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, LevelScene, …; 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. Thuật toán BFS

3.1.1. Giới thiệu thuật toán BFS

Breadth-First Search (BFS) là một thuật toán tìm kiếm hoạt động bằng cách khám phá tất cả các nút ở cùng một mức độ trước khi chuyển sang mức tiếp theo. Trong bài toán Maze Explorer, BFS được sử dụng để tìm đường đi ngắn nhất từ điểm bắt đầu (S) đến cửa ra (G) sau khi thu thập tất cả các ngôi sao. BFS sử dụng cấu trúc dữ liệu Queue (FIFO). Thuật toán này đảm bảo tìm được đường đi ngắn nhất khi tất cả các cạnh có cùng trọng số.

3.1.2. Ý tưởng thuật toán

Thuật toán BFS hoạt động dựa trên nguyên lý khởi tạo từ trạng thái ban đầu tại vị trí S với mask bằng 0, sau đó mở rộng từ mỗi trạng thái hiện tại để khám phá tất cả các trạng thái kề có thể. Các trạng thái được xử lý theo thứ tự FIFO, nghĩa là trạng thái nào vào queue trước sẽ được xử lý trước, đảm bảo việc khám phá theo từng lớp độ sâu. Mục tiêu của thuật toán là tìm trạng thái thỏa mãn điều kiện đứng tại vị trí goal và đã thu thập đủ tất cả các sao. Ý tưởng cốt lõi của BFS nằm ở cách biểu diễn trạng thái bằng bộ ba (x, y, collected_mask), trong đó (x, y) là vị trí hiện tại trên lưới và collected_mask là bitmask đánh dấu các sao đã thu thập. Việc sử dụng bitmask cho phép theo dõi việc thu thập sao một cách hiệu quả và chính xác. Queue đảm bảo các trạng thái được xử lý theo thứ tự khoảng cách từ điểm bắt đầu, tạo ra tính chất tối ưu của thuật toán.

3.1.3. Mô tả thuật toán

```
from collections import deque
from typing import Deque, Dict, Iterable, List, Optional, Set, Tuple



Position = Tuple[int, int]
State = Tuple[int, int, int]  # (x, y, collected_mask)



def _parse_level(rows: List[str]) -> Tuple[Position, Position, List[Position],
int, int]:
    """Trích xuất S, G, danh sách sao từ ma trận ký tự.
```

```
    Ký hiệu lưới:
    - "1": tường (không đi xuyên)
    - "S": vị trí bắt đầu
    - "G": cửa/đích
    - "*": ngôi sao

    Trả về: (start, goal, stars, width, height)
    """
    height = len(rows)
    width = len(rows[0]) if height > 0 else 0
    start: Optional[Position] = None
    goal: Optional[Position] = None
    stars: List[Position] = []

    for y in range(height):
        for x in range(width):
            ch = rows[y][x]
            if ch == "S":
                start = (x, y)
            elif ch == "G":
                goal = (x, y)
            elif ch == "*":
                stars.append((x, y))

    if start is None or goal is None:
        raise ValueError("Level không hợp lệ: thiếu S hoặc G")

    return start, goal, stars, width, height

def _is_blocked(rows: List[str], x: int, y: int, width: int, height: int) -> bool:
    if x < 0 or y < 0 or x >= width or y >= height:
        return True
    return rows[y][x] == "1"
```

```python
def _reconstruct_path(
    parents: Dict[State, Tuple[Optional[State], str]],
    end_state: State,
) -> Tuple[List[Position], List[str]]:
    """Lần ngược từ end_state để tạo path (danh sách tọa độ) và moves (UDLR)."""
    path_rev: List[Position] = []
    moves_rev: List[str] = []
    cur: Optional[State] = end_state
    while cur is not None:
        (x, y, _) = cur
        path_rev.append((x, y))
        prev, move = parents[cur]
        if move:
            moves_rev.append(move)
        cur = prev

    path_rev.reverse()
    moves_rev.reverse()
    return path_rev, moves_rev


def bfs_collect_all_stars_with_trace(rows: List[str]) -> Dict[str, object]:
    """
    Trả về thêm:
      - "expanded_order": List[(x, y)] theo thứ tự lấy ra từ hàng đợi (đã mở rộng)
    """
    start, goal, stars, width, height = _parse_level(rows)

    star_index: Dict[Position, int] = {pos: i for i, pos in enumerate(stars)}
    all_mask = (1 << len(stars)) - 1

    start_mask = 0

    queue: Deque[State] = deque()
    parents: Dict[State, Tuple[Optional[State], str]] = {}
    visited: Set[State] = set()
```

```
expanded_order: List[Position] = []

start_state: State = (start[0], start[1], start_mask)
queue.append(start_state)
parents[start_state] = (None, "")
visited.add(start_state)

directions: List[Tuple[int, int, str]] = [
    (0, -1, "U"),
    (0, 1, "D"),
    (-1, 0, "L"),
    (1, 0, "R"),
]

end_state: Optional[State] = None

while queue:
    x, y, mask = queue.popleft()
    expanded_order.append((x, y))

    if (x, y) == goal and mask == all_mask:
        end_state = (x, y, mask)
        break

    for dx, dy, move in directions:
        nx, ny = x + dx, y + dy
        if _is_blocked(rows, nx, ny, width, height):
            continue

        next_mask = mask
        pos = (nx, ny)
        if pos in star_index:
            next_mask = mask | (1 << star_index[pos])
```

```
            nxt: State = (nx, ny, next_mask)
            if nxt in visited:
                continue
            visited.add(nxt)
            parents[nxt] = ((x, y, mask), move)
            queue.append(nxt)

    if end_state is None:
        return {
            "path": [],
            "moves": [],
            "steps": 0,
            "stars_total": len(stars),
            "found": False,
            "expanded_order": expanded_order,
            "nodes_expanded": len(expanded_order),
        }

    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),
    }
```

