TRƯỜNG ĐH SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

TIỂU LUẬN CUỐI KỲ

Môn học: Trí tuệ nhân tạo

Áp dụng các thuật toán tìm kiếm cổ điển để giải bài toán mê cung trong game Maze Explorer

GVHD: PGS.TS. Hoàng Văn Dũng

Danh sách nhóm sinh viên thực hiện

MSSV	Họ tên	Mức độ đóng góp
23110110	Lê Quang Hưng	100%
23110078	Nguyễn Thái Bảo	100%
23110111	Lương Nguyễn Thành Hưng	100%

TP. Hồ Chí Minh, tháng 10 năm 2025

LỜI CẢM ƠN

Trước hết, nhóm chúng em xin gửi lời cảm ơn chân thành và sâu sắc đến thầy Hoàng Văn Dũng, người đã tận tình hướng dẫn, định hướng và truyền đạt những kiến thức quý báu cho chúng em trong suốt quá trình thực hiện đề tài.

Trong suốt quá trình học tập và nghiên cứu, thầy không chỉ giúp chúng em củng cố kiến thức chuyên môn mà còn rèn luyện cho chúng em phương pháp tư duy, tác phong làm việc khoa học, cũng như tinh thần trách nhiệm trong học thuật. Những góp ý và sự chỉ dẫn tận tâm của thầy chính là nguồn động lực to lớn giúp nhóm chúng em hoàn thành đề tài này.

Mặc dù đã có nhiều cố gắng nhưng đề tài khó tránh khỏi những thiếu sót. Nhóm chúng em rất mong nhận được sự chỉ bảo và góp ý thêm của quý thầy để chúng em có thể hoàn thiện hơn trong việc học tập và nghiên cứu sau này. Xin trân trọng cảm ơn.

BẢNG PHÂN CÔNG CÔNG VIỆC

MSSV	Họ tên	Nhiệm vụ
23110110	**Lê Quang Hưng**	- Xây dựng Game Environment dùng thư viện Pygame - Viết lời cảm ơn - Viết cơ sở lý thuyết dùng để thực hiện project. - Hoàn thiện báo cáo
23110078	**Nguyễn Thái Bảo**	- Viết phần phát biểu bài toán. - Viết nội dung về các thuật toán tìm kiếm mù (BFS, DFS, UCS). - Tích hợp các thuật toán tìm kiếm mù vào game.
23110111	**Lương Nguyễn Thành Hưng**	- Viết mục đích và yêu cầu cần thực hiện, phạm vi và đối tượng nghiên cứu. - Viết nội dung về các thuật toán tìm kiếm có thông tin (Greedy, A*). - Tích hợp các thuật toán tìm kiếm có thông tin vào game.

MỤC LỤC

DANH MỤC CÁC TỪ / CỤM TỪ VIẾT TẮT

BFS	Breadth-First Search (Tìm kiếm theo chiều rộng)
FIFO	First In First Out
DFS	Depth-First Search (Tìm kiếm theo chiều sâu)
LIFO	Last In First Out
UCS	Uniform Cost Search (Tìm kiếm theo chi phí đồng nhất)
MST	Minimum Spanning Tree (Cây khung nhỏ nhất)

DANH MỤC HÌNH ẢNH

CHƯƠNG 1: MÔ TẢ BÀI TOÁN

1.1. Phát biểu bài toán

Bài toán mê cung là bài toán quen thuộc được dùng nhiều trong việc ứng dụng trí tuệ nhân tạo. Khi áp dụng các thuật toán trong việc tìm đường đi sẽ giúp ta có cái nhìn trực quan, phân tích được từng điểm mạnh điểm yếu của các thuật toán. Trong tiểu luận cuối kì môn Trí tuệ nhân tạo nhóm em chọn game Maze Explorer để ứng dụng các thuật toán BFS, DFS, UCS, Greedy và A*, có phát triển thêm ngoài việc đi đến đích thì bắt buộc phải ăn tất cả các ngôi sao, từ đó phân tích các số liệu nhận được để so sánh các giải thuật AI trong cùng một hoàn cảnh

Trò chơi gồm các ô tường không thể đi qua, các ô đường đi có thể đi qua, vị trí đứng khi bắt đầu game là S, vị trí của lối thoát là G, các ngôi sao sẽ có kí hiệu là "*". Nhân vật sẽ di chuyển theo bốn hướng lên, xuống, trái, phải, thu thập đầy đủ các ngôi sao để lối thoát sáng lên và đi đến lối thoát, phạm vi di chuyển là các ô đường đi có thể đi qua.

1.2. Mục đích và yêu cầu cần thực hiện

Mục đích chính của tiểu luận là áp dụng các thuật toán tìm kiếm cổ điển trong trí tuệ nhân tạo để giải quyết bài toán mê cung, đồng thời mở rộng yêu cầu bằng việc thu thập tất cả các ngôi sao trước khi đến lối thoát. Qua đó, nhóm nhằm minh họa cách các thuật toán như BFS, DFS, UCS, Greedy và A* giúp phân tích ưu nhược điểm của từng thuật toán như hiệu suất thời gian, bộ nhớ sử dụng, độ dài đường đi tối ưu và khả năng xử lý các mê cung phức tạp. Tiểu luận cũng hướng đến việc xây dựng một trò chơi đơn giản nhưng trực quan, hỗ trợ người dùng trải nghiệm và so sánh kết quả giữa các thuật toán, từ đó nâng cao hiểu biết về các thuật toán cổ điển là nền tảng của AI hiện đại.

Yêu cầu cần thực hiện bao gồm:

Xây dựng giao diện trò chơi Maze Explorer bằng ngôn ngữ lập trình Python (sử dụng thư viện Pygame) để hiển thị mê cung, vị trí bắt đầu (S), lối thoát (G), các ngôi sao (*) và các ô tường/đường đi.

Triển khai các thuật toán tìm kiếm cổ điển không thông tin (như BFS, DFS, UCS) và có thông tin (như Greedy, A*) để tìm đường đi, đảm bảo thu thập đầy đủ ngôi sao trước khi đến G.

Thu thập dữ liệu thực nghiệm: Đo lường thời gian thực thi, số lượng nút mở rộng, độ dài đường đi và thời gian sử dụng cho từng thuật toán trên các mê cung có kích thước khác nhau (từ đơn giản đến phức tạp).

Phân tích và so sánh kết quả: Bảng so sánh để đánh giá thuật toán nào phù hợp nhất trong các tình huống cụ thể, đồng thời đề xuất cải tiến nếu cần.

Đảm bảo trò chơi có tính tương tác: Cho phép người dùng chọn thuật toán, tạo mê cung từ file, tự tạo mê cung mới hoặc chỉnh sửa các mê cung sẵn có, hiển thị được quá trình tìm kiếm và thực thi lời giải tìm được.

1.3. Phạm vi và đối tượng nghiên cứu

Phạm vi nghiên cứu của tiểu luận tập trung vào việc triển khai và phân tích, so sánh các thuật toán tìm kiếm cổ điển trong trí tuệ nhân tạo, áp dụng vào môi trường mê cung hai chiều dạng lưới, với kích thước mê cung được giới hạn từ 10x10 đến 50x50 ô nhằm đảm bảo tính khả thi về thời gian tính toán và tài nguyên hệ thống. Tiểu luận mở rộng bài toán cơ bản bằng cách tích hợp yêu cầu thu thập toàn bộ các ngôi sao như một biến thể đơn giản của vấn đề tìm đường đi, mà không bao quát các yếu tố phức tạp hơn như mê cung động hoặc không gian ba chiều. Các mê cung được thiết kế dưới dạng tĩnh, loại bỏ yếu tố ngẫu nhiên trong quá trình thực thi thuật toán, và chỉ hỗ trợ các hướng di chuyển cơ bản gồm bốn chiều (lên, xuống, trái, phải). Tiểu luận không đi sâu vào các khía cạnh tối ưu hóa mã nguồn nâng cao hoặc tích hợp các phương pháp machine learning, mà ưu tiên tập trung vào phân tích lý thuyết cơ bản và đánh giá thực nghiệm đối với các thuật toán cổ điển, nhằm làm rõ hiệu suất và hạn chế của chúng trong ngữ cảnh cụ thể.

Đối tượng mục tiêu của tiểu luận bao gồm các thuật toán tìm kiếm cổ điển gồm BFS, DFS, UCS, Greedy Best-First và A* khi giải bài toán tìm đường trên lưới 2D trong Maze Explorer với ràng buộc thu thập tất cả các ngôi sao trước khi tới đích. Nghiên cứu tập trung vào các đại lượng đặc trưng hiệu năng (thời gian thực thi, số nút mở rộng, độ dài đường đi) và yếu tố ảnh hưởng (kích thước mê cung 10×10 đến 50×50, cấu trúc hành lang/ngã rẽ, vị trí S/G, phân bố sao). Với thuật toán có heuristic, đối tượng còn bao gồm hàm heuristic Manhattan + số sao còn lại và các tính chất admissible/consistent trong ngữ cảnh bài toán. Các trường hợp kiểm thử (levels) do nhóm thiết kế, môi trường rời rạc 4 hướng (lên/xuống/trái/phải), chi phí bước đồng nhất.

CHƯƠNG 2: CƠ SỞ LÝ THUYẾT DÙNG ĐỂ THỰC HIỆN PROJECT

2.1. Công cụ và môi trường lập trình

Để xây dựng và phát triển project, nhóm chúng em sử dụng Python làm ngôn ngữ chính, kết hợp với Pygame để phát triển trò chơi 2D. Python được lựa chọn vì cú pháp đơn giản, cộng đồng lớn, dễ tiếp cận nhưng vẫn mạnh mẽ trong xử lý đồ họa, AI và dữ liệu.

Môi trường phát triển chính là Visual Studio Code (VS Code), hỗ trợ nhiều extension cho Python và Pygame, đồng thời tích hợp Git giúp quản lý mã nguồn thuận tiện. Ngoài ra, GitHub được sử dụng để lưu trữ và cộng tác trực tuyến giữa các thành viên trong nhóm.

2.2. Thư viện hỗ trợ lập trình

Pygame: Thư viện chính dùng để xây dựng game 2D, hỗ trợ quản lý vòng lặp game, xử lý sự kiện bàn phím, hiển thị hình ảnh, âm thanh. **NumPy:** Hỗ trợ các thao tác trên ma trận, được sử dụng để biểu diễn bản đồ mê cung dưới dạng lưới (grid). **JSON:** Dùng để lưu trữ dữ liệu lịch sử chơi (History/Stats), giúp dễ dàng đọc–ghi và hiển thị kết quả. **OS**: Hỗ trợ thao tác với hệ thống file khi load các level từ thư mục levels/.

2.3. Ngôn ngữ lập trình

Ngôn ngữ chính là Python, một ngôn ngữ thông dịch, đa dụng, có nhiều ưu điểm: hỗ trợ lập trình hướng đối tượng, thuận lợi khi xây dựng các class như Player; có hệ sinh thái phong phú, đặc biệt trong lĩnh vực AI và Game Development; dễ dàng kết hợp với các thuật toán tìm kiếm.

2.4. Phương pháp và kỹ thuật sử dụng

Mô hình Scene-based: Game được thiết kế theo mô hình nhiều Scene (Menu, Level Select, Level, History), giúp dễ dàng chuyển đổi và quản lý.

Thuật toán tìm kiếm trong AI: Các thuật toán như BFS, DFS, UCS, Greedy và A* được áp dụng để tìm đường trong mê cung. Đây là cơ sở lý thuyết nền tảng cho việc giải quyết các bài toán tìm kiếm trạng thái.

Quản lý dữ liệu game: Sử dụng JSON để lưu lại lịch sử chơi (điểm, thời gian, số bước di chuyển, số sao thu thập). Điều này giúp dễ dàng phân tích và

mở rộng thành hệ thống thống kê thành tích.

CHƯƠNG 3: PHÂN TÍCH, THIẾT KẾ GIẢI PHÁP

3.1. Mô hình hóa bài toán và trạng thái

Bài toán được mô hình hoá trên lưới 2D bốn hướng với chi phí cho mỗi bước đều là 1. Mỗi trạng thái được biểu diễn bởi bộ ba (x, y, mask), trong đó **(x, y)** là tọa độ hiện tại của agent và **mask** là bitmask biểu thị tập các ngôi sao đã thu thập. Trạng thái đích đạt được khi agent đứng tại ô G và mask đã bao phủ toàn bộ ngôi sao trong bản đồ. Từ một trạng thái hợp lệ, các phép toán (lên, xuống, trái, phải) tạo ra tối đa 4 trạng thái kế tiếp nếu ô tiếp theo được xét không phải là tường. Cách biểu diễn này được áp dụng thống nhất cho các thuật toán được sử dụng trong đề tài (BFS, DFS, UCS, Greedy và A*).

Trong hàm heuristic dùng cho thuật toán A* thì nhóm chúng em có sử dụng thêm ma trận khoảng cách để giảm không gian tìm kiếm, cụ thể là sẽ bao gồm 2 bước tiền xử lý: (i) tìm một tập POI bao gồm S, G và các ngôi sao; (ii) tính và lưu lại khoảng cách ngắn nhất giữa mọi cặp POI bằng BFS trên lưới chướng ngại.

3.2. Kiến trúc hệ thống

Hệ thống được xây dựng theo kiến trúc scene-based, tách rõ phần giao diện, logic và module thuật toán

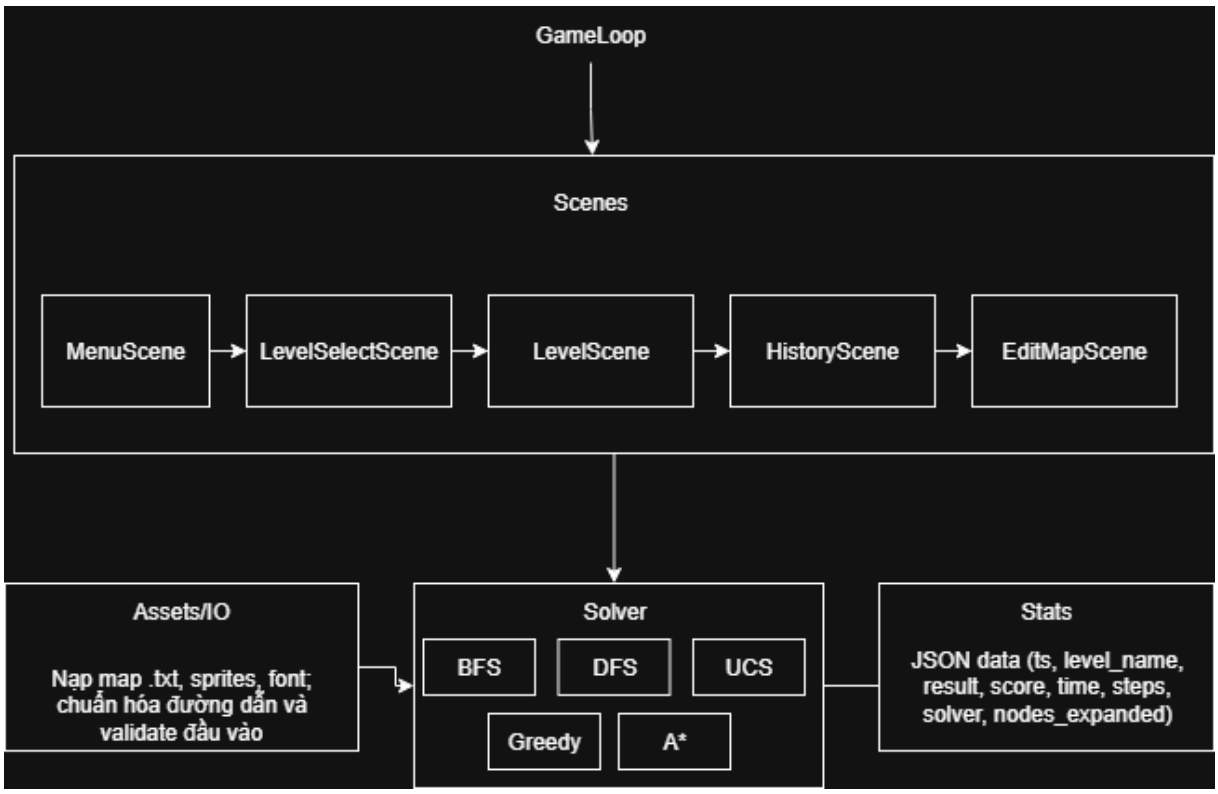
GameLoop: có nhiệm vụ điều phối vòng lặp chính (xử lý sự kiện, cập nhật, vẽ giao diện).

Các Scene: MenuScene, LevelSelectScene, LevelScene, HistoryScene, EditMapScene.

Solver: tập module cài đặt các thuật toán

Stats: ghi log bao gồm found, steps, nodes_expanded, time, đường đi, … vào JSON để phục vụ cho việc phân tích.

Assets/IO: nạp map .txt, sprites, font; chuẩn hoá đường dẫn và validate đầu vào.



Hình 3. 1: Sơ đồ kiến trúc tổng quát

3.3. Chuẩn hóa chung kết quả trả về giữa các thuật toán

```
path, moves = _reconstruct_path(parents, end_state)
    return {
        "path": path,
        "moves": moves,
        "steps": len(moves),
        "stars_total": len(stars),
        "found": True,
        "expanded_order": expanded_order,
        "nodes_expanded": len(expanded_order),
    }
```

Quy ước: path là danh sách các tọa độ đường đi, moves là danh sách các bước di chuyển (UDLR), steps là độ dài đường đi để giải bài toán với thuật toán được chọn tương ứng, expanded_order là thứ tự mở rộng các nút, nodes_expanded đếm mỗi lần lấy một đỉnh ra khỏi stack, queue và priority queue để đảm bảo so sánh công bằng giữa các thuật toán.

CHƯƠNG 4: THỰC NGHIỆM, ĐÁNH GIÁ, PHÂN TÍCH

4.1. Cài đặt thuật toán

4.1.1. BFS

Thuật toán BFS nhận đầu vào là danh sách chuỗi ký tự biểu diễn mê cung và trả về một dictionary chứa đường đi, các bước di chuyển, số bước, tổng số sao và trạng thái tìm thấy giải pháp. Quá trình thực hiện bắt đầu bằng việc phân tích mê cung để xác định vị trí bắt đầu, kết thúc và các sao cần thu thập. Sau đó, thuật toán khởi tạo các cấu trúc dữ liệu cần thiết bao gồm star_index để ánh xạ vị trí sao với chỉ số bit, all_mask là bitmask đại diện cho tất cả sao, queue để lưu trữ các trạng thái cần xử lý, visited set để tránh lặp lại và parents dictionary để truy vết đường đi. Vòng lặp chính của thuật toán liên tục lấy trạng thái từ đầu queue theo nguyên tắc FIFO, thêm vị trí vào danh sách expanded_order để theo dõi quá trình mở rộng. Với mỗi trạng thái hiện tại, thuật toán kiểm tra điều kiện thắng là đứng tại vị trí goal và đã thu thập đủ tất cả sao. Nếu chưa thỏa mãn, thuật toán mở rộng tất cả các trạng thái kề có thể, thêm những trạng thái chưa được thăm vào queue và cập nhật thông tin cha để có thể truy vết đường đi sau này. Độ phức tạp thời gian của BFS là O(b^d) với b là branching factor và d là độ sâu của giải pháp, trong khi độ phức tạp không gian cũng là O(b^d) để lưu trữ queue và visited set.

```
ALGORITHM: BFS
INPUT:
    - grid: Ma trận ký tự (S=start, G=goal, *=star,
1=wall)
OUTPUT:
    - path: Đường đi từ S đến G
    - moves: Chuỗi di chuyển (U/D/L/R)
    - found: Có tìm thấy đường đi hợp lệ không
------------------------------------------------------
-------------------------
PHASE 1: Khởi tạo
1. Parse grid để tìm:
   - start_pos: Vị trí S
   - goal_pos: Vị trí G
```

```
   - stars: Danh sách các vị trí *
2. Tạo bitmask cho sao:
   - all_mask = (2^số_sao - 1)  // Tất cả sao đã thu
thập
   - start_mask = 0              // Chưa thu thập sao
nào
3. Khởi tạo BFS:
   - State = (x, y, collected_mask)
   - queue = [(start_x, start_y, start_mask)]
   - visited = {(start_x, start_y, start_mask)}
   - parents = {} // Để truy vết đường đi
-------------------------------------------------------
-------------------------
PHASE 2: BFS Expansion
WHILE queue không rỗng:
    state = queue.dequeue()
    (x, y, mask) = state
    // Kiểm tra điều kiện thắng
    IF (x, y) == goal AND mask == all_mask:
        RETURN reconstruct_path(state)
    // Duyệt 4 hướng: Up, Down, Left, Right
    FOR EACH direction IN [(0,-1), (0,1), (-1,0),
(1,0)]:
        next_x = x + direction.dx
        next_y = y + direction.dy
        // Kiểm tra hợp lệ
        IF is_wall(next_x, next_y) OR
out_of_bounds(next_x, next_y):
            CONTINUE
        // Cập nhật mask nếu gặp sao
        next_mask = mask
        IF (next_x, next_y) là vị trí sao:
            star_bit = star_index[(next_x, next_y)]
            next_mask = mask OR (1 << star_bit)
        next_state = (next_x, next_y, next_mask)
```

```
        // Thêm vào queue nếu chưa thăm
        IF next_state NOT IN visited:
            visited.add(next_state)
            parents[next_state] = (state, direction)
            queue.enqueue(next_state)
RETURN "No path found"
```

4.1.2. DFS

Thuật toán DFS nhận đầu vào là danh sách chuỗi ký tự biểu diễn mê cung và trả về một dictionary chứa đường đi, các bước di chuyển, số bước, tổng số sao, trạng thái tìm thấy và thứ tự mở rộng các trạng thái. Quá trình thực hiện bắt đầu bằng việc phân tích mê cung để xác định các thành phần cần thiết, sau đó khởi tạo các cấu trúc dữ liệu tương tự như BFS nhưng sử dụng Stack thay vì Queue. Stack được khởi tạo với trạng thái bắt đầu và các biến hỗ trợ như visited set, parents dictionary và expanded_order list. Vòng lặp chính của DFS liên tục lấy trạng thái từ đỉnh stack theo nguyên tắc LIFO, thêm vị trí vào danh sách expanded_order để theo dõi quá trình mở rộng. Với mỗi trạng thái hiện tại, thuật toán kiểm tra điều kiện thắng và nếu chưa thỏa mãn thì mở rộng các trạng thái kề có thể. Các trạng thái mới được thêm vào cuối stack và đánh dấu đã thăm, đồng thời cập nhật thông tin cha để truy vết. Độ phức tạp thời gian của DFS là O(b^m) với m là độ sâu tối đa của cây tìm kiếm, trong khi độ phức tạp không gian là O(b*m) để lưu trữ stack và visited set, thường tiết kiệm bộ nhớ hơn BFS.

```
ALGORITHM: DFS
INPUT:
    - grid: Ma trận ký tự (S=start, G=goal, *=star,
1=wall)

OUTPUT:
    - path: Đường đi từ S đến G
    - moves: Chuỗi di chuyển (U/D/L/R)
    - found: Có tìm thấy đường đi hợp lệ không
PHASE 1: Khởi tạo
1. Parse grid để tìm start_pos, goal_pos, stars
2. Tạo bitmask:
   - all_mask = (2^số_sao - 1)
```

```
   - start_mask = 0
3. Khởi tạo DFS:
   - State = (x, y, collected_mask)
   - stack = [(start_x, start_y, start_mask)]
   - visited = {(start_x, start_y, start_mask)}
   - parents = {}
PHASE 2: DFS Exploration
WHILE stack không rỗng:
    state = stack.pop()
    (x, y, mask) = state
    // Kiểm tra điều kiện thắng
    IF (x, y) == goal AND mask == all_mask:
        RETURN reconstruct_path(state)
    // Duyệt 4 hướng: Up, Down, Left, Right
    FOR EACH direction IN [(0,-1), (0,1), (-1,0),
(1,0)]:
        next_x = x + direction.dx
        next_y = y + direction.dy
        // Kiểm tra hợp lệ
        IF is_wall(next_x, next_y) OR
out_of_bounds(next_x, next_y):
            CONTINUE
        // Cập nhật mask nếu gặp sao
        next_mask = mask
        IF (next_x, next_y) là vị trí sao:
            star_bit = star_index[(next_x, next_y)]
            next_mask = mask OR (1 << star_bit)
        next_state = (next_x, next_y, next_mask)
        // Thêm vào stack nếu chưa thăm
        IF next_state NOT IN visited:
            visited.add(next_state)
            parents[next_state] = (state, direction)
            stack.push(next_state)
RETURN "No path found"
```

4.1.3. UCS

Thuật toán UCS nhận đầu vào là danh sách chuỗi ký tự biểu diễn mê cung và trả về một dictionary chứa đường đi, các bước di chuyển, số bước, tổng số sao, trạng thái tìm thấy và thứ tự mở rộng các trạng thái. Quá trình thực hiện bắt đầu bằng việc phân tích mê cung để xác định các thành phần cần thiết, sau đó khởi tạo priority queue với trạng thái bắt đầu có chi phí bằng 0. Thuật toán duy trì g_scores dictionary để theo dõi chi phí từ điểm bắt đầu đến mỗi trạng thái, cùng với visited set, parents dictionary và expanded_order list. Vòng lặp chính của UCS liên tục lấy trạng thái có chi phí thấp nhất từ priority queue bằng hàm heappop, thêm vị trí vào danh sách expanded_order để theo dõi quá trình mở rộng. Với mỗi trạng thái hiện tại, thuật toán kiểm tra điều kiện thắng và nếu chưa thỏa mãn thì mở rộng các trạng thái kề có thể. Với mỗi trạng thái kề, thuật toán tính tentative_g là chi phí từ điểm bắt đầu đến trạng thái đó, sau đó thêm vào priority queue với chi phí này và cập nhật các cấu trúc dữ liệu hỗ trợ. Độ phức tạp thời gian của UCS là O(b^(1+C*/ε)) với C* là chi phí của giải pháp tối ưu và ε là chi phí tối thiểu, trong khi độ phức tạp không gian cũng là O(b^(1+C*/ε)) để lưu trữ priority queue và visited set.

```
ALGORITHM: UCS (Uniform Cost Search)
INPUT:
    - grid: Ma trận ký tự (S=start, G=goal, *=star,
1=wall)
OUTPUT:
    - path: Đường đi ngắn nhất từ S đến G
    - moves: Chuỗi di chuyển (U/D/L/R)
    - found: Có tìm thấy đường đi hợp lệ không
PHASE 1: Khởi tạo
1. Parse grid để tìm start_pos, goal_pos, stars
2. Tạo bitmask:
   - all_mask = (2^số_sao - 1)
   - start_mask = 0
3. Khởi tạo UCS:
   - State = (x, y, collected_mask)
   - priority_queue = [(0, start_state)]
   - g_scores = {start_state: 0}
   - closed_set = set()
```

```
   - parents = {}
PHASE 2: UCS Exploration
WHILE priority_queue không rỗng:
    (g_score, state) = heap_pop(priority_queue)
    (x, y, mask) = state
    IF state IN closed_set:
        CONTINUE
    closed_set.add(state)
    IF (x, y) == goal AND mask == all_mask:
        RETURN reconstruct_path(state)
    FOR EACH direction IN [(0,-1), (0,1), (-1,0),
(1,0)]:
        next_x = x + direction.dx
        next_y = y + direction.dy
        // Kiểm tra hợp lệ
        IF is_wall(next_x, next_y) OR
out_of_bounds(next_x, next_y):
            CONTINUE
        next_mask = mask
        IF (next_x, next_y) là vị trí sao:
            star_bit = star_index[(next_x, next_y)]
            next_mask = mask OR (1 << star_bit)
        next_state = (next_x, next_y, next_mask)
        IF next_state IN closed_set:
            CONTINUE
        tentative_g = g_scores[state] + 1
        IF tentative_g < g_scores.get(next_state, ∞):
            g_scores[next_state] = tentative_g
            parents[next_state] = (state, direction)
            heap_push(priority_queue, (tentative_g,
next_state))
RETURN "No path found"
```

4.1.4. Best First Search

Thuật toán bắt đầu bằng việc tiền xử lý bản đồ để xác định vị trí xuất phát

S, đích G và danh sách các sao, đồng thời ánh xạ từng ngôi sao sang một chỉ số bit để hình thành $mask$ và giá trị all_mask tương ứng với "đã thu thập hết". Trạng thái khởi tạo là $(S_x, S_y, 0)$ và được đưa vào một hàng đợi ưu tiên cùng với giá trị heuristic ban đầu. Ở mỗi vòng lặp, thuật toán lấy ra trạng thái có h nhỏ nhất và kiểm tra điều kiện dừng; nếu đã đứng ở G và $mask = all\_mask$, đường đi được tái dựng ngược từ mảng cha và thuật toán kết thúc. Ngược lại, bốn trạng thái kề hợp lệ được sinh ra bằng cách di chuyển một ô theo bốn hướng; nếu ô mới là ngôi sao, $mask$ được cập nhật bằng phép "or" bit; với mỗi trạng thái mới chưa từng thăm, heuristic được tính theo công thức đã nêu và trạng thái được chèn vào hàng đợi ưu tiên. Quá trình lặp tiếp tục cho đến khi tìm thấy nghiệm hoặc hàng đợi trống (trong trường hợp không tồn tại lời giải). Về độ phức tạp, chi phí thời gian và số nút mở rộng phụ thuộc vào chất lượng heuristic và cấu trúc mê cung; trong trường hợp xấu, số nút mở rộng có thể tiến gần $O(b\^d)$ với b là bậc phân nhánh và d là độ sâu nghiệm, nhưng khi heuristic định hướng tốt, thuật toán thường mở rộng ít nút hơn đáng kể so với các phương pháp tìm kiếm mù.

```
ALGORITHM: Greedy_Best_First_Search
INPUT:
    - grid: Ma trận ký tự (S=start, G=goal, *=star,
1=wall)
OUTPUT:
    - path: Đường đi từ S đến G
    - moves: Chuỗi di chuyển (U/D/L/R)
    - found: Có tìm thấy đường đi hợp lệ không
PHASE 1: Khởi tạo
1. Parse grid để tìm start_pos, goal_pos, stars
2. Tạo bitmask:
   - all_mask = (2^số_sao - 1)
   - start_mask = 0
3. Khởi tạo Greedy Search:
   - State = (x, y, collected_mask)
   - priority_queue = [(h_score, start_state)]
   - visited = set()
   - parents = {}
```

```
4. Tính heuristic ban đầu:
   h_start = manhattan(start, goal) +
số_sao_chưa_thu_thập
PHASE 2: Greedy Exploration
WHILE priority_queue không rỗng:
    (h_score, state) = heap_pop(priority_queue
    (x, y, mask) = state
    IF (x, y) == goal AND mask == all_mask:
        RETURN reconstruct_path(state)
    FOR EACH direction IN [(0,-1), (0,1), (-1,0),
(1,0)]:
        next_x = x + direction.dx
        next_y = y + direction.dy
        IF is_wall(next_x, next_y) OR
out_of_bounds(next_x, next_y):
            CONTINUE
        next_mask = mask
        IF (next_x, next_y) là vị trí sao:
            star_bit = star_index[(next_x, next_y)]
            next_mask = mask OR (1 << star_bit)
        next_state = (next_x, next_y, next_mask)
        IF next_state IN visited:
            CONTINUE
        visited.add(next_state)
        // HEURISTIC CALCULATION (Trọng tâm của
Greedy)
        remaining_stars = [s for s in stars if NOT
(next_mask & (1 << star_index[s]))]
        IF remaining_stars không rỗng:
            nearest_star_dist = MIN(manhattan((next_x,
next_y), s) for s in remaining_stars)
            h_score = nearest_star_dist +
manhattan((next_x, next_y), goal)
        ELSE:
```

```
            // Đã thu thập hết sao, chỉ cần đến goal
            h_score = manhattan((next_x, next_y),
goal)
        parents[next_state] = (state, direction)
        heap_push(priority_queue, (h_score,
next_state))
RETURN "No path found"
```

*4.1.5. A**

Thuật toán khởi đầu bằng việc tách S, G và danh sách sao, ánh xạ mỗi sao sang một bit trong $mask$ và xác định all_mask. Trước khi tìm kiếm, thuật toán tiền xử lý **ma trận khoảng cách ngắn nhất** giữa các điểm quan tâm bằng BFS trên lưới để có thể lấy d tức thời khi tính heuristic. Tại thời điểm chạy, A* sử dụng một **min-heap** lưu cặp $(f, state)$, cùng các bảng g_scores (chi phí tốt nhất đã biết tới state), $parents$ (phục dựng đường đi), và một closed_set để đánh dấu các trạng thái đã "đóng". Ở mỗi vòng lặp, thuật toán lấy trạng thái có f nhỏ nhất ra mở rộng; nếu trạng thái này đang đứng ở G và $mask = all\_mask$, đường đi tối ưu sẽ được tái dựng ngược qua $parents$. Ngược lại, với mỗi láng giềng hợp lệ theo bốn hướng, thuật toán cập nhật $mask$ nếu đi qua ô sao, tính $g' = g + 1$ và h theo công thức MST, rồi đẩy vào heap với $f' = g' + h$ nếu tìm được đường tốt hơn. Nhờ tính admissible/consistent của heuristic, A* sẽ dừng với lời giải tối ưu và không cần xét lại các trạng thái đã đóng; về chi phí tính toán, số nút mở rộng phụ thuộc mạnh vào chất lượng heuristic và cấu trúc mê cung, nhưng trong thực nghiệm nó thường nhỏ hơn đáng kể so với UCS hoặc Best-First khi phải "ghé thăm" nhiều sao trên bản đồ.

```
ALGORITHM: A*_Search_with_MST_Heuristic
INPUT:
    - grid: Ma trận ký tự (S=start, G=goal, *=star,
1=wall)
OUTPUT:
    - path: Đường đi NGẮN NHẤT từ S đến G
    - moves: Chuỗi di chuyển (U/D/L/R)
    - found: Có tìm thấy đường đi hợp lệ không
PHASE 0: TIỀN XỬ LÝ (Preprocessing)
```

```
1. Parse grid để tìm start_pos, goal_pos, stars
2. PRECOMPUTE DISTANCES giữa tất cả POI (Points of
Interest):
   POI = {S, G, star_1, star_2, ..., star_n}
   FOR EACH poi IN POI:
       distances[poi] = BFS_from(poi) // Tính dist đến
tất cả ô
   FOR EACH pair (poi_i, poi_j) IN POI × POI:
       dist_matrix[(poi_i, poi_j)] =
distances[poi_i][poi_j]
   → Tạo ma trận khoảng cách THỰC TẾ giữa các POI
(không phải Manhattan)
3. Tạo bitmask:
   - all_mask = (2^số_sao - 1)
   - start_mask = 0
PHASE 1: KHỞI TẠO A*
1. State = (x, y, collected_mask)
2. priority_queue = [(f_score, start_state)] // MIN-
HEAP (f_score, state)
3. g_scores = {start_state: 0} // Chi phí từ start
4. closed_set = set()       // State đã xử lý
5. parents = {}
6. mst_cache = {} // Cache MST theo mask
7. Tính f_score ban đầu:
   h_start = heuristic_MST(start_state)
   f_start = 0 + h_start
   queue.push((f_start, start_state))
PHASE 2: A* EXPLORATION
WHILE priority_queue không rỗng:
    (f_score, state) = heap_pop(priority_queue)
    (x, y, mask) = state
    IF state IN closed_set:
        CONTINUE
    closed_set.add(state)
    // Kiểm tra điều kiện thắng
```

```
    IF (x, y) == goal AND mask == all_mask:
        RETURN reconstruct_path(state)
    FOR EACH direction IN [(0,-1), (0,1), (-1,0),
(1,0)]:
        next_x = x + direction.dx
        next_y = y + direction.dy
        IF is_wall(next_x, next_y) OR
out_of_bounds(next_x, next_y):
            CONTINUE
        // Cập nhật mask nếu gặp sao
        next_mask = mask
        IF (next_x, next_y) là vị trí sao:
            star_bit = star_index[(next_x, next_y)]
            next_mask = mask OR (1 << star_bit)
        next_state = (next_x, next_y, next_mask)
        IF next_state IN closed_set:
            CONTINUE
        tentative_g = g_scores[state] + 1
        IF tentative_g < g_scores.get(next_state, ∞):
            g_scores[next_state] = tentative_g
            parents[next_state] = (state, direction)
            h_score = heuristic_MST(next_state)
            f_score = tentative_g + h_score
            heap_push(priority_queue, (f_score,
next_state))
RETURN "No path found"
PHASE 3: HEURISTIC MST (Trọng tâm của A*)
FUNCTION heuristic_MST(state):
    """
    Công thức: h(state) = d(cur, R) + MST(R) + d(R, G)
    Trong đó:
    - cur: vị trí hiện tại
    - R: tập sao còn lại chưa thu thập
    - G: goal
```

```
    - d(a, B): khoảng cách ngắn nhất từ a đến một phần
tử trong tập B
    - MST(R): trọng số cây khung nhỏ nhất của tập R
    """
    (x, y, mask) = state
    current_pos = (x, y)
    // 1. Xác định sao còn lại
    remaining_stars = [stars[i] for i in
range(len(stars)) if NOT (mask & (1 << i))]
    // 2. Trường hợp đặc biệt: đã thu thập hết sao
    IF remaining_stars rỗng:
        RETURN get_distance(current_pos, goal)
    // 3. Tính d(cur, R): khoảng cách đến sao gần nhất
    min_dist_to_stars = MIN(get_distance(current_pos,
star) for star in remaining_stars)
    // 4. Tính MST(R): sử dụng cache để tối ưu
    IF mask NOT IN mst_cache:
        mst_cache[mask] = compute_MST(remaining_stars)
    mst_weight = mst_cache[mask]
    // 5. Tính d(R, G): khoảng cách từ sao xa nhất đến
goal
    min_dist_stars_to_goal = MIN(get_distance(star,
goal) for star in remaining_stars)
    // 6. Tổng hợp
    h = min_dist_to_stars + mst_weight +
min_dist_stars_to_goal
    RETURN h
```

4.2. Hướng dẫn thực thi phần mềm

4.2.1. Khởi chạy game

Bước 1: Clone Repository từ GitHub

Trước tiên, mở Terminal tại thư mục muốn chứa project và gõ lệnh:

```
git clone https://github.com/chishiya2309/AI_Project_Maze_Explorer.git
```

Sau khi clone xong, di chuyển vào thư mục dự án

```
cd AI_Project_Maze_Explorer
```

```
PS D:\> git clone https://github.com/chishiya2309/AI_Project_Maze_Explorer.git
Cloning into 'AI_Project_Maze_Explorer'...
remote: Enumerating objects: 533, done.
remote: Counting objects: 100% (154/154), done.
remote: Compressing objects: 100% (118/118), done.
remote: Total 533 (delta 54), reused 103 (delta 34), pack-reused 379 (from 1)
Receiving objects: 100% (533/533), 8.84 MiB | 5.39 MiB/s, done.
Resolving deltas: 100% (223/223), done.
PS D:\> ^C
PS D:\> cd AI_Project_Maze_Explorer
```

Bước 2: Tạo và kích hoạt môi trường ảo (khuyến khích):

Để đảm bảo các thư viện cài đặt không xung đột với hệ thống, nên tạo môi trường ảo:

Trên Windows:

```
python -m venv venv

venv\Scripts\activate
```

Trên macOS / Linux:

```
python3 -m venv venv

source venv/bin/activate
```

```
PS D:\AI_Project_Maze_Explorer> python -m venv venv
PS D:\AI_Project_Maze_Explorer> venv\Scripts\activate
```

Bước 3: Cài đặt các thư viện cần thiết

Cài đặt các dependencies được liệt kê trong file requirements.txt:

```
pip install -r requirements.txt
```

```
(venv) PS D:\AI_Project_Maze_Explorer> pip install -r requirements.txt
Collecting pygame>=2.1.0
  Downloading pygame-2.6.1-cp39-cp39-win_amd64.whl (10.6 MB)
     |████████████████████████████████| 10.6 MB 547 kB/s
Collecting opencv-python>=4.5.0
  Downloading opencv_python-4.12.0.88-cp37-abi3-win_amd64.whl (39.0 MB)
     |████████████████████████████████| 39.0 MB 3.3 MB/s
Collecting numpy<2.3.0,>=2
  Downloading numpy-2.0.2-cp39-cp39-win_amd64.whl (15.9 MB)
     |████████████████████████████████| 15.9 MB 2.2 MB/s
Installing collected packages: numpy, pygame, opencv-python
Successfully installed numpy-2.0.2 opencv-python-4.12.0.88 pygame-2.6.1
```

Bước 4: Khởi chạy game

Sau khi cài đặt xong, chạy game bằng lệnh:

```
python main.py
```

4.2.2. Điều hướng trong menu chính

Phím mũi tên ↑↓: Di chuyển giữa các tùy chọn

Enter/Space: Kích hoạt tùy chọn được chọn

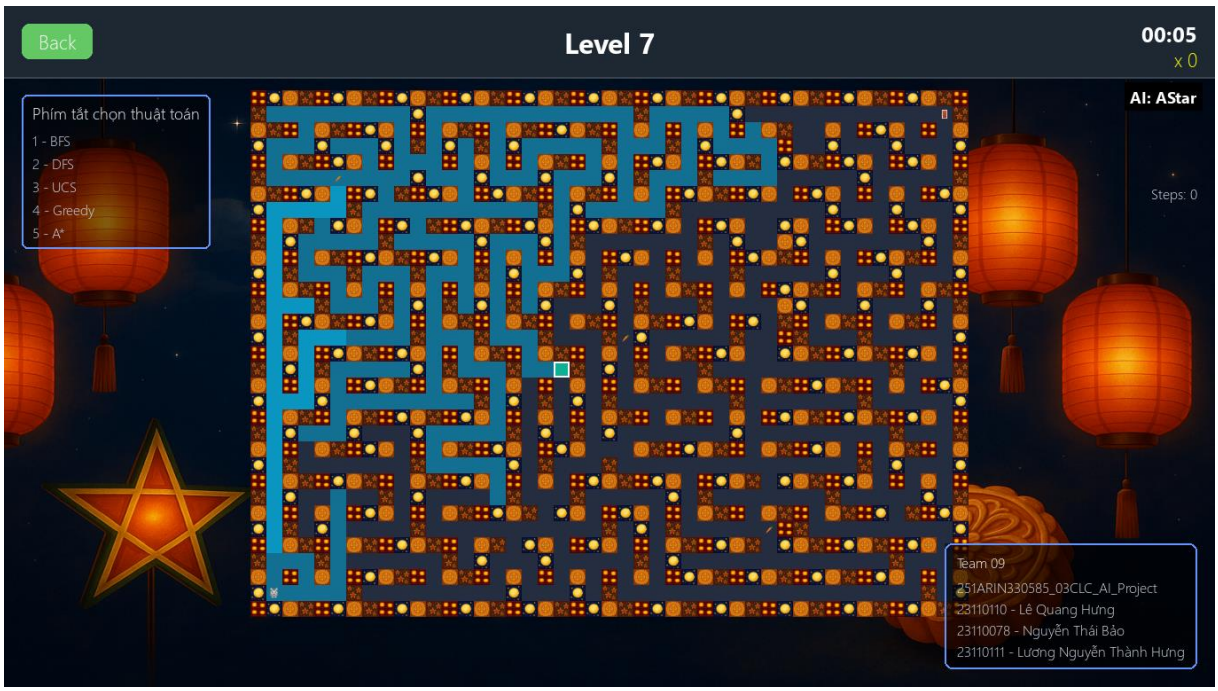
Chuột: Click trực tiếp vào tùy chọn mong muốn

4.2.3. Chơi game

Chọn "Start" từ menu chính. Sử dụng phím mũi tên hoặc chuột để chọn level. Nhấn Enter để bắt đầu chơi. Tự chơi bằng cách điều hướng bằng WASD hoặc ↑←↓→. Chọn thuật toán để chơi bằng cách nhấn phím số với bảng sau:

Phím số	Thuật toán
1	BFS
2	DFS
3	UCS
4	Best First Search
5	A*

Khi chọn thuật toán thì chương trình sẽ thể hiện quá trình tìm kiếm lời giải và khi tìm được lời giải thì sẽ ngay lập tức thực thi lời giải, tên thuật toán sẽ được hiển thị ở góc trên bên phải và phía dưới header.



Hình 4. 1: Quá trình tìm kiếm lời giải của A ở Level 7*

Hình 4. 2: Quá trình thực thi lời giải của A ở Level 7*

4.2.4. Tạo / chỉnh sửa bản đồ

Chọn "Edit Map" từ menu chính.

Chỉnh sửa level có sẵn: Click vào card level.

Tạo level mới: Click vào "New Map" → Nhập kích thước → Nhấn Enter.

Phím 1-5: Chọn công cụ (Wall, Floor, Start, Goal, Star).

Click chuột: Đặt tile.

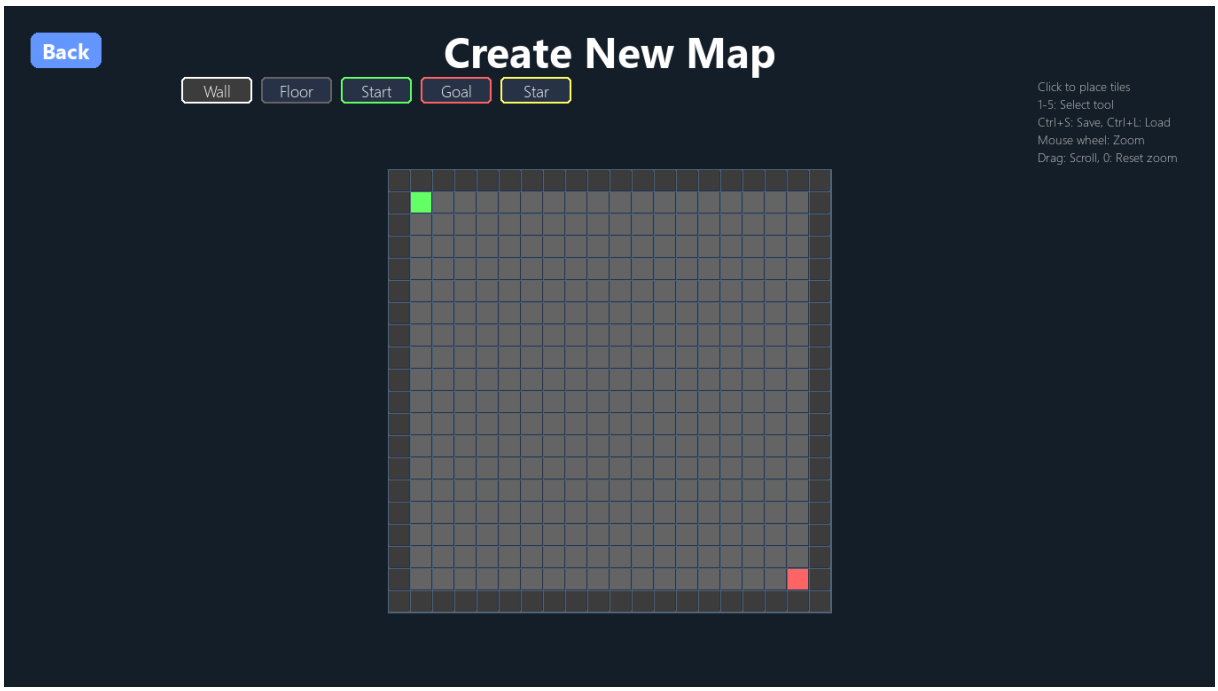
Kéo chuột: Vẽ liên tục.

Mouse wheel: Thu phóng.

Drag chuột giữa: Cuộn bản đồ.

Ctrl+S: Lưu bản đồ.

Hình 4. 3: Khởi tạo map mới với kích thước là 20x20



Hình 4. 4: Map mặc định khi khởi tạo map mới

4.2.5. Xem lịch sử

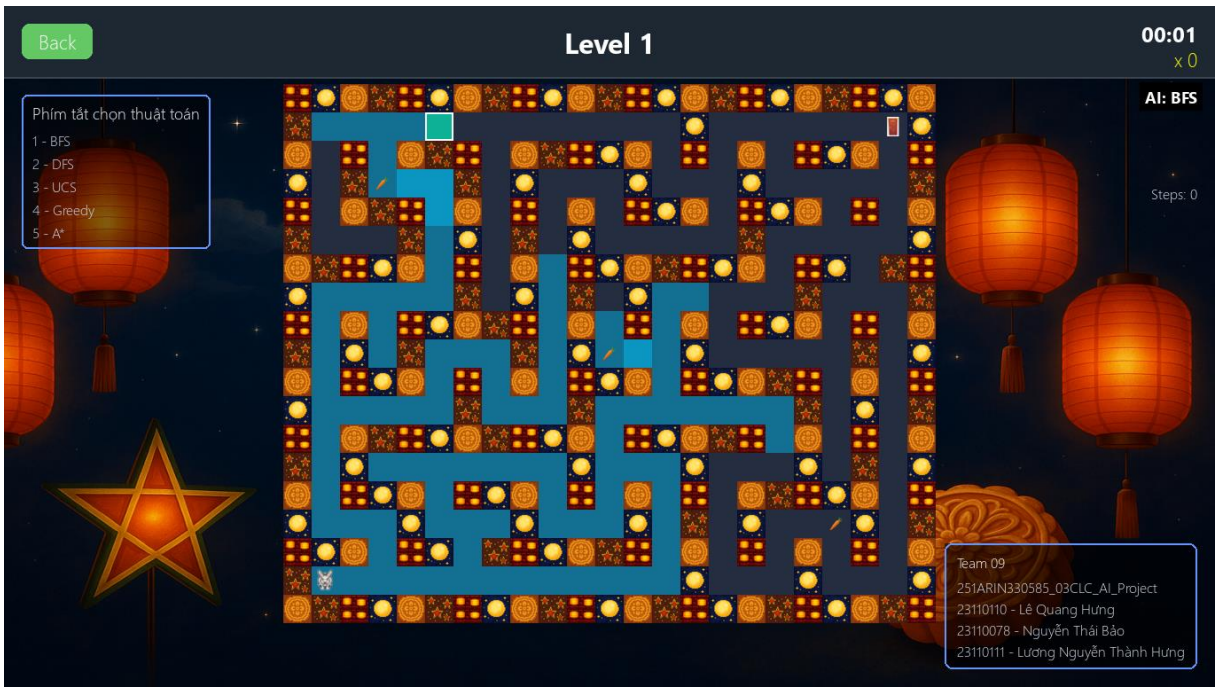
Chọn "History" từ menu chính

Sử dụng phím mũi tên hoặc scroll wheel để duyệt.

Thông tin hiển thị: kết quả, điểm số, thời gian, số bước, thuật toán sử dụng, số nút đã duyệt.

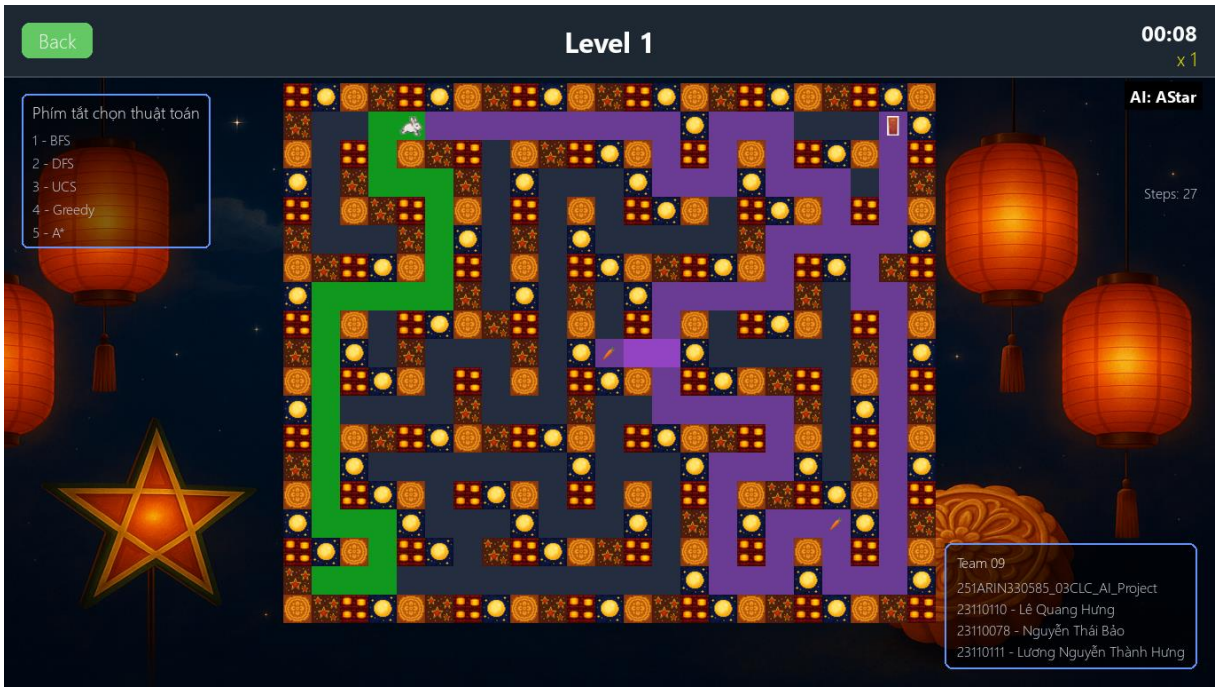
4.3. Kết quả sau khi chạy thuật toán

Sau khi chọn 1 level thành công, map sẽ được load, ta chọn 1 thuật toán để chạy như hướng dẫn ở mục 4.2.3.



Hình 4. 5: Quá trình tìm kiếm lời giải khi chạy thuật toán

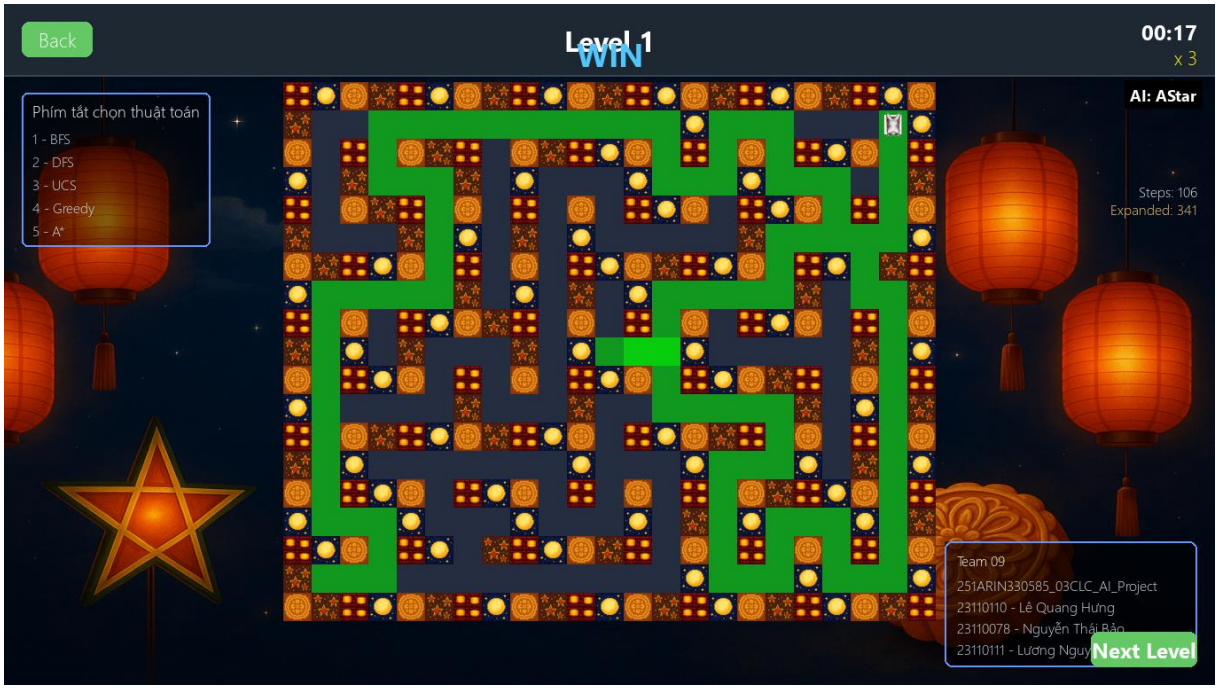
Trong quá trình tìm kiếm thì các màu được thể hiện như sau: các ô đã mở rộng thì dùng CYAN tươi (0, 200, 255, 110), alpha trung bình để nổi bật, còn ô hiện tại của tìm kiếm thì dùng teal đậm (0, 255, 200, 160) kèm viền trắng 2px.



Hình 4. 6: Quá trình thực thi lời giải sau khi tìm được lời giải

Trong quá trình thực thi lời giải thì đường đi đã thực thi được thể hiện bằng màu xanh lá sáng (0, 255, 0, 130), còn đường đi còn lại dùng màu

tím/magenta (200, 80, 255, 110).



Hình 4. 7: Kết quả sau khi chạy thuật toán

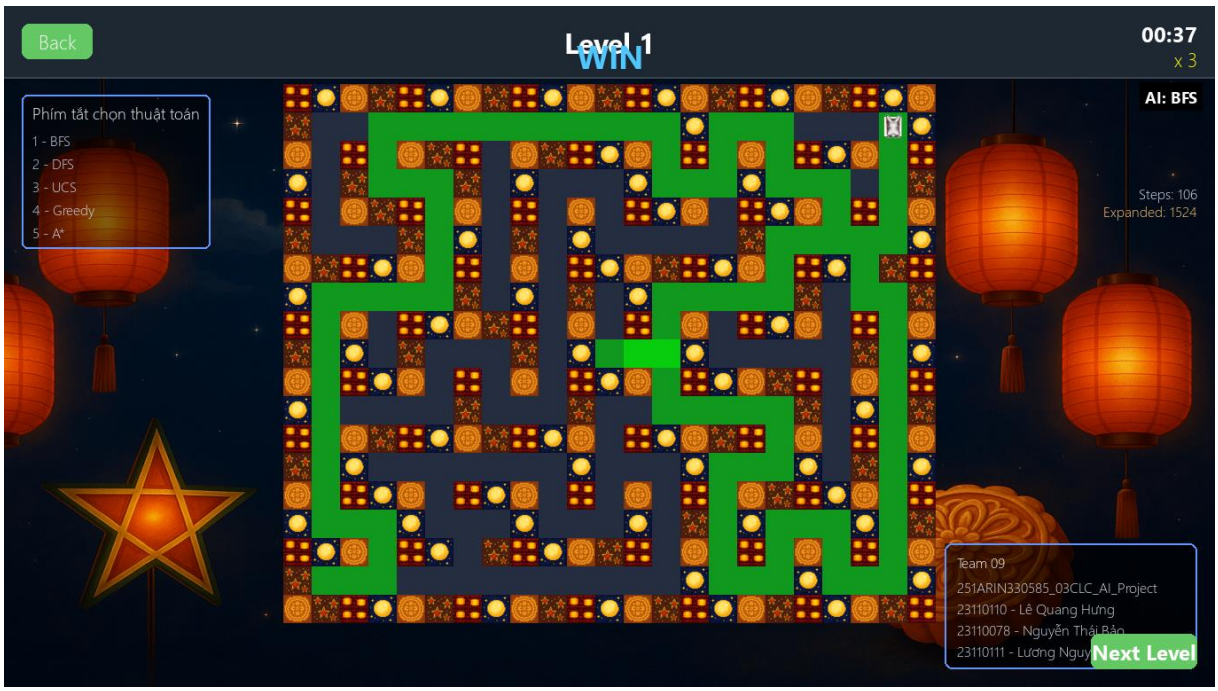
Với thuật toán A Star phía trên và map Level 1, ta thấy thuật toán mất thời gian hoàn thành là 17 giây, chi phí đường đi là 106 nút và số lượng nút đã mở rộng là 341.

4.4. So sánh các thuật toán

4.4.1. Sử dụng một Level 1

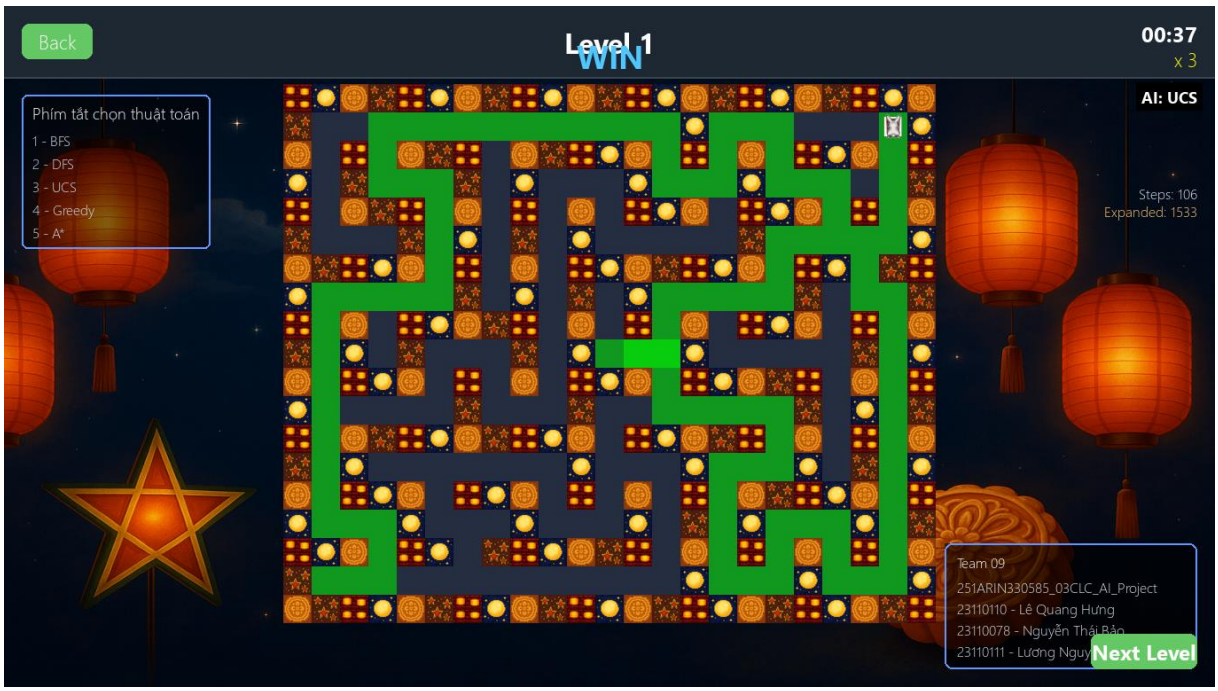
Nhóm chúng em chọn level_01.txt làm bản đồ ma trận để thể hiện 5 thuật toán được cài đặt.

Kết quả chạy 5 thuật toán bao gồm: BFS, DFS, UCS, Best First Search và A* được thể hiện lần lượt như các hình dưới đây:

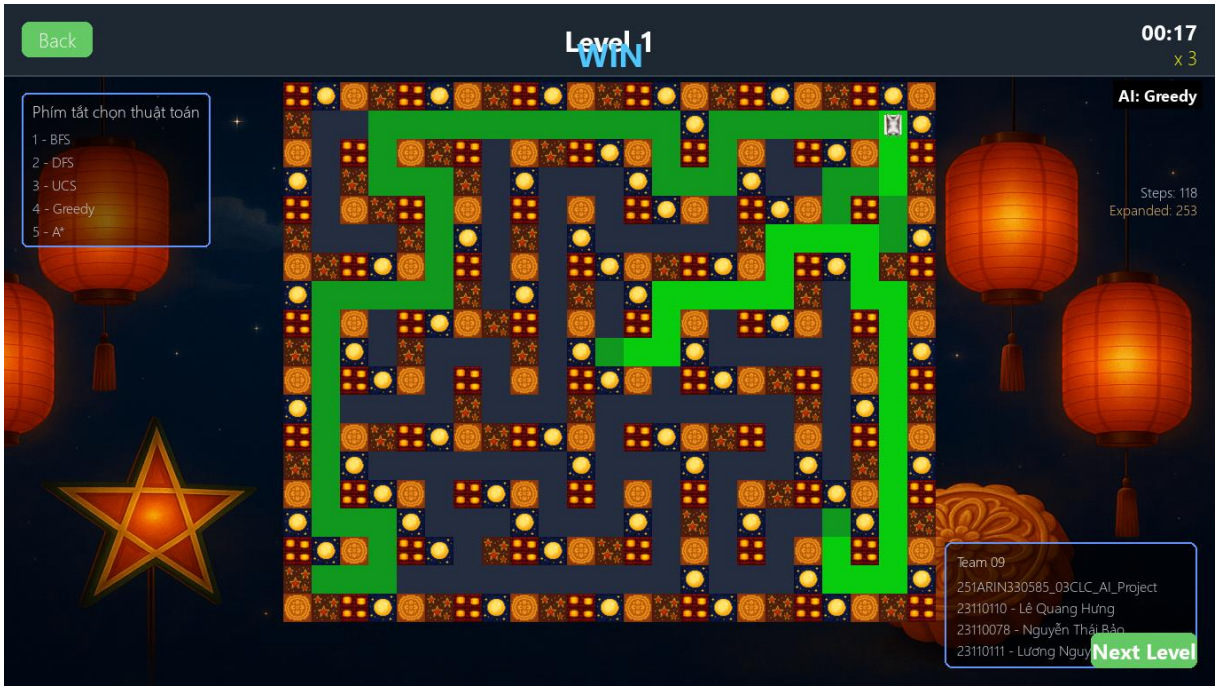


Hình 4. 8: Thuật toán BFS trên Level 1

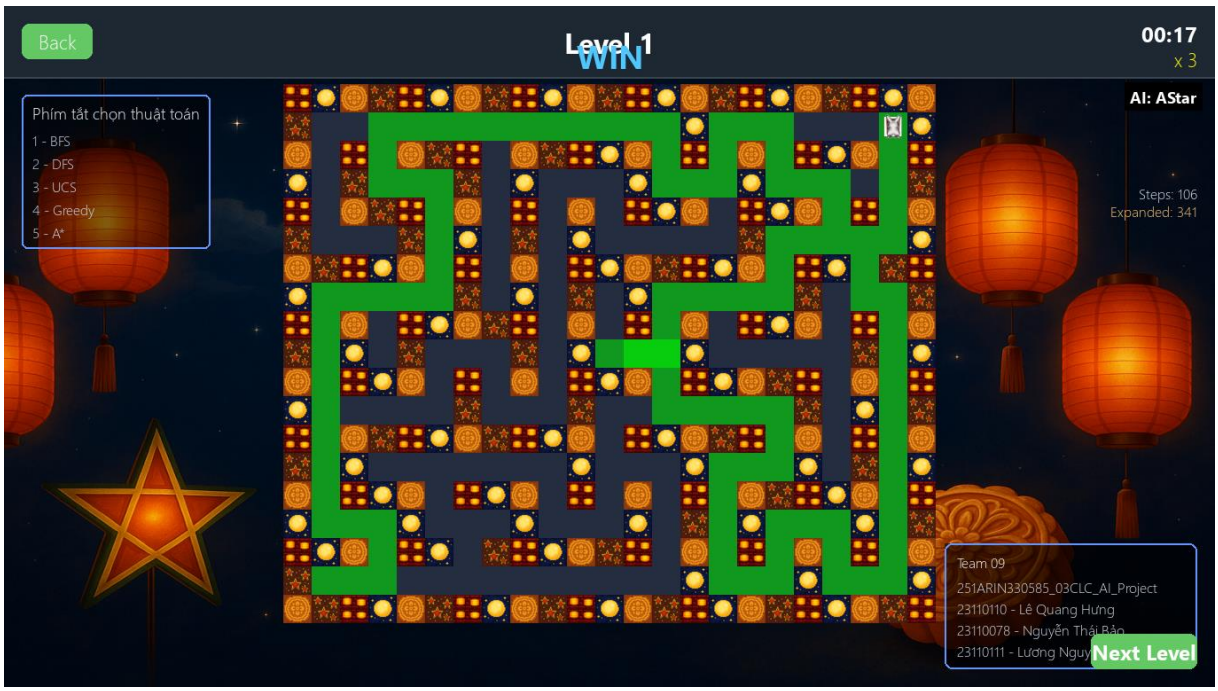
Hình 4. 9: Thuật toán DFS trên Level 1



Hình 4. 10: Thuật toán UCS trên Level 1



Hình 4. 11: Thuật toán Best First Search trên Level 1



Hình 4. 12: Thuật toán A trên Level 1*

Kết quả được tóm tắt qua bảng dưới đây

	Chi phí đường đi (bước)	Thời gian (giây)	Số nút đã mở rộng (nút)
BFS	106	37	1524
DFS	296	40	399
UCS	106	37	1533
Best First Search	118	17	253
A*	106	17	341

*** Chi phí đường đi:**

BFS, UCS, và A* đều tìm được đường đi tối ưu với chi phí 106. DFS cho chi phí cao nhất (296), do đặc trưng đi sâu ngẫu nhiên nên thường rơi vào các nhánh dài không tối ưu. Best First Search cho chi phí trung gian (118), thể hiện khả năng tìm lời giải nhanh nhưng chưa đảm bảo tối ưu do phụ thuộc vào heuristic. Nhận xét: A*, BFS và UCS đạt hiệu quả tối ưu về chi phí, nhưng A* đạt kết quả này với ít nút hơn, chứng tỏ heuristic phát huy tác dụng.

*** Thời gian thực thi:**

Best First Search và A* có thời gian thực thi nhanh nhất (17 giây). BFS

và UCS mất thời gian tương đương (37 giây) do mở rộng gần như toàn bộ không gian trạng thái. DFS có thời gian lâu nhất (40 giây) vì đi sâu nhưng dễ phải backtrack. Nhận xét: Thuật toán có sử dụng heuristic (Best First Search, A*) rút ngắn đáng kể thời gian tìm kiếm so với các thuật toán mù (BFS, DFS, UCS).

*** Số nút mở rộng:**

Best First Search mở ít nút nhất (253), tiếp theo là A* (341). BFS và UCS mở hơn 1500 nút, phản ánh tính chất "mở rộng đều" không định hướng. DFS mở ít nút hơn BFS nhưng không hiệu quả do chất lượng lời giải kém. Nhận xét: Các thuật toán có định hướng bằng heuristic (A*, Best First Search) giúp giảm đáng kể số lượng nút cần duyệt, chứng tỏ tính khả thi cao khi mở rộng bài toán lớn.

4.4.2. Sử dụng nhiều map và tính trung bình

Nhóm chúng em tiếp tục so sánh 5 thuật toán dựa trên 3 thông số là thời gian, chi phí và số nút được mở rộng. Ở đây dùng dữ liệu của 8 Level từ Level 1 đến Level 8 với kích thước và độ khó tăng dần qua các Level, lấy trung bình thông số theo từng thuật toán để so sánh kết quả.

Thời gian (giây)					
	BFS	DFS	UCS	Best First Search	A*
Level 1	37	40	37	17	17
Level 2	44	36	44	27	14
Level 3	43	68	47	18	12
Level 4	64	62	65	35	30
Level 5	64	89	64	38	27
Level 6	102	87	103	60	29
Level 7	118	122	119	62	28
Level 8	121	130	128	68	32
Trung bình	74	79	76	41	24

Chi phí (nút)					
	BFS	DFS	UCS	Best First Search	A*
Level 1	106	296	106	118	106
Level 2	90	238	90	114	90
Level 3	88	510	88	122	88
Level 4	150	426	150	162	150
Level 5	156	642	156	246	156
Level 6	162	626	162	322	162
Level 7	158	900	158	286	158
Level 8	184	894	184	368	184
Trung bình	137	567	137	217	137

Số nút đã mở rộng (nút)					
	BFS	DFS	UCS	Best First Search	A*
Level 1	1524	399	1533	253	341
Level 2	2060	494	2059	840	223
Level 3	1948	593	1958	242	169
Level 4	2841	791	2851	1006	809
Level 5	2784	926	2801	604	608
Level 6	4462	884	4479	1428	653
Level 7	5623	1121	5637	1757	603
Level 8	5808	1658	5819	1571	711
Trung bình	3381	858	3392	963	515

*** So sánh về thời gian thực thi:**

Kết quả cho thấy A* có thời gian trung bình ngắn nhất (24 giây), vượt trội so với các thuật toán còn lại. Tiếp đến là Best First Search (41 giây), trong khi BFS và UCS có thời gian tương đương (74–76 giây), và DFS chậm nhất (79 giây).

Điều này phản ánh rõ vai trò của heuristic trong việc định hướng tìm kiếm: A* sử dụng hàm đánh giá $\boldsymbol{f(n) = g(n) + h(n)}$ cân bằng giữa chi phí đã đi và ước lượng còn lại, giúp hội tụ nhanh đến lời giải tối ưu. Best First Search chỉ dựa vào $\boldsymbol{h(n)}$, nên tìm lời giải nhanh nhưng không luôn tối ưu.

Trong khi đó, BFS và UCS phải duyệt đều toàn không gian, dẫn đến thời

gian tăng đáng kể khi kích thước và độ phức tạp của mê cung tăng. DFS có xu hướng đi sâu theo nhánh, nên mặc dù mở ít nút hơn BFS, tổng thời gian lại kéo dài do tốn công backtrack.

Kết luận: Thuật toán có heuristic (A*, Best First) đạt hiệu suất cao nhất về thời gian, đặc biệt khi độ phức tạp của môi trường tăng dần qua các Level.

*** So sánh về chi phí đường đi:**

Cả A*, BFS và UCS đều cho cùng một chi phí trung bình là 137, chứng tỏ chúng đều tìm được đường đi tối ưu. Ngược lại, Best First Search có chi phí cao hơn (217), và DFS cho kết quả kém nhất (567).

Điều này phù hợp với lý thuyết: BFS và UCS đảm bảo tối ưu nếu chi phí bước đi là đồng nhất. A* đạt tối ưu nhờ heuristic chấp nhận được (admissible). Best First Search và DFS không xét chi phí thực tế khi mở rộng nút, nên thường dừng sớm ở lời giải cục bộ, làm tăng tổng chi phí.

Kết luận: Trong tiêu chí tối ưu đường đi, A*, BFS, và UCS là ba thuật toán mạnh nhất, trong đó A* đạt hiệu quả cao nhất khi xét thêm yếu tố thời gian.

*** So sánh về số nút đã mở rộng:**

Kết quả trung bình cho thấy DFS mở ít nút hơn BFS/UCS (858 so với khoảng 3380), nhưng lại không hiệu quả do lời giải kém tối ưu. Trong khi đó, A* mở ít nút nhất (515), thấp hơn gần 7 lần so với BFS và UCS. Best First Search đứng thứ ba với 963 nút trung bình.

Điều này khẳng định vai trò then chốt của heuristic: A* hướng dẫn quá trình tìm kiếm tập trung vào khu vực tiềm năng, giúp giảm mạnh số lượng trạng thái duyệt mà vẫn đảm bảo lời giải tối ưu. Best First Search chỉ quan tâm đến ước lượng đích nên dù nhanh, nó có thể bỏ qua các đường tốt hơn, dẫn đến chi phí tăng. BFS và UCS mở rộng gần như toàn bộ không gian trạng thái, gây tốn tài nguyên và thời gian xử lý.

Kết luận: A* đạt hiệu quả cao nhất khi xét đồng thời tiêu chí "tốc độ" và "số nút mở rộng", thể hiện khả năng định hướng tìm kiếm hiệu quả.

Sự khác biệt rõ ràng giữa nhóm thuật toán **mù (BFS, DFS, UCS)** và **có heuristic (Best First, A*)** cho thấy việc sử dụng thông tin ước lượng đích có tác dụng lớn trong việc giảm chi phí tính toán và thời gian.

Đặc biệt, **A*** nổi bật là **thuật toán cân bằng toàn diện nhất** giữa hiệu quả tìm kiếm và tối ưu hóa đường đi, phù hợp cho các bài toán phức tạp trong môi trường không gian lớn như Maze Explorer.

CHƯƠNG 5: KẾT LUẬN

5.1. Đánh giá những kết quả đã thực hiện được

Qua quá trình nghiên cứu, thiết kế và triển khai, nhóm đã xây dựng thành công và cải tiến từ trò chơi *Maze Explorer* – một ứng dụng minh họa trực quan cho các thuật toán tìm kiếm trong trí tuệ nhân tạo. Sản phẩm không chỉ đáp ứng yêu cầu cơ bản của môn học mà còn tích hợp giao diện trực quan, dễ sử dụng, cho phép người chơi tương tác với nhiều cấp độ bản đồ (level) khác nhau.

Về mặt kỹ thuật, nhóm đã cài đặt và so sánh hiệu suất của năm thuật toán tìm kiếm kinh điển gồm: BFS, DFS, UCS, Best First Search và A*. Kết quả thực nghiệm cho thấy A* đạt hiệu quả cao nhất khi xét đồng thời các tiêu chí về tốc độ, số nút mở rộng và chi phí tìm kiếm, nhờ khả năng tận dụng hàm heuristic để định hướng quá trình tìm kiếm. Trong khi đó, các thuật toán mù (BFS, DFS, UCS) tuy đảm bảo tính đầy đủ nhưng tiêu tốn nhiều thời gian và tài nguyên hơn.

Sản phẩm được hoàn thiện với cấu trúc mô-đun rõ ràng theo mô hình *Scene-based*, bao gồm các màn hình chức năng như Menu, Level Select, Level, History và Ending Scene. Tính năng lưu trữ và hiển thị lịch sử chơi được thực hiện thông qua tệp *stats.json*, giúp người dùng dễ dàng theo dõi tiến trình và kết quả. Bên cạnh đó, hệ thống bản đồ được đọc từ file *.txt* giúp dễ dàng mở rộng và chỉnh sửa.

Nhìn chung, project đã đạt được các mục tiêu ban đầu: minh họa hoạt động của các thuật toán tìm kiếm, trực quan hóa quá trình tìm đường, và tạo ra một sản phẩm có tính ứng dụng, mang tính giáo dục và giải trí cao.

5.2. Định hướng phát triển

Mặc dù sản phẩm đã hoàn thiện về các chức năng cơ bản, nhóm nhận thấy vẫn còn nhiều tiềm năng để mở rộng và tối ưu hóa trong tương lai. Việc phát triển thêm các tính năng mới không chỉ giúp trò chơi trở nên sinh động và hấp dẫn hơn, mà còn góp phần nâng cao giá trị học thuật, tạo điều kiện để người dùng hiểu sâu hơn về các thuật toán trí tuệ nhân tạo thông qua trải nghiệm trực quan.

Trước hết, nhóm hướng tới việc mở rộng mô hình trò chơi bằng cách bổ sung các yếu tố tương tác mới như vật cản động, kẻ thù (*enemy*), hoặc vật phẩm thưởng. Những yếu tố này sẽ giúp tăng tính thử thách và chiến lược cho người

chơi, đồng thời tạo ra những tình huống phức tạp hơn để kiểm chứng hiệu quả của các thuật toán tìm kiếm. Qua đó, trò chơi không chỉ mang tính giải trí mà còn có thể được sử dụng như một công cụ mô phỏng có giá trị nghiên cứu.

Bên cạnh đó, nhóm dự kiến tích hợp các kỹ thuật trí tuệ nhân tạo nâng cao như *Machine Learning* và *Reinforcement Learning*. Việc áp dụng các phương pháp này sẽ cho phép hệ thống tự động học hỏi và tối ưu chiến lược tìm đường thay vì chỉ dựa vào các thuật toán tĩnh. Đây là bước tiến quan trọng giúp sản phẩm tiến gần hơn tới những ứng dụng thực tế của AI trong môi trường tự động hóa và robot di chuyển.

Ngoài ra, nhóm cũng đề xuất tối ưu hóa thuật toán và giao diện người dùng. Về mặt kỹ thuật, việc nâng cấp hàm heuristic có thể giúp cải thiện tốc độ tìm kiếm và giảm chi phí tính toán trong các bản đồ lớn. Về mặt trải nghiệm, giao diện sẽ được thiết kế thân thiện, hiệu ứng động nhằm mang lại cảm giác trực quan, sinh động và cuốn hút hơn cho người chơi.

Trong tương lai, nhóm cũng mong muốn phát triển tính năng cộng đồng cho trò chơi. Cụ thể, xây dựng một hệ thống lưu trữ trực tuyến để người dùng có thể chia sẻ, đánh giá và tải xuống các bản đồ do người khác tạo ra. Điều này sẽ khuyến khích sự sáng tạo và tương tác giữa các người chơi, đồng thời giúp mở rộng kho dữ liệu huấn luyện phục vụ cho việc thử nghiệm các thuật toán khác nhau.

Cuối cùng, nhóm định hướng ứng dụng trò chơi trong giảng dạy. Bằng cách mở rộng *Maze Explorer* thành một công cụ mô phỏng học tập (*AI Visualization Tool*), giảng viên và sinh viên ngành Công nghệ thông tin có thể sử dụng sản phẩm để minh họa trực quan hoạt động của các thuật toán tìm kiếm. Qua đó, người học có thể dễ dàng quan sát, so sánh hiệu quả giữa các thuật toán, từ đó tăng khả năng tiếp thu và tư duy phân tích.

Tổng thể, những định hướng trên không chỉ giúp mở rộng phạm vi ứng dụng của sản phẩm mà còn góp phần nâng cao tính thực tiễn, khuyến khích tinh thần sáng tạo, tự học và nghiên cứu sâu hơn về lĩnh vực trí tuệ nhân tạo - đúng với mục tiêu ban đầu của dự án.

TÀI LIỆU THAM KHẢO

1. Stuart Russell & Peter Norvig. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson Education Limited.

2. Kong, Q., Siauw, T., & Bayen, A. M. (2021). *Python programming and numerical methods: A guide for engineers and scientists*. Elsevier. ISBN 9780128195499.

3. Python Software Foundation. (2024). *Python 3.13.7 Documentation*. Truy cập từ: https://docs.python.org/3/

4. Pygame Community. (2024). *Pygame Documentation*. Truy cập từ: https://www.pygame.org/docs/

5. GeeksforGeeks. (2025). *Search Algorithms in Artificial Intelligence*. Truy cập từ: https://www.geeksforgeeks.org/search-algorithms-in-ai/

6. Wikipedia. (2025). *Pathfinding Algorithm*. Truy cập từ: https://en.wikipedia.org/wiki/Pathfinding