **Không gian trạng thái** (State Space Search).

Thay vì xem mỗi ô trên bản đồ là một đỉnh đồ thị, thuật toán này coi một "Trạng thái" (State) của toàn bộ trò chơi là một đỉnh. Một trạng thái được định nghĩa là tổ hợp:

$$S = \{P_{(x,y)}, [E_1, E_2, \dots, E_n], Gate_{status}\}$$

Trong đó, thuật toán theo dõi đồng thời vị trí người chơi, vị trí của tất cả quái vật và trạng thái đóng/mở của Cổng. Tư duy của thuật toán là **mô phỏng tương lai**: "Nếu tôi đi bước này, quái vật sẽ đi đâu, và tôi có sống sót không?".

2.3.2 Phạm vi áp dụng trong Game

Thuật toán này đóng vai trò cốt lõi trong hai module:

- **Map Validator (Kiểm duyệt màn chơi):** Trong `maze_generator.py`, sau khi sinh ngẫu nhiên một map, thuật toán này được gọi để kiểm tra. Nếu trả về `False`, hệ thống sẽ hủy map đó vì người chơi không thể thắng.
- **Auto-Solver (Người giải tự động):** Đảm bảo luôn tồn tại ít nhất một chuỗi hành động dẫn đến chiến thắng (Winning Path) cho mọi màn chơi.

2.3.3 Các bước thực hiện

Quy trình dựa trên hàm BFS trong file `search.py`:

1. **Khởi tạo:** Dựa trạng thái ban đầu S_0 vào Hàng đợi (Queue) và tập đã duyệt (Visited).
2. **Kiểm tra đích:** Nếu vị trí người chơi trùng với ô Thoát hiểm ('S'), trả về kết quả tìm thấy đường.
3. **Phân nhánh (Branching):** Từ trạng thái hiện tại, thử 5 hành động: *Lên, Xuống, Trái, Phải, Dừng yên*.
4. **Mô phỏng (Simulation):**
 - Cập nhật vị trí Người chơi.
 - Kích hoạt logic di chuyển của **tất cả** Quái vật (gọi hàm `attempt_move`).
 - Kiểm tra va chạm: Nếu Người chơi bị bắt, nhánh này bị loại bỏ.
5. **Lưu trạng thái:** Nếu Người chơi còn sống và trạng thái mới chưa từng xuất hiện trong Visited, thêm vào Queue.

2.3.4 MÃ GIẢ THUẬT TOÁN (BFS)

MÃ GIẢ DỰA TRÊN HÀM BFS TRONG FILE `search.py`:

HÀM BFS(Explorer, MW, MR, SW, SR, Gate, Maze)

{

// 1. Khởi tạo

Start_Node = [Explorer, MW, MR, SW, SR, Gate]

Queue.push(Start_Node)

// 2. Vòng lặp chính

TRONG KHI (Queue không rỗng) THỰC HIỆN:

Current = Queue.pop() // Lấy trạng thái ra

// Kiểm tra điều kiện thắng

NẾU (Current.Explorer tại ô 'S') THÌ:

TRẢ VỀ Trace_Back(Current) // Tìm thấy lời giải

// Duyệt các hướng: Lên, Xuống, Trái, Phải

VỚI MỌI Dir TRONG [Up, Down, Left, Right]:

Tmp_State = Clone(Current)

// a. Check tướng chấn (eligible_character_move)

NẾU (Move_Valid(Dir)) THÌ:

// b. Mô phỏng chuyển động toàn cục

Is_Dead = attempt_move(Tmp_State, Dir)

NẾU (NOT Is_Dead VÀ Not_In_Queue(Tmp_State)) THÌ:

Queue.push(Tmp_State)

// Xử lý Đứng yên

Tmp_State = Clone(Current)

```

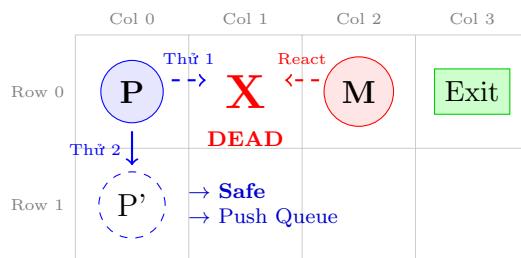
attempt_move(Tmp_State, Wait)
NẾU (Tmp_State ≠ Current) THÌ: // Chỉ thêm nếu trạng thái thay đổi
    Queue.push(Tmp_State)

TRẢ_VỀ NULL // Không giải được
}

```

2.3.5 Ví dụ và Chạy từng bước

Xét tình huống: Người chơi (P) tại (0, 0), Mummy (M) tại (0, 2). Đích tại (0, 3). P đi trước 1 bước, M đi sau 2 bước.



Phân tích chi tiết quá trình mô phỏng:

Tại trạng thái này, thuật toán BFS sẽ sinh ra các nhánh (Branching) tương ứng với các nước đi khả thi của người chơi và kiểm tra hậu quả của chúng:

- **Trường hợp 1: Người chơi thử đi sang Phải (Move Right)**
 - Người chơi di chuyển từ (0, 0) sang (0, 1). Khoảng cách với Mummy lúc này chỉ còn 1 ô.
 - *Phản ứng của Mummy:* Ngay lập tức, Mummy (từ 0, 2) di chuyển 1 bước sang trái về (0, 1).
 - *Kết quả:* Vị trí của Mummy và Người chơi trùng nhau tại (0, 1). Đây là trạng thái **Chết (Dead State)**. Do đó, thuật toán sẽ **cắt bỏ nhánh này**, không lưu vào hàng đợi.
- **Trường hợp 2: Người chơi thử đi Xuống (Move Down)**
 - Người chơi di chuyển từ (0, 0) xuống (1, 0) (Vị trí P' trong hình).

- *Phản ứng của Mummy:* Mummy (từ 0, 2) kích hoạt thuật toán Greedy để đuổi theo P' tại (1, 0). Do quy tắc ưu tiên trực ngang (Horizontal First), Mummy sẽ cố gắng đưa tọa độ cột của nó về cùng cột với người chơi (Cột 0) trước khi xét đến trực dọc.
- *Hành trình Mummy:*
 1. Bước 1: $(0, 2) \rightarrow (0, 1)$ (Sang trái).
 2. Bước 2: $(0, 1) \rightarrow (0, 0)$ (Sang trái tiếp).
- *Kết quả:* Sau lượt đi, Người chơi ở hàng dưới (1, 0) còn Mummy ở hàng trên (0, 0). Không xảy ra va chạm.
- *Kết luận:* Đây là trạng thái **An toàn (Safe State)**. Thuật toán sẽ lưu trạng thái mới này vào Queue để tiếp tục tìm đường từ vị trí (1, 0).

Thông qua cơ chế mô phỏng trước này, thuật toán BFS trong `search.py` có thể tránh được những nước đi "tự sát" (như sang phải) mà thuật toán tìm đường thông thường có thể mắc phải.

2.3.6 Đánh giá thuật toán

- **Độ phức tạp thời gian:** $O(b^d)$ với $b \approx 5$ (số hành động) và d là độ sâu lời giải. Do tính chất hàm mũ, thuật toán tồn tại nhiều tài nguyên tính toán.
- **Độ phức tạp không gian:** $O(b^d)$ do phải lưu trữ bản sao (Deep Copy) của toàn bộ màn chơi cho mỗi trạng thái trong tập Visited.
- **Kết luận:** Thuật toán đảm bảo tính **Đầy đủ (Complete)** và **Tối ưu (Optimal)**. Phù hợp để chạy ngầm (Background Task) khi sinh map, không phù hợp chạy thời gian thực liên tục.

2.4 Thuật toán Sinh Mê cung (DFS Backtracking)

2.4.1 Ý tưởng thuật toán

Để tạo ra một màn chơi ngẫu nhiên nhưng đảm bảo tính hợp lệ (không có khu vực bị cô lập), đồ án sử dụng thuật toán **DFS Quay lui (Recursive Backtracker)**.

Ý tưởng của thuật toán giống như một "người thợ đào hầm":

- Bắt đầu từ một ô bất kỳ, người thợ đào một đường đi sang ô hàng xóm chưa được thăm.

- Cứ tiếp tục đào sâu mãi (Depth-First) cho đến khi gặp ngõ cụt (không còn ô hàng xóm nào chưa thăm).
- Khi gặp ngõ cụt, người thợ sẽ **lùi lại (Backtrack)** theo đường cũ cho đến khi gặp một ngã ba có lối rẽ mới chưa đi, và tiếp tục đào từ đó.
- Quá trình kết thúc khi người thợ lùi về đúng điểm xuất phát.

2.4.2 Các bước thực hiện

Quy trình dựa trên lớp `MazeGenerator` trong file `maze_generator.py`:

1. **Khởi tạo:** Tạo lưới ô vuông đầy tường chẵn (%). Chọn một ô ngẫu nhiên làm điểm bắt đầu (*Current*). Dán dấu ô này là *Dã thăm (Visited)*. Đưa ô này vào Ngăn xếp (*Stack*).
2. **Vòng lặp chính (Tạo khung xương):** Trong khi Stack không rỗng:

- Lấy ô hiện tại (*Current*) từ đỉnh Stack.
- Tìm tất cả các ô hàng xóm (*Neighbors*) chưa được thăm của *Current*.
- **Nếu có hàng xóm chưa thăm:**
 - Chọn ngẫu nhiên một hàng xóm (*Next*).
 - Phá bỏ bức tường ngăn cách giữa *Current* và *Next*.
 - Dán dấu *Next* là *Dã thăm*.
 - Đẩy *Next* vào Stack.
- **Nếu không còn hàng xóm (Ngõ cụt):**
 - Loại bỏ *Current* khỏi Stack (Pop) để quay lui (Backtrack).

3. **Hậu xử lý (Tạo vòng lặp):**

- Thuật toán duyệt lại toàn bộ các bức tường còn lại trên bản đồ.
- Xác định các bức tường "tiềm năng" (nằm kẹp giữa hai ô trống theo chiều dọc hoặc ngang).

- Dựa trên tham số **Density** (Mật độ), chọn ngẫu nhiên một số lượng tường tiềm năng để phá bỏ. Bước này giúp tạo ra các đường đi tắt và vòng lặp, tránh việc bản đồ chỉ toàn ngõ cụt.

4. **Thiết lập đích:** Đặt ô thoát hiểm ('S') tại vị trí cố định (ví dụ: mép trái bản đồ) để đảm bảo tính hợp lệ cho thuật toán kiểm duyệt.

5. **Kết thúc:** Trả về ma trận mê cung hoàn chỉnh để sử dụng trong Game.

2.4.2 Mã giả thuật toán (Maze Gen)

Mã giả dựa trên lớp `MazeGenerator` trong file `maze_generator.py`:

```
HÀM Generate_Maze(Width, Height, Density)
{
    // 1. Khởi tạo lưới đầy tường
    Grid = Init_Grid(Width, Height, Wall=True)
    Start_Cell = Random(0, 0)
    Stack = [Start_Cell]
    Visited[Start_Cell] = True

    // 2. Vòng lặp đào hầm (DFS - Tạo khung xương)
    TRONG KHI (Stack không rỗng) THỰC HIỆN:
        Current = Stack.peek() // Xem ô trên đỉnh Stack
        Neighbors = Get_Unvisited_Neighbors(Current, Grid)

        NẾU (Neighbors không rỗng) THÌ:
            // a. Bước tới: Chọn hướng đi tiếp
            Next = Random_Choice(Neighbors)

            // b. Phá tường nối 2 ô
            Remove_Wall(Current, Next)
            Visited[Next] = True
```

```
Stack.push(Next)
```

KHÁC // Gặp ngõ cụt:

```
// c. Quay lui (Backtrack)
```

```
Stack.pop()
```

// 3. Hậu xử lý: Tạo vòng lặp (Dựa trên Density)

```
Potential_Walls = []
```

VỚI MỌI Tường TRONG Grid THỰC HIỆN:

NẾU (Tường kẹp giữa 2 ô trống) THÌ: Potential_Walls.add(Tường)

```
Shuffle(Potential_Walls)
```

```
Limit = Calculate(Length(Potential_Walls), Density)
```

VỚI i TỪ 0 ĐẾN Limit THỰC HIỆN:

```
Grid[Potential_Walls[i]] = Empty // Phá thêm tường
```

// 4. Thiết lập của ra

```
Grid[Exit_Pos] = 'S'
```

TRẢ VỀ Grid

```
}
```

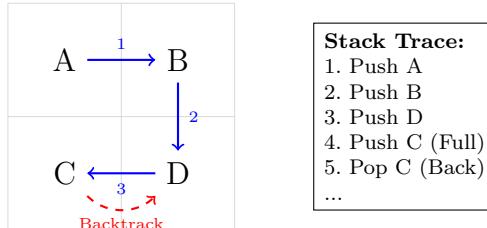
2.4.3 Ví dụ chạy mô phỏng

Xét lối nhỏ 2×2 (4 ô: A, B, C, D). Quá trình sinh mê cung diễn ra như sau:

Phân tích quá trình:

- **Bước 1 (Khởi tạo):** Bắt đầu tại ô **A** (Góc trên trái). Stack = [A].
- **Bước 2 (Tiến):** Từ A, chọn ngẫu nhiên sang **B** (Phải). Phá tường A-B. Stack = [A, B].
- **Bước 3 (Tiến):** Từ B, chỉ còn đường xuống **D**. Phá tường B-D. Stack = [A, B, D].
- **Bước 4 (Tiến):** Từ D, sang trái **C**. Phá tường D-C. Stack = [A, B, D, C].

- **Bước 5 (Ngõ cụt):** Tại C, các ô xung quanh (A, D) đều đã thăm. Không còn đường đi.
- **Bước 6 (Lùi):** Pop C, lùi về D. Pop D, lùi về B... cứ thế cho đến khi Stack rỗng.



Hình 3: Mô phỏng DFS Backtracking: Đường màu xanh là quá trình đào hầm (Push), đường màu đỏ đứt nét là quá trình quay lui (Pop) khi gặp ngõ cụt.

2.4.4 Đánh giá thuật toán

- **Ưu điểm:**

- Tạo ra *Perfect Maze*: Đảm bảo chỉ có duy nhất một đường đi giữa hai điểm bất kỳ, không có vòng lặp (Loops) và không có vùng cô lập.
- Dễ cài đặt bằng Dệ quy hoặc Stack.

- **Nhược điểm:**

- Mê cung có xu hướng tạo ra các hành lang dài và ngoằn ngoèo (Long corridors), độ khó giải đố thấp.
- **Giải pháp cải tiến:** Trong dự án, sau khi sinh xong, nhóm thực hiện thêm bước **Xóa tường ngẫu nhiên (Braiding)** để tạo thêm các vòng lặp, giúp game thú vị hơn và tránh việc người chơi dễ dàng đoán đường.

3 Kiến trúc chi tiết của game

3.1 Kiến trúc Tổng quan

Hệ thống được thiết kế dựa trên mô hình **MVC (Model-View-Controller)** để đảm bảo tính tách biệt giữa logic xử lý, dữ liệu và giao diện hiển thị. Cụ thể cấu trúc các module như sau:

- **Controller (Điều phối):** Module `main.py` đóng vai trò trung tâm, quản lý vòng lặp trò chơi (Game Loop), xử lý sự kiện đầu vào từ người dùng và điều phối các thành phần khác.
- **Model (Logic & Dữ liệu):**
 - `characters.py`: Xử lý logic di chuyển và thuật toán AI của quái vật (Greedy, BFS).
 - `database.py`: Quản lý dữ liệu người dùng, bảng xếp hạng và lưu trữ trạng thái game.
 - `maze_generator.py`: Thuật toán sinh màn chơi ngẫu nhiên kết hợp với module kiểm chứng `search.py`.
- **View (Giao diện):** Module `graphics.py` chịu trách nhiệm render hình ảnh (Sprite) và xử lý các hiệu ứng chuyển động (Animation).
- **Debug & Testing (Kiểm thử):** Module `ascii_game.py` cung cấp giao diện dòng lệnh (Console) giúp kiểm tra nhanh logic game và thuật toán tìm đường mà không phụ thuộc vào thư viện đồ họa.

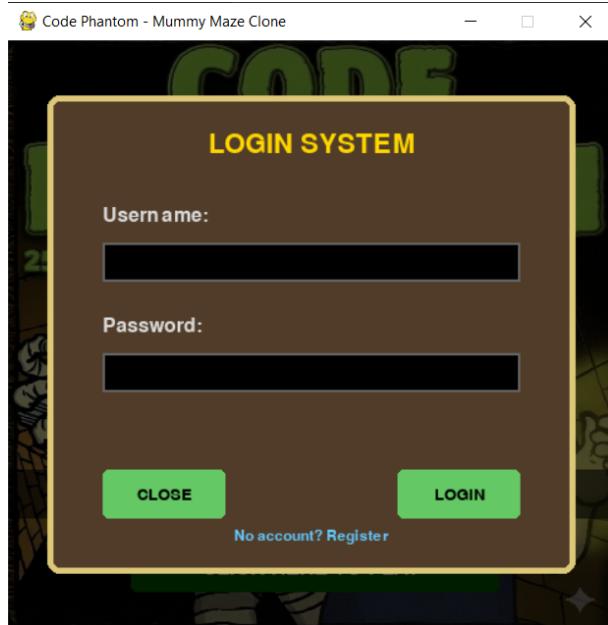
3.2 Sơ lược Giao diện Game

3.2.1 Giao diện Menu và Hệ thống người dùng

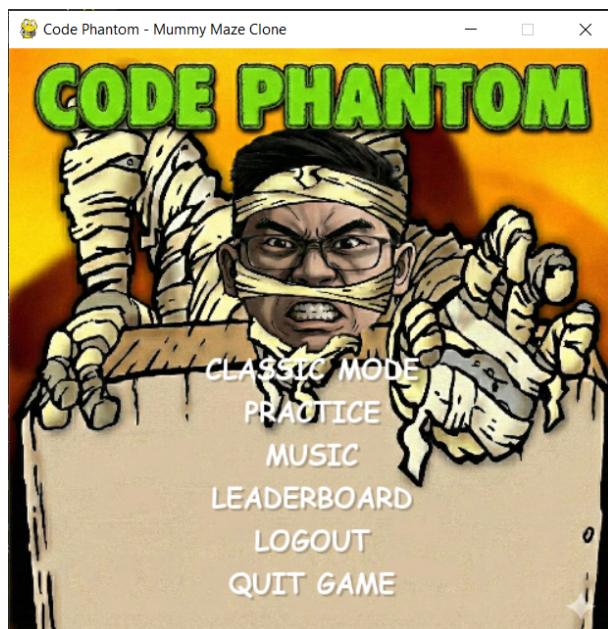
Giao diện người dùng (UI) được thiết kế theo phong cách tối giản với tông màu chủ đạo tạo cảm giác bí ẩn, phù hợp với chủ đề "Mummy Maze".

Màn hình Đăng nhập/Đăng ký: Đây là cổng giao tiếp đầu tiên, đảm bảo tính bảo mật và cá nhân hóa trải nghiệm. Hệ thống tích hợp các trường nhập liệu (Input Fields) cho Tên đăng nhập và Mật khẩu. Các nút điều hướng được bố trí rõ ràng, kèm theo cơ chế kiểm tra tính hợp lệ để ngăn chặn việc nhập thiếu thông tin hoặc sai định dạng trước khi gửi truy vấn xuống cơ sở dữ liệu.

Menu chính (Main Menu): Dóng vai trò là trung tâm điều hướng. Menu cung cấp các tùy chọn rõ ràng cho người chơi: tham gia chế độ cổ điển (Classic Mode), luyện tập (Practice), xem bảng xếp hạng hoặc tùy chỉnh cài đặt âm thanh. Hình ảnh nền và font chữ được cách điệu để tăng tính thẩm mỹ và thu hút người chơi ngay từ cái nhìn đầu tiên.



Hình 4: Màn hình Đăng nhập/Đăng ký



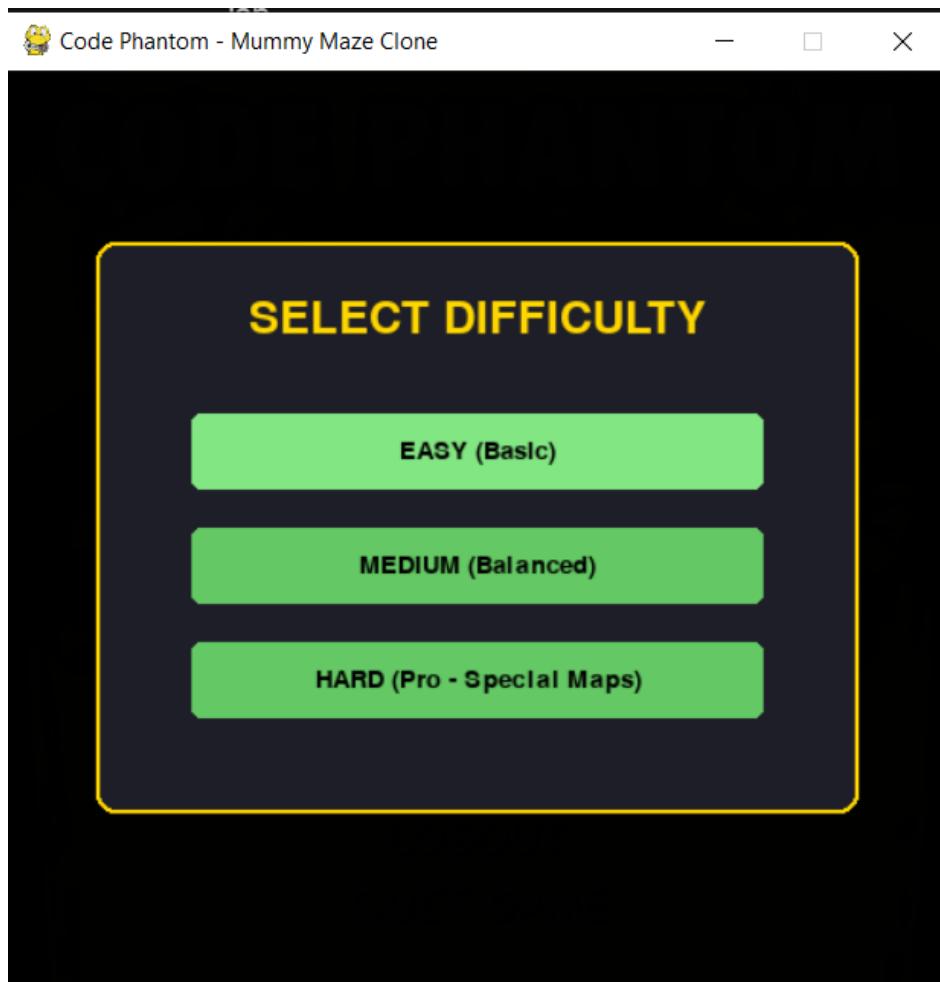
Hình 5: Menu chọn chế độ chơi

3.2.2 Giao diện Thiết lập màn chơi

Trước khi bước vào mê cung, người chơi được cung cấp quyền lựa chọn mức độ thử thách thông qua màn hình **Select Difficulty**. Hệ thống phân chia thành 3 cấp độ rõ rệt:

- **Easy (Basic)**: Dành cho người mới bắt đầu với bản đồ nhỏ, ít chướng ngại vật.
- **Medium (Balanced)**: Cân bằng giữa kích thước bản đồ và số lượng kẻ thù.
- **Hard (Pro - Special Maps)**: Thử thách tối đa với bản đồ lớn, cấu trúc mê cung phức tạp và kẻ thù thông minh hơn.

Việc phân chia này giúp game tiếp cận được đa dạng đối tượng người chơi, từ giải trí nhẹ nhàng đến những người thích tư duy thuật toán chuyên sâu.



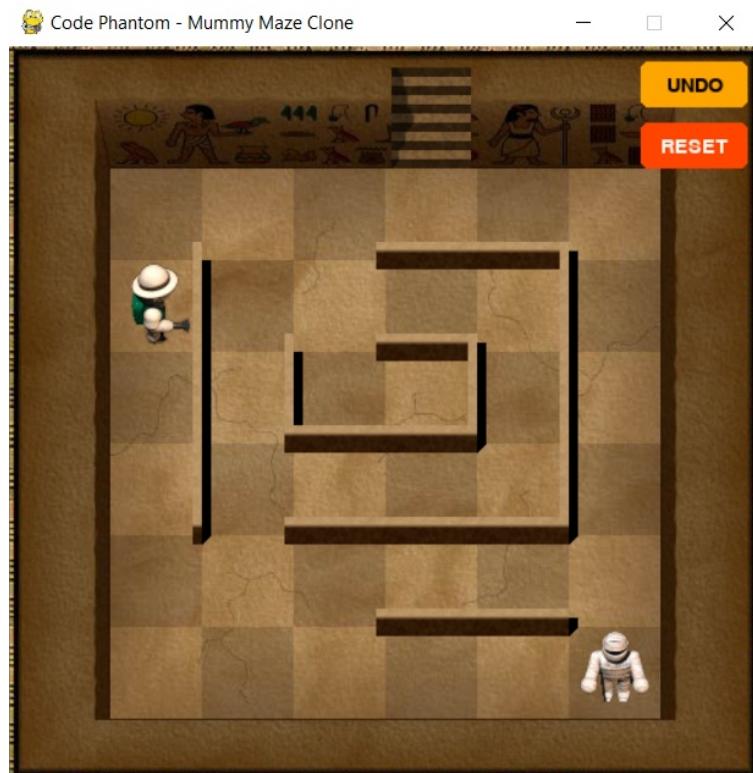
Hình 6: Giao diện tùy chọn độ khó trước khi vào game

3.2.3 Giao diện Gameplay chính

Màn hình chơi game là nơi người dùng tương tác nhiều nhất, được chia thành hai khu vực chính:

Khu vực bản đồ (Game Board): Chiếm phần lớn diện tích trung tâm, hiển thị mê cung dưới dạng lưới (Grid-based). Các đối tượng như Tường, Nhân vật (Explorer), Xác ướp (Mummy) và Cửa thoát hiểm được render trực quan bằng các sprite 2D.

Thanh công cụ chức năng (HUD - Heads-Up Display): Nằm ở góc trên bên phải, cung cấp các công cụ hỗ trợ người chơi như nút UNDO (quay lại nước đi trước - áp dụng cấu trúc dữ liệu Stack) và RESET (chơi lại màn hiện tại). Việc bố trí này giúp người chơi dễ dàng thao tác mà không bị che khuất tầm nhìn vào mê cung. Thông tin về màn chơi và trạng thái hiện tại cũng được cập nhật theo thời gian thực.



Hình 7: Giao diện màn chơi thực tế với các đối tượng tương tác

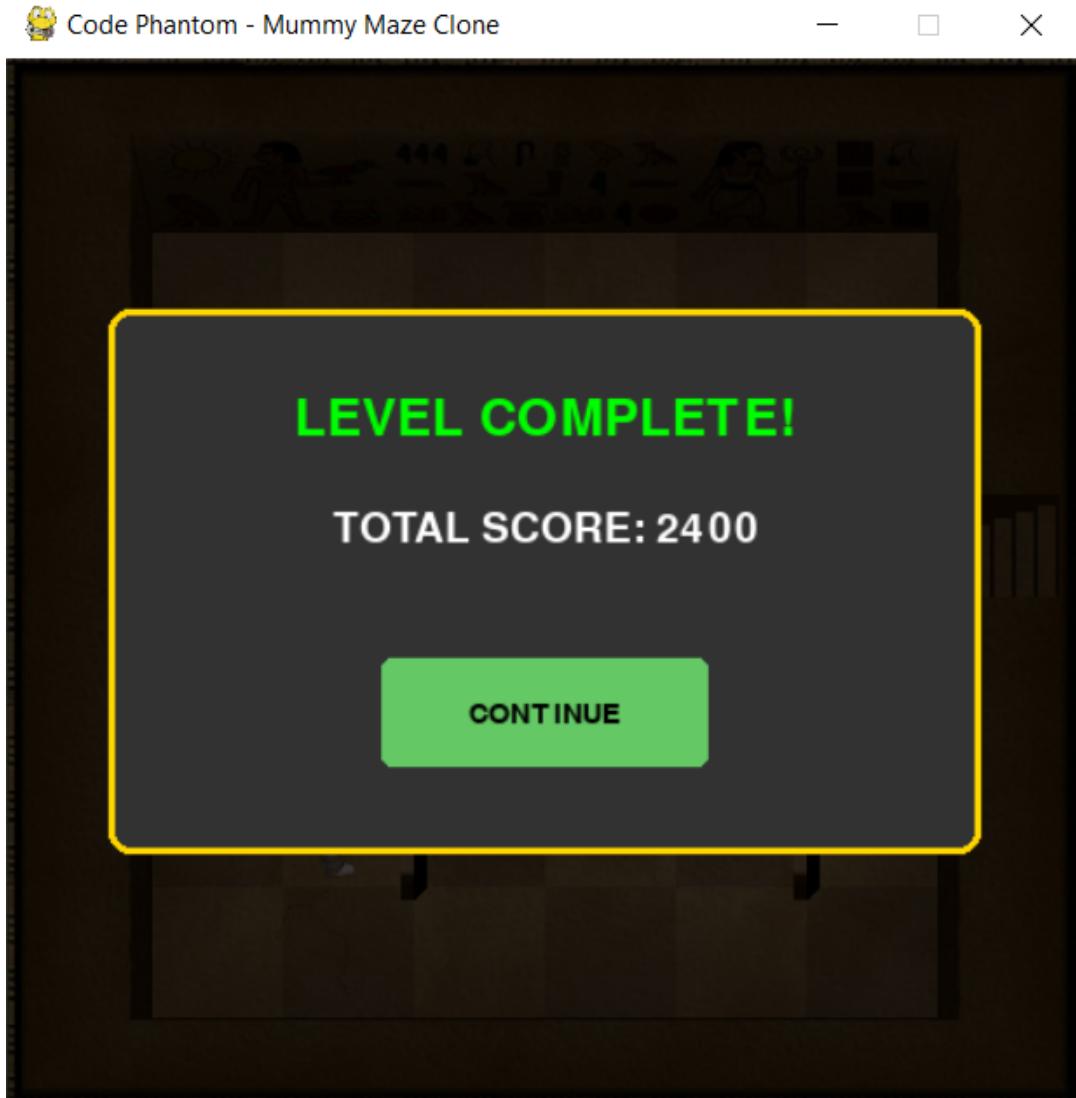
3.2.4 Hệ thống Phản hồi trạng thái

Hệ thống cung cấp phản hồi trực quan ngay lập tức khi màn chơi kết thúc, giúp người chơi nắm bắt được thành tích của mình:

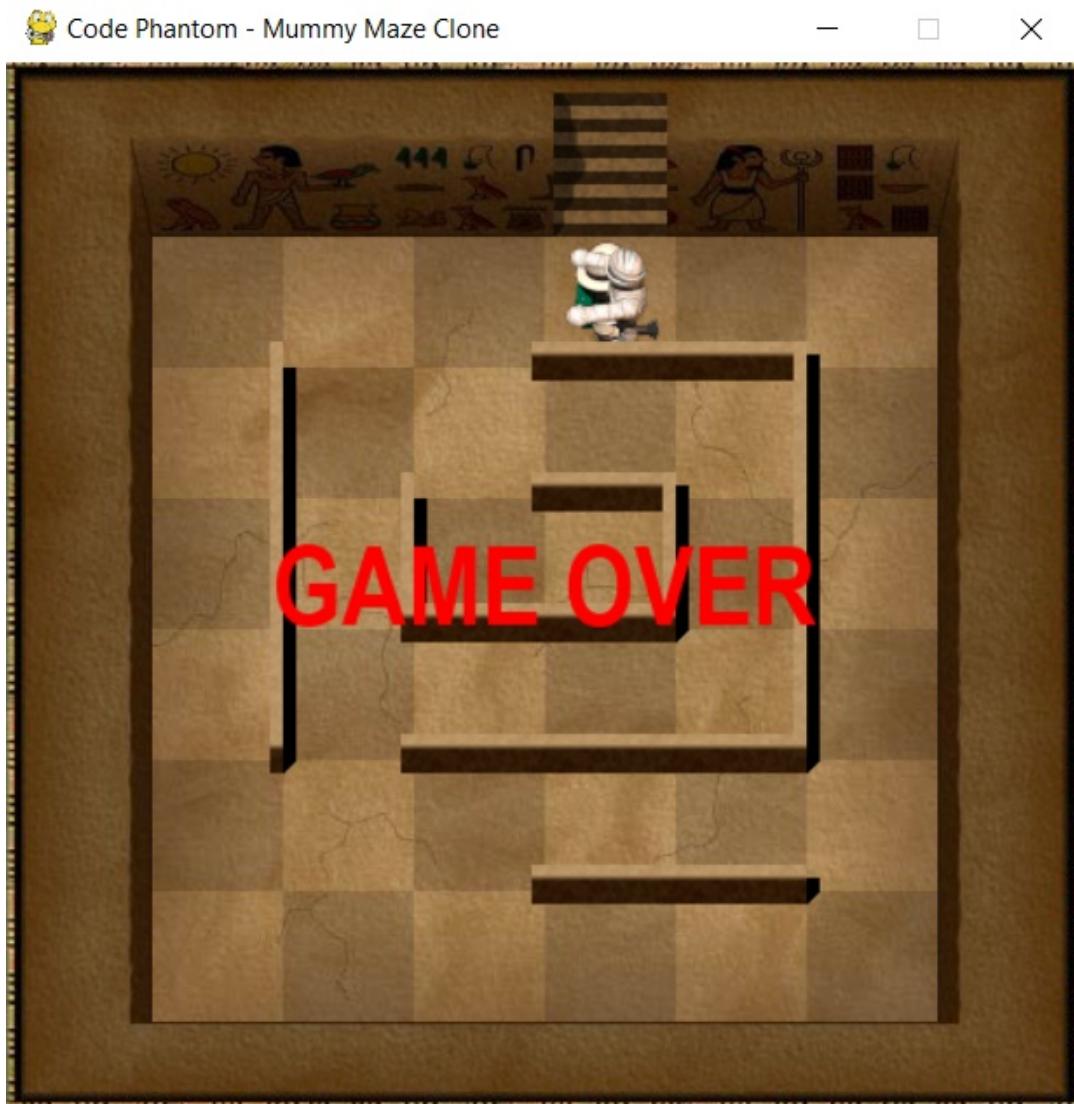
- **Màn hình Chiến thắng (Level Complete):** Xuất hiện khi người chơi thoát khỏi mê cung

thành công. Giao diện hiển thị thông báo chúc mừng nổi bật, tổng điểm đạt được (Total Score) và nút điều hướng để tiếp tục sang màn chơi kế tiếp (Continue).

- **Màn hình Thất bại (Game Over):** Xuất hiện khi người chơi va chạm với xác ướp hoặc bọ cạp. Dòng chữ "GAME OVER" màu đỏ được hiển thị trực tiếp trên nền màn chơi hiện tại, giúp người chơi nhận biết ngay vị trí mình đã mắc sai lầm.



Hình 8: Thông báo hoàn thành màn chơi



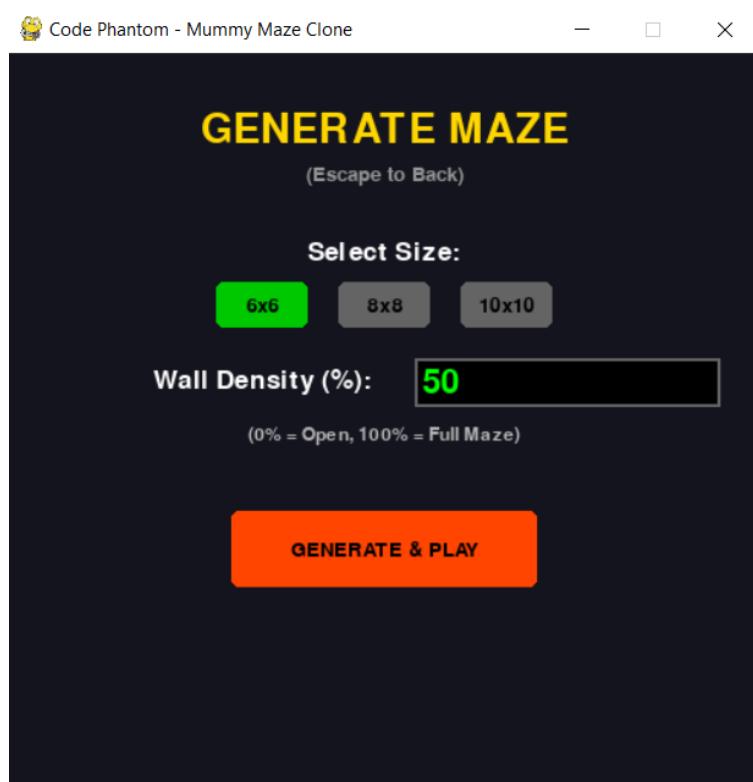
Hình 9: Thông báo thất bại khi người chơi va chạm với xác ướp hoặc bọ cạp

3.2.5 Các màn hình chức năng phụ trợ

Hệ thống tích hợp các giao diện nâng cao để mở rộng trải nghiệm người dùng:

Giao diện tùy chỉnh sinh Mê cung (Maze Generator): Cho phép người dùng can thiệp vào thuật toán sinh bản đồ. Người chơi có thể lựa chọn kích thước lưới (6x6, 8x8, 10x10) và mật độ tường (Wall Density). Giao diện này minh họa trực quan cho việc áp dụng thuật toán DFS/BFS trong việc tạo màn chơi ngẫu nhiên, đảm bảo tính đa dạng và không trùng lặp.

Bảng xếp hạng trực tuyến (Leaderboard): Hiển thị danh sách Top người chơi có điểm số cao nhất. Dữ liệu được truy xuất từ file hoặc cơ sở dữ liệu và hiển thị dưới dạng bảng gồm Thứ hạng, Tên người chơi và Điểm số, tạo động lực cạnh tranh giữa các người dùng.



Hình 10: Giao diện tùy chỉnh sinh Mê cung

RANK	PLAYER NAME	SCORE
#1	GUEST	2200
#2	1	100
#3	USERS	0
#4	BNBN	0
#5	FDFD	0

Hình 11: Bảng xếp hạng trực tuyến

3.3 Hệ thống Nhân vật và cách điều khiển

Hệ thống nhân vật được thiết kế dựa trên lớp cơ sở `Character` trong file `Character.py`, chia làm hai phe đối lập với cơ chế hành vi riêng biệt:

- **Người chơi (Explorer):**

- *Cơ chế*: Di chuyển theo lối ô vuông 4 hướng (Lên, Xuống, Trái, Phải). Mỗi lượt đi 1 bước.
- *Tương tác*: Thu thập Chìa khóa để mở Cổng và tìm đường đến Cầu thang thoát hiểm.

- **Kẻ thù (Enemy):**

- *Mummy Trắng*: Ưu tiên di chuyển theo trực **Ngang (Horizontal First)** khi sử dụng thuật toán tìm đường tham lam. Di chuyển 2 bước mỗi lượt (gấp đôi tốc độ người chơi).
- *Mummy Đỏ*: Ưu tiên di chuyển theo trực **Dọc (Vertical First)**. Di chuyển 2 bước mỗi lượt (gấp đôi tốc độ người chơi). Sự khác biệt này giúp hai quái vật có thể gọng kìm người chơi từ hai hướng khác nhau.
- *Bọ cạp (Scorpion)*: Biến thể của Mummy, thường được bố trí tại các nút giao quan trọng để tăng mật độ thử thách. Di chuyển 1 bước mỗi lượt (bằng với tốc độ người chơi).

3.3 Các chức năng chi tiết của Game

Dựa trên quá trình hiện thực hóa, các chức năng của trò chơi được chia thành hai nhóm: Cơ bản và Nâng cao

3.3.1 Vẽ Mê cung

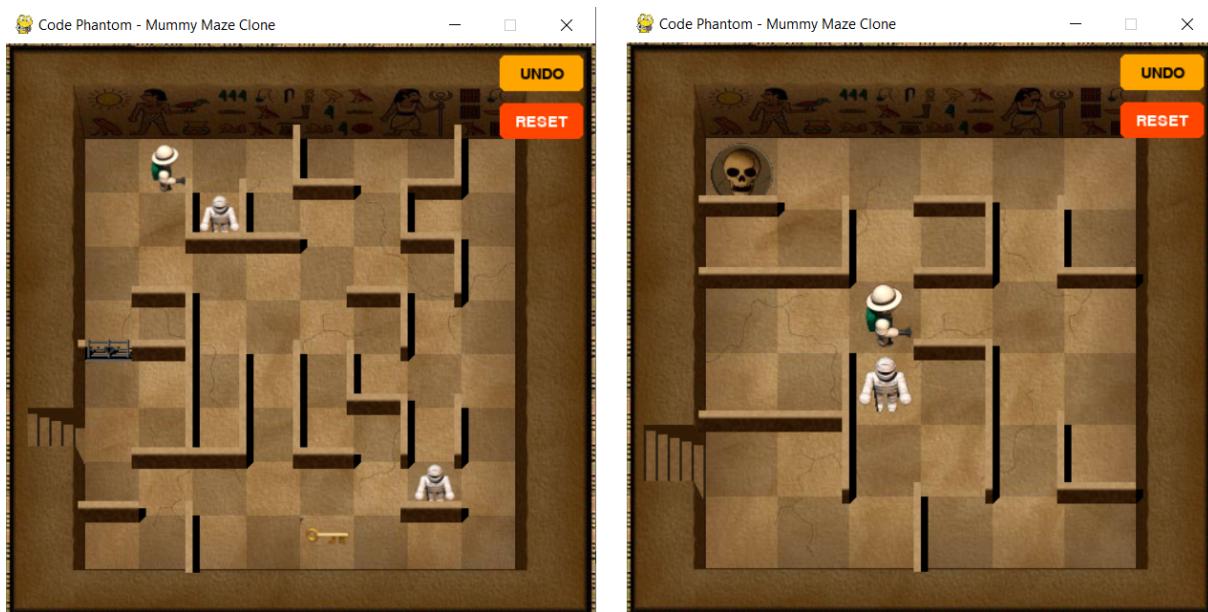
- **Chức năng Cơ bản:**

- Hệ thống hỗ trợ 3 kích thước bản đồ cố định: 6×6 , 8×8 , và 10×10 .
- Mê cung được thiết kế săn, đảm bảo tỷ lệ tường bên trong không vượt quá 50% kích thước mê cung

- Đảm bảo tính hợp lệ: Mê cung được thiết kế hoặc sinh ra luôn đảm bảo tồn tại ít nhất một đường đi từ vị trí khởi đầu đến lối ra (được kiểm chứng bởi module `search.py`).

- **Chức năng Nâng cao:**

- **Sinh Mê cung ngẫu nhiên (Random Generation):** Cho phép người chơi tự tạo màn chơi mới bằng thuật toán DFS. Người chơi có thể chọn một trong ba kích thước 6×6 , 8×8 , và 10×10 và *Mật độ tường (Wall Density)* để tăng độ khó. Chức năng này đã được giới thiệu ở phần **Các màn hình chức năng phụ trợ** trong mục **Sơ lược Giao diện Game**
- **Cơ chế Tương tác vật lý:**
 - * **Cổng và Chìa khóa (Gate & Key):** Cổng chắn đường sẽ đóng/mở luân phiên mỗi khi người chơi đi vào ô chứa Chìa khóa.
 - * **Bẫy (Traps):** Các ô bẫy cố định trên bản đồ sẽ tiêu diệt Nhà thám hiểm nếu bước vào.



Hình 12: Cổng và Chìa khóa ở hình bên trái và Bẫy ở hình bên phải

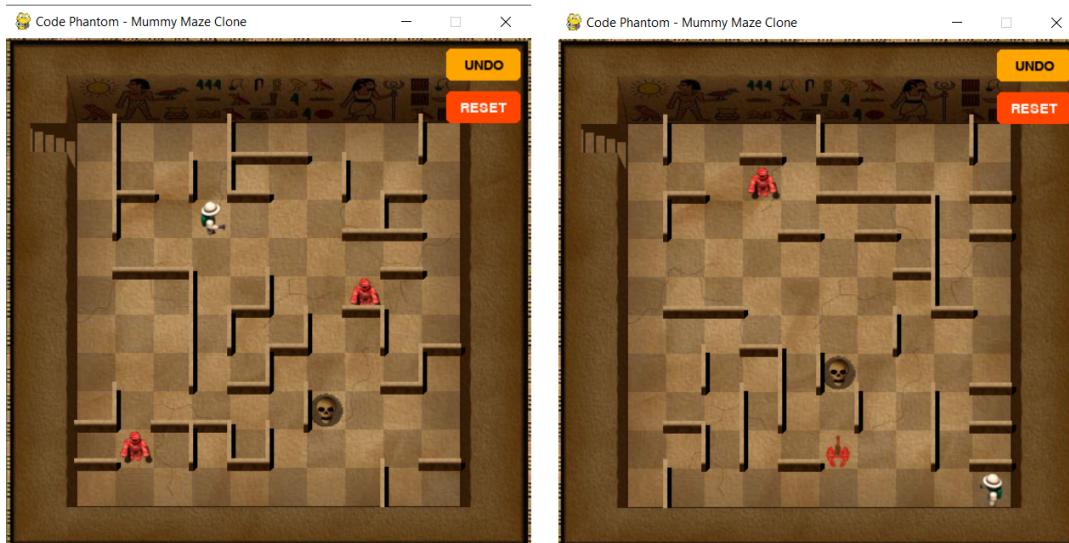
3.3.2 Về Hệ thống Nhân vật

- **Chức năng Cơ bản:**

- **Nhà Thám Hiểm (Explorer):** Di chuyển theo 4 hướng (Lên, Xuống, Trái, Phải) sử dụng bàn phím. Mỗi lượt đi 1 ô.
- **Kẻ thù (Enemies):** Xác ướp di chuyển 2 ô mỗi lượt (gấp đôi tốc độ người chơi).
- **Thuật toán AI cơ bản:** Ở độ khó *Easy*, kẻ thù sử dụng thuật toán *Greedy* (Tham lam) - ưu tiên di chuyển theo trực ngang hoặc dọc để rút ngắn khoảng cách với người chơi.

- **Chức năng Nâng cao:**

- **Điều khiển bằng Chuột :** Hỗ trợ người chơi click chuột vào hướng muốn đi để di chuyển thay vì dùng phím.
- **Đa dạng hóa Kẻ thù:** Ngoài Xác ướp (Mummy), game bổ sung thêm **Bọ cạp (Scorpion)** với hình ảnh hiển thị khác biệt.
- **Trí tuệ nhân tạo (AI) Nâng cao:**
 - * **Medium (BFS):** Kẻ thù sử dụng thuật toán Tìm kiếm theo chiều rộng để tìm đường ngắn nhất đuổi theo người chơi.
 - * **Hard (Zone Defense):** Kẻ thù biết cách "phòng thủ khu vực" - nếu người chơi ở xa, chúng sẽ tự động di chuyển về phía cửa thoát hiểm và đi tuần (Patrol) để chặn đầu.
- **Số lượng kẻ thù động:** Số lượng quái vật tăng dần theo kích thước bản đồ (1 quái cho map 6×6 , lên đến 3 quái cho map 10×10).

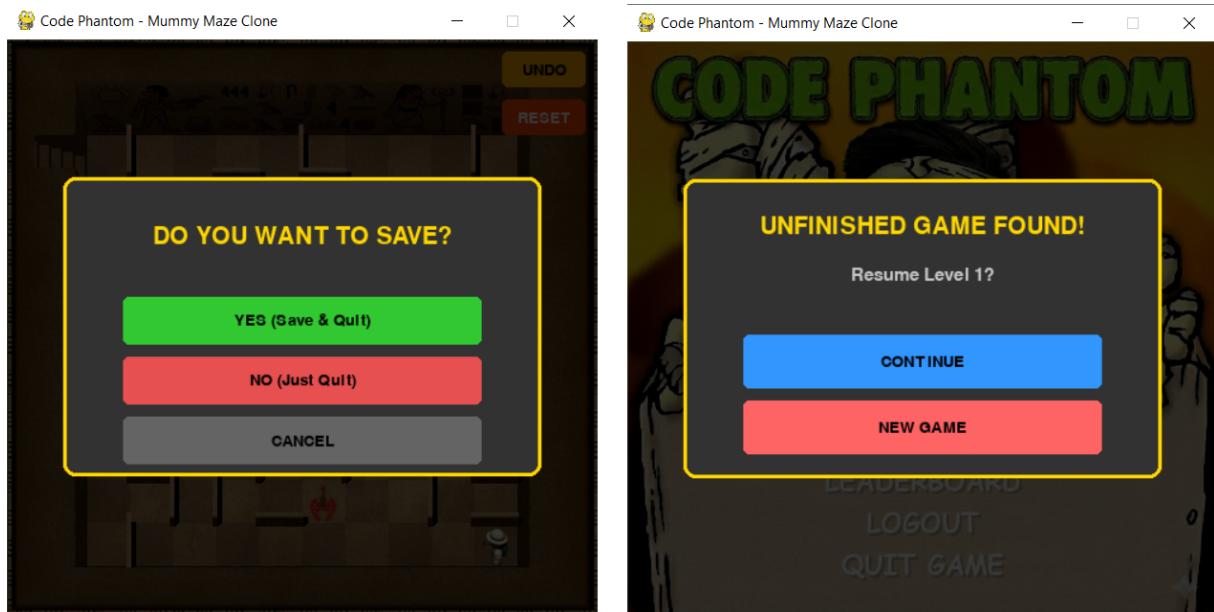


Hình 13: Tăng số lượng Xác ướp ở hình bên Trái và quái vật bổ sung Bọ cạp ở hình bên Phải

3.3.3 Các chức năng Hỗ trợ người chơi

- **Chức năng Cơ bản:**

- **Undo (Quay lui):** Sử dụng cấu trúc dữ liệu *Stack* để lưu trữ trạng thái, cho phép người chơi quay lại bước đi trước đó không giới hạn số lần.
- **Reset:** Cho phép chơi lại màn hiện tại từ đầu ngay lập tức.
- Hai chức năng **Undo & Reset** đã được giới thiệu ở phần **Giao diện Gameplay chính**
- **Save & Load Game:** Hệ thống sử dụng JSON để lưu trữ toàn bộ trạng thái màn chơi (vị trí tất cả nhân vật, trạng thái cổng, độ khó) để người chơi có thể tiếp tục sau khi thoát game.

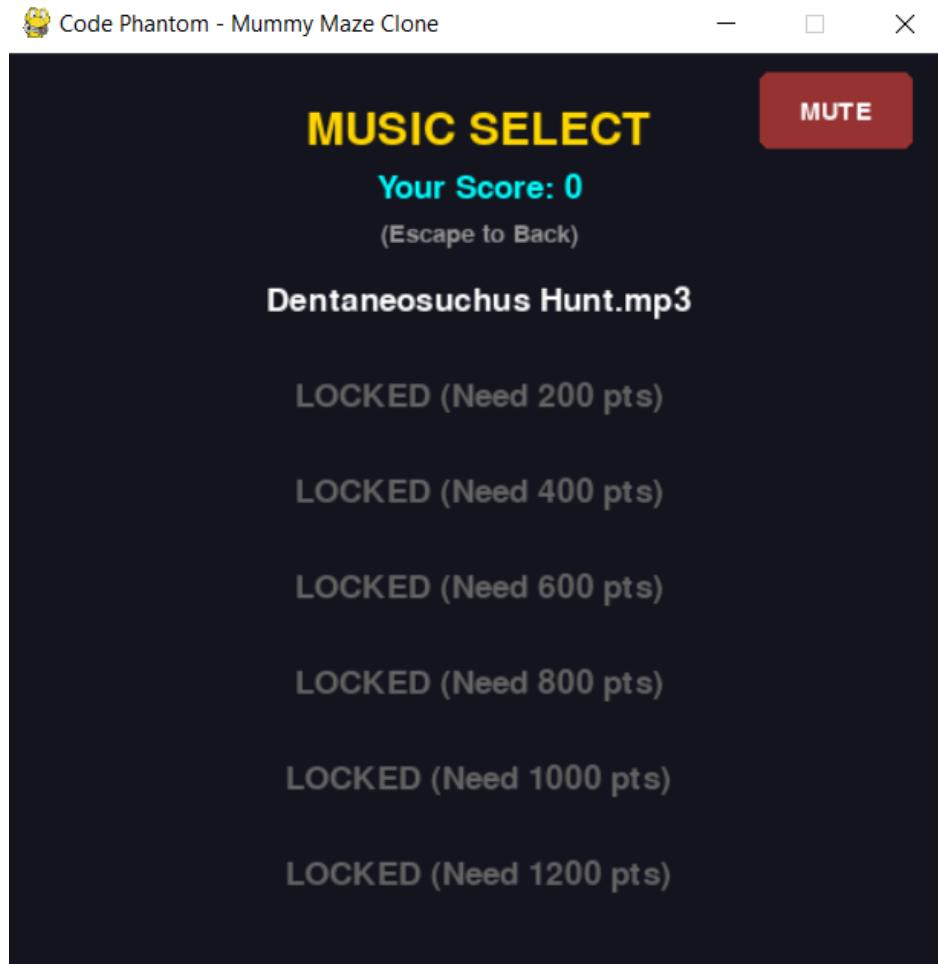


Hình 14: Quy trình Lưu trữ và Khôi phục trạng thái màn chơi (Save & Load System)

- **Chức năng Nâng cao:**

- **Hệ thống Tài khoản:** Người dùng có thể Đăng ký và Đăng nhập để lưu giữ dữ liệu cá nhân.
- **Bảng Xếp Hạng (Leaderboard):** Sau mỗi màn chơi, người chơi nhận được một số điểm nhất định. Hiển thị Top 7 người chơi có điểm số cao nhất, dữ liệu được cập nhật từ file `users_data.json`.

- Phần **Hệ thống Tài khoản** đã được giới thiệu thể hiện ở phần Đăng nhập/ Đăng kí và phần **Bảng Xếp Hạng (Leaderboard)** cũng đã được giới thiệu ở phần giao diện
- **Hệ thống Âm thanh & Mở khóa:** Tích hợp nhạc nền và hiệu ứng âm thanh. Các bản nhạc mới sẽ được mở khóa (Unlock) dựa trên điểm số tích lũy của người chơi (Threshold System).



Hình 15: Hệ thống mở khóa nội dung (Music/Level) dựa trên điểm tích lũy

3.4 Các Kịch bản của trò chơi

Hệ thống được thiết kế với các kịch bản tương tác và điều kiện kết thúc rõ ràng để đảm bảo tính logic của trò chơi giải đố:

- **Kịch bản Chiến thắng:**

- *Điều kiện:* Người chơi di chuyển thành công đến ô Cầu thang thoát hiểm ('S').

- *Hệ quả:* Hệ thống hiển thị thông báo "LEVEL COMPLETE", cộng điểm tích lũy (+100 điểm) vào cơ sở dữ liệu và mở khóa màn chơi tiếp theo.

- **Kịch bản Thua cuộc :**

- *Điều kiện:* Tọa độ của Người chơi trùng với tọa độ của bất kỳ Quái vật nào (Mummy/Scorpion) hoặc trùng với ô Bẫy ('T').
- *Hệ quả:* Màn hình hiển thị "GAME OVER", người chơi buộc phải chọn "RESET" để chơi lại màn hiện tại từ đầu.

- **Kịch bản Tương tác Giải đố:**

- *Cơ chế Chìa khóa (Key):* Khi người chơi hoặc quái vật di chuyển vào ô chứa Chìa khóa ('K'), trạng thái của Cổng sắt ('G') sẽ bị đảo ngược (Đang đóng → Mở, Đang mở → Đóng). Đây là cơ chế cốt lõi để tạo ra đường đi mới hoặc chặn quái vật.

- **Kịch bản Hỗ trợ :**

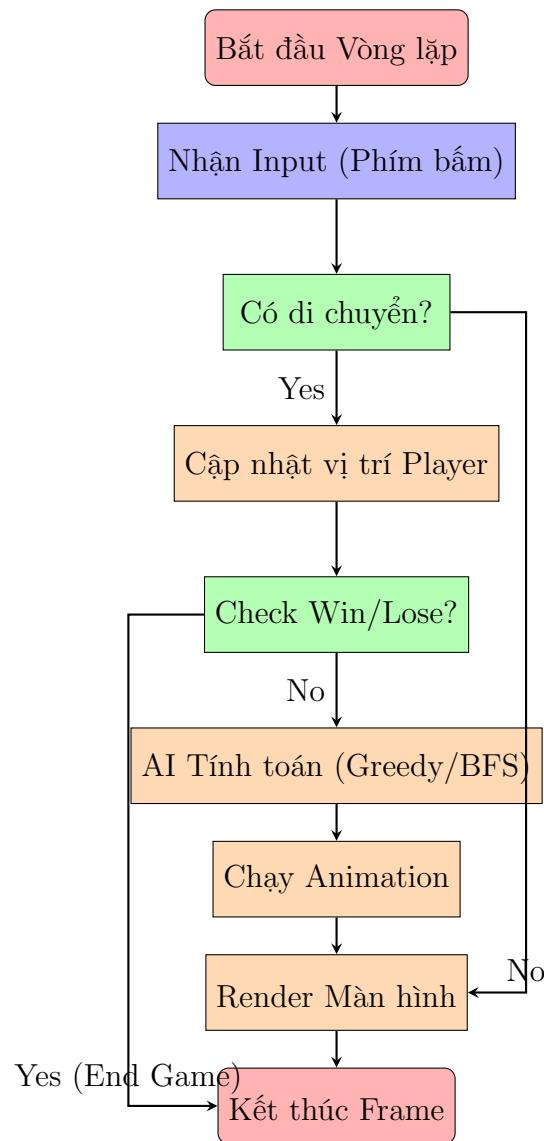
- *Tính năng Undo:* Người chơi có thể quay lại bước đi trước đó (sử dụng Stack) để sửa sai lầm mà không cần chơi lại từ đầu.
- *Lưu game (Save Game):* Khi thoát game giữa chừng, trạng thái hiện tại được lưu lại để người chơi có thể tiếp tục (Resume) vào lần sau.

3.5 Vòng lặp Trò chơi (Game Loop)

Vòng lặp trò chơi (Game Loop) trong module `main.py` chịu trách nhiệm duy trì hoạt động liên tục của ứng dụng. Quy trình xử lý trong một khung hình (Frame) tuân theo cơ chế **Turn-based** (Theo lượt) như sau:

- **Xử lý Input & Player:** Hệ thống nhận tín hiệu điều khiển, cập nhật vị trí người chơi và kiểm tra va chạm vật lý (tường, bẫy) ngay lập tức.
- **Kiểm tra Điều kiện dừng:** Xác định trạng thái Thắng (chạm đích) hoặc Thua (bị bắt) trước khi xử lý tiếp.

- **Lượt của AI:** Nếu người chơi di chuyển hợp lệ và chưa thua, hệ thống mới kích hoạt module AI để tính toán nước đi cho quái vật (dựa trên thuật toán BFS/Greedy).
- **Render:** Cuối cùng, module đồ họa sẽ xóa frame cũ và vẽ lại toàn bộ trạng thái mới lên màn hình.



Hình 16: Sơ đồ hoạt động của Vòng lặp trò chơi (Game Loop)

4 Quá trình phát triển game

Quá trình xây dựng trò chơi *Mummy Maze* được nhóm thực hiện qua 4 giai đoạn chính, tuân thủ quy trình phát triển phần mềm cơ bản: Phân tích, Thiết kế, Hiện thực hóa và Kiểm thử.

4.1 Giai đoạn 1: Phân tích và Thiết kế

Trong giai đoạn đầu, nhóm tập trung nghiên cứu cơ chế của tựa game gốc "Mummy Maze Deluxe" để xác định các yêu cầu cốt lõi:

- **Phân tích luật chơi:** Xác định cơ chế di chuyển theo lượt: Người chơi đi 1 bước, Quái vật đi 2 bước (hoặc theo quy luật riêng).
- **Lựa chọn công nghệ:** Quyết định sử dụng ngôn ngữ **Python** và thư viện **Pygame** vì khả năng xử lý đồ họa 2D tốt và dễ triển khai các thuật toán AI.
- **Kiến trúc hệ thống:** Thông nhất sử dụng mô hình **MVC** để tách biệt phần xử lý logic (nhân vật, thuật toán) khỏi phần hiển thị (đồ họa, âm thanh).

4.2 Giai đoạn 2: Phát triển Core Game

Đây là giai đoạn xây dựng khung xương sống cho trò chơi, tập trung vào các module nền tảng:

- **Xây dựng Engine cơ bản:** Hiện thực hóa vòng lặp trò chơi (Game Loop) trong `main.py` và hệ thống render hình ảnh trong `graphics.py`.
- **Xử lý va chạm và Di chuyển:** Xây dựng class `Explorer` và `Enemy` trong `characters.py`. Đảm bảo nhân vật không đi xuyên tường và tương tác đúng với các vật phẩm (Chìa khóa, Cổng).
- **Tích hợp Assets:** Cắt ghép các Spritesheet (nhân vật, tường, sàn) và tích hợp hệ thống âm thanh (nhạc nền, hiệu ứng).

4.3 Giai đoạn 3: Phát triển Thuật toán và Tính năng nâng cao

Sau khi gameplay cơ bản đã hoạt động, nhóm tiến hành tích hợp các thuật toán phức tạp - điểm nhấn của đồ án:

1. **Trí tuệ nhân tạo (AI):** Nâng cấp hành vi của quái vật từ di chuyển ngẫu nhiên sang thuật toán **BFS** (trong `characters.py`) để tự động tìm đường ngắn nhất đuổi theo người chơi ở độ khó cao.
2. **Sinh màn chơi ngẫu nhiên:** Phát triển module `maze_generator.py` sử dụng thuật toán **DFS Backtracking** để tạo ra các mê cung không trùng lặp.
3. **Module Kiểm chứng (Solver):** Xây dựng `search.py` để tự động kiểm tra tính giải được của map ngẫu nhiên trước khi đưa cho người chơi.
4. **Hệ thống dữ liệu:** Hoàn thiện `database.py` để lưu trữ thông tin người dùng, bảng xếp hạng và trạng thái Save/Load game bằng JSON.

4.4 Giai đoạn 4: Kiểm thử và Tối ưu hóa

Để đảm bảo sản phẩm ổn định, nhóm áp dụng hai phương pháp kiểm thử song song:

- **Console Testing (Kiểm thử logic):** Sử dụng module `ascii_game.py` để chạy game trên giao diện dòng lệnh. Cách này giúp debug nhanh các lỗi logic di chuyển và thuật toán tìm đường mà không bị ảnh hưởng bởi tải đồ họa.
- **Graphic Testing (Kiểm thử trải nghiệm):** Chạy thực nghiệm trên `main.py` để cân chỉnh độ khó (Easy/Medium/Hard), kiểm tra độ mượt của Animation và xử lý các lỗi giao diện (UI/UX).

5 Những hạn chế và hướng phát triển

Mặc dù nhóm đã hoàn thành các mục tiêu cơ bản đề ra ban đầu, hệ thống vẫn tồn tại một số hạn chế về mặt kỹ thuật và tính năng cần được khắc phục trong các phiên bản tiếp theo.

5.1 Các hạn chế hiện tại

- **Thuật toán sinh Mê cung chưa hoàn thiện:** Module `maze_generator.py` hiện tại chỉ dừng lại ở việc sinh cấu trúc tường và đường đi. Hệ thống chưa có logic tự động đặt vật phẩm quan trọng như **Chìa khóa (Key)** và **Cổng (Gate)** một cách ngẫu nhiên mà vẫn đảm bảo tính logic (Key phải nằm ở vùng tiếp cận được trước khi qua Gate). Do đó, chế độ *Practice Mode* hiện tại chưa có cơ chế giải đố Key-Gate. Ngoài ra, không thể chọn kích thước bất kỳ cho mê cung cũng là một hạn chế của game.
- **Hiệu suất thuật toán tìm đường:** Thuật toán BFS trong module `search.py` đảm bảo tìm ra đường đi ngắn nhất, nhưng có độ phức tạp không gian lớn ($O(b^d)$). Với các bản đồ kích thước lớn (ví dụ: 20x20 trở lên) hoặc số lượng quái vật tăng lên, việc duyệt toàn bộ không gian trạng thái sẽ gây ra độ trễ (latency) đáng kể.
- **Cấu trúc mã nguồn:** File `main.py` hiện đang đảm nhiệm quá nhiều vai trò (God Class): vừa quản lý logic game, vừa xử lý giao diện UI, vừa quản lý Save/Load. Điều này gây khó khăn cho việc bảo trì và mở rộng sau này.
- **Giới hạn hiển thị:** Các thông số tọa độ trong `graphics.py` đang được thiết lập cố định (Hardcoded) cho độ phân giải màn hình cụ thể. Game chưa hỗ trợ tính năng thay đổi kích thước cửa sổ (Resize) hoặc chế độ toàn màn hình (Fullscreen) linh hoạt.

5.2 Hướng phát triển trong tương lai

Để hoàn thiện sản phẩm thành một tựa game thương mại hoàn chỉnh, nhóm đề xuất các hướng phát triển sau:

1. Nâng cấp Thuật toán AI:

- Thay thế BFS bằng thuật toán **A* (A-Star)** với hàm Heuristic phù hợp để tối ưu hóa tốc độ tìm đường.

- Áp dụng thuật toán **Minimax** hoặc **Reinforcement Learning** để quái vật có khả năng phối hợp "bẫy" người chơi thay vì chỉ đuổi theo độc lập.
2. **Cải tiến Procedural Content Generation (PCG):** Phát triển thuật toán sinh map nâng cao: sau khi tạo mê cung, hệ thống sẽ phân tích các điểm thắt nút (choke points) để đặt Cổng và rải Chìa khóa vào các ngõ cụt xa nhất, tăng độ khó cho màn chơi. Ngoài ra, việc sinh ra các mê cung với kích thước ngẫu nhiên sẽ là một hướng phát triển tốt.
 3. **Mở rộng tính năng Online:** Thay thế hệ thống lưu trữ JSON cục bộ bằng cơ sở dữ liệu đám mây (như Firebase hoặc MySQL). Điều này cho phép tính năng **Bảng xếp hạng toàn cầu (Global Leaderboard)** và đồng bộ hóa tiến độ chơi trên nhiều thiết bị.
 4. **Tối ưu hóa Kiến trúc (Refactoring):** Tách module `main.py` thành các lớp nhỏ hơn như `UIManager`, `GameManager`, `SoundManager` theo đúng chuẩn thiết kế để mã nguồn gọn gàng và dễ mở rộng.

6 Tài nguyên và Trích dẫn

Để hoàn thiện đồ án này, nhóm đã sử dụng và tham khảo các tài nguyên sau đây.

6.1 Tài nguyên Đồ họa và Âm thanh

Loại tài nguyên	Mô tả	Nguồn/Tác giả
Sprites	Nhân vật Mummy, Explorer	<i>Unity Asset Store</i>
Tileset	Hình ảnh tường, sàn gạch	<i>OpenGameArt.org</i>
Sound FX	Tiếng bước chân, tiếng va chạm	<i>Freesound.org</i>
UI Design	Các nút bấm, Menu	Tự thiết kế

Bảng 1: Danh sách các tài nguyên đa phương tiện được sử dụng.

6.2 Tham khảo Mã nguồn và Thuật toán

- **Thuật toán BFS (Search):** Tham khảo logic duyệt trạng thái từ mã nguồn của tác giả *Pham Manh Tien* ([Github : https://github.com/tienpm/MummyMazeDeluxeSolver](https://github.com/tienpm/MummyMazeDeluxeSolver)) và tùy biến lại để phù hợp với logic game hiện tại.
- **Thuật toán Backtracking (Maze Gen):** Dựa trên lý thuyết thuật toán DFS được mô tả tại trang *GeeksForGeeks*.
- **Thư viện hỗ trợ:** Dự án sử dụng thư viện Pygame để xử lý đồ họa và Numpy để xử lý ma trận.

7 Tài liệu tham khảo

- [1] R. Nystrom, *Game Programming Patterns*. Genever Benning, 2014.
(Tham khảo kiến trúc: *Game Loop, Update Method*).
- [2] W. McGugan, *Beginning Game Development with Python and Pygame*, Apress, 2007.
(Tham khảo kỹ thuật: *Sprite Management, Event Handling*).
- [3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2020.
(Cơ sở lý thuyết: *Breadth-First Search (BFS) và Heuristic Search*).
- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.
(Cơ sở lý thuyết: *Graph Traversal, DFS, Stack*).
- [5] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*. Springer, 2016.
(Tham khảo phương pháp sinh màn chơi: *Constructive Generation*).
- [6] PopCap Games, *Mummy Maze Deluxe*. Electronic Arts, 2002. [PC Game].
(Nguồn cảm hứng: *Gameplay Mechanics, Level Design*).
- [7] Google, *Gemini*. [Large language model]. 2026.
(Công cụ hỗ trợ: *Gợi ý ý tưởng, kiểm tra lỗi logic và soạn thảo văn bản*).