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.

3.2. Thuật toán DFS

3.2.1. Giới thiệu thuật toán DFS

Depth-First Search (DFS) là thuật toán tìm kiếm đồ thị hoạt động bằng cách đi sâu vào một nhánh trước khi quay lại thử các nhánh khác. Trong bài toán Maze Explorer, DFS tìm đường đi từ điểm bắt đầu (S) đến cửa ra (G) sau khi thu thập tất cả các ngôi sao (*). Thuật toán này sử dụng cấu trúc dữ liệu Stack theo nguyên tắc LIFO (Last In, First Out), khám phá theo chiều sâu trước khi mở rộng chiều rộng. Khác với BFS, DFS không đảm bảo tìm được đường đi ngắn nhất nhưng có thể tìm được giải pháp nhanh hơn trong một số trường hợp và tiết kiệm bộ nhớ hơn.

3.2.2. Ý tưởng thuật toán

Thuật toán DFS hoạt động dựa trên nguyên lý khởi tạo từ trạng thái ban đầu, sau đó đi sâu vào một nhánh cụ thể trước khi quay lại thử các nhánh khác. Khi không thể đi tiếp trong nhánh hiện tại, thuật toán sẽ quay lui và thử nhánh khác, tiếp tục cho đến khi tìm được trạng thái thỏa mãn điều kiện thắng. Ý tưởng cốt lõi của DFS nằm ở việc sử dụng Stack đảm bảo trạng thái được thêm vào cuối cùng sẽ được xử lý trước tiên, tạo ra hành vi "đào sâu" vào một đường đi trước khi thử các đường khác. Mặc dù có thể tìm được giải pháp nhanh trong một số trường hợp, DFS không đảm bảo tính tối ưu của đường đi tìm được.

3.2.3. Mô tả thuật toán

```
from collections import deque
from typing import Dict, List, Optional, Set, Tuple
```

```
Position = Tuple[int, int]
State = Tuple[int, int, int]  # (x, y, collected_mask)

def _parse_level(rows: List[str]) -> Tuple[Position, Position, List[Position],
int, int]:
    """Trích xuất S, G, danh sách sao từ ma trận ký tự."""
    height = len(rows)
    width = len(rows[0]) if height > 0 else 0
    start: Optional[Position] = None
    goal: Optional[Position] = None
    stars: List[Position] = []

    for y in range(height):
        for x in range(width):
            ch = rows[y][x]
            if ch == "S":
                start = (x, y)
            elif ch == "G":
                goal = (x, y)
            elif ch == "*":
                stars.append((x, y))

    if start is None or goal is None:
        raise ValueError("Level không hợp lệ: thiếu S hoặc G")

    return start, goal, stars, width, height

def _is_blocked(rows: List[str], x: int, y: int, width: int, height: int) -> bool:
    if x < 0 or y < 0 or x >= width or y >= height:
        return True
    return rows[y][x] == "1"

def _reconstruct_path(
    parents: Dict[State, Tuple[Optional[State], str]],
```

```
    end_state: State,
) -> Tuple[List[Position], List[str]]:
    """Lần ngược từ end_state để tạo path và moves (UDLR)."""
    path_rev: List[Position] = []
    moves_rev: List[str] = []
    cur: Optional[State] = end_state
    while cur is not None:
        (x, y, _) = cur
        path_rev.append((x, y))
        prev, move = parents[cur]
        if move:
            moves_rev.append(move)
        cur = prev

    path_rev.reverse()
    moves_rev.reverse()
    return path_rev, moves_rev

def dfs_collect_all_stars_with_trace(rows: List[str]) -> Dict[str, object]:
    """Tìm đường đi bằng DFS: thu thập hết sao rồi tới cửa (G).

    - Input: rows (danh sách chuỗi ký tự của level)
    - Output: dict gồm:
        {
          "path": List[(x,y)],        # dãy ô đi qua từ S tới G
          "moves": List[str],          # 'U','D','L','R'
          "steps": int,                # số bước
          "stars_total": int,
          "found": bool,               # có tìm thấy hay không
          "expanded_order": List[(x,y)]  # thứ tự các ô được mở rộng
        }
    """
    start, goal, stars, width, height = _parse_level(rows)

    # Ánh xạ vị trí sao -> bit index
```

```
    star_index: Dict[Position, int] = {pos: i for i, pos in enumerate(stars)}
    all_mask = (1 << len(stars)) - 1

    start_mask = 0

    # DFS sử dụng stack (LIFO)
    stack: List[State] = []
    parents: Dict[State, Tuple[Optional[State], str]] = {}
    visited: Set[State] = set()
    expanded_order: List[Position] = []

    start_state: State = (start[0], start[1], start_mask)
    stack.append(start_state)
    parents[start_state] = (None, "")
    visited.add(start_state)

    directions: List[Tuple[int, int, str]] = [
        (0, -1, "U"),
        (0, 1, "D"),
        (-1, 0, "L"),
        (1, 0, "R"),
    ]

    end_state: Optional[State] = None

    while stack:
        x, y, mask = stack.pop()  # LIFO: lấy phần tử cuối
        expanded_order.append((x, y))

        # Điều kiện thắng: đứng ở G và đã gom đủ sao
        if (x, y) == goal and mask == all_mask:
            end_state = (x, y, mask)
            break

        for dx, dy, move in directions:
```

```
            nx, ny = x + dx, y + dy
            if _is_blocked(rows, nx, ny, width, height):
                continue

            next_mask = mask
            pos = (nx, ny)
            if pos in star_index:
                next_mask = mask | (1 << star_index[pos])

            nxt: State = (nx, ny, next_mask)
            if nxt in visited:
                continue

            visited.add(nxt)
            parents[nxt] = ((x, y, mask), move)
            stack.append(nxt)

    if end_state is None:
        return {
            "path": [],
            "moves": [],
            "steps": 0,
            "stars_total": len(stars),
            "found": False,
            "expanded_order": expanded_order,
            "nodes_expanded": len(expanded_order),
        }

    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),
    }
```

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.

3.3. Thuật toán UCS

3.3.1. Giới thiệu thuật toán UCS

Uniform Cost Search (UCS) là thuật toán tìm kiếm đồ thị tối ưu hoạt động bằng cách luôn chọn trạng thái có chi phí thấp nhất để mở rộng tiếp. Trong bài toán Maze Explorer, UCS tìm đường đi ngắn nhất từ điểm bắt đầu (S) đến điểm kết thúc (G) sau khi thu thập tất cả các sao (*). Thuật toán này sử dụng cấu trúc dữ liệu Priority Queue (Min-Heap) để đảm bảo tìm được đường đi có chi phí thấp nhất. Mỗi bước di chuyển trong UCS có chi phí đồng nhất, thường được đặt bằng 1, và thuật toán đảm bảo tính tối ưu của giải pháp cuối cùng.

3.3.2. Ý tưởng thuật toán

Thuật toán UCS hoạt động dựa trên nguyên lý khởi tạo từ trạng thái ban đầu với chi phí bằng 0, sau đó luôn chọn trạng thái có chi phí thấp nhất từ priority queue để mở rộng tiếp. Với mỗi trạng thái được chọn, thuật toán tính toán chi phí mới cho các trạng thái kề và cập nhật priority queue tương ứng. Mục tiêu của UCS là tìm trạng thái thỏa mãn điều kiện với chi phí tối thiểu có

thể. Ý tưởng cốt lõi của UCS nằm ở việc sử dụng priority queue đảm bảo trạng thái có chi phí thấp nhất luôn được xử lý trước, cùng với việc duy trì g-score là chi phí từ điểm bắt đầu đến trạng thái hiện tại. Thuật toán đảm bảo tính tối ưu khi tất cả chi phí đều dương và không có chu trình âm.

3.3.3. Mô tả thuật toán

```
from collections import deque
from typing import Dict, List, Optional, Set, Tuple
from heapq import heappush, heappop

Position = Tuple[int, int]
State = Tuple[int, int, int]  # (x, y, collected_mask)

def _parse_level(rows: List[str]) -> Tuple[Position, Position, List[Position],
int, int]:
    """Trích xuất S, G, danh sách sao từ ma trận ký tự."""
    height = len(rows)
    width = len(rows[0]) if height > 0 else 0
    start: Optional[Position] = None
    goal: Optional[Position] = None
    stars: List[Position] = []

    for y in range(height):
        for x in range(width):
            ch = rows[y][x]
            if ch == "S":
                start = (x, y)
            elif ch == "G":
                goal = (x, y)
            elif ch == "*":
                stars.append((x, y))

    if start is None or goal is None:
        raise ValueError("Level không hợp lệ: thiếu S hoặc G")
```

```
    return start, goal, stars, width, height


def _is_blocked(rows: List[str], x: int, y: int, width: int, height: int) -> bool:
    if x < 0 or y < 0 or x >= width or y >= height:
        return True
    return rows[y][x] == "1"


def _reconstruct_path(
    parents: Dict[State, Tuple[Optional[State], str]],
    end_state: State,
) -> Tuple[List[Position], List[str]]:
    """Lần ngược từ end_state để tạo path và moves (UDLR)."""
    path_rev: List[Position] = []
    moves_rev: List[str] = []
    cur: Optional[State] = end_state
    while cur is not None:
        (x, y, _) = cur
        path_rev.append((x, y))
        prev, move = parents[cur]
        if move:
            moves_rev.append(move)
        cur = prev

    path_rev.reverse()
    moves_rev.reverse()
    return path_rev, moves_rev


def ucs_collect_all_stars_with_trace(rows: List[str]) -> Dict[str, object]:
    """Tìm đường đi ngắn nhất bằng UCS: thu thập hết sao rồi tới cửa (G).

    - Input: rows (danh sách chuỗi ký tự của level)
    - Output: dict gồm:
        {
          "path": List[(x,y)],        # dãy ô đi qua từ S tới G
          "moves": List[str],          # 'U','D','L','R'
```

```
      "steps": int,                # số bước
      "stars_total": int,
      "found": bool,               # có tìm thấy hay không
      "expanded_order": List[(x,y)]  # thứ tự các ô được mở rộng
    }
"""
start, goal, stars, width, height = _parse_level(rows)

# Ánh xạ vị trí sao -> bit index
star_index: Dict[Position, int] = {pos: i for i, pos in enumerate(stars)}
all_mask = (1 << len(stars)) - 1

start_mask = 0

# Hàng đợi ưu tiên cho UCS (min-heap): (g_score, state)
queue: List[Tuple[int, State]] = []
parents: Dict[State, Tuple[Optional[State], str]] = {}
g_scores: Dict[State, int] = {}  # Chi phí từ start đến state
closed_set: Set[State] = set()
expanded_order: List[Position] = []

start_state: State = (start[0], start[1], start_mask)
heappush(queue, (0, start_state))
parents[start_state] = (None, "")
g_scores[start_state] = 0
# closed_set sẽ chứa các state đã pop ra và xử lý xong

directions: List[Tuple[int, int, str]] = [
    (0, -1, "U"),
    (0, 1, "D"),
    (-1, 0, "L"),
    (1, 0, "R"),
]

end_state: Optional[State] = None
```

```
while queue:
    g_score, (x, y, mask) = heappop(queue)
    # Bỏ qua nếu state này đã được đóng trước đó
    if (x, y, mask) in closed_set:
        continue
    closed_set.add((x, y, mask))
    expanded_order.append((x, y))

    # Điều kiện thắng: đứng ở G và đã gom đủ sao
    if (x, y) == goal and mask == all_mask:
        end_state = (x, y, mask)
        break

    for dx, dy, move in directions:
        nx, ny = x + dx, y + dy
        if _is_blocked(rows, nx, ny, width, height):
            continue

        next_mask = mask
        pos = (nx, ny)
        if pos in star_index:
            next_mask = mask | (1 << star_index[pos])

        nxt: State = (nx, ny, next_mask)
        if nxt in closed_set:
            continue

        # Tính chi phí g (từ start đến nxt)
        tentative_g = g_scores[(x, y, mask)] + 1

        # Chỉ cập nhật và đẩy vào heap nếu tìm thấy đường tốt hơn
        if tentative_g < g_scores.get(nxt, float('inf')):
            g_scores[nxt] = tentative_g
            parents[nxt] = ((x, y, mask), move)
```

```
                heappush(queue, (tentative_g, nxt))

    if end_state is None:
        return {
            "path": [],
            "moves": [],
            "steps": 0,
            "stars_total": len(stars),
            "found": False,
            "expanded_order": expanded_order,
            "nodes_expanded": len(expanded_order),
        }

    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),
    }
```

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.

3.4. Thuật toán Best First Search

3.4.1. Giới thiệu thuật toán Best First Search

Best-First Search (BFSr) là một họ thuật toán tìm kiếm dựa trên chiến lược "tham lam theo độ hứa hẹn" của trạng thái, trong đó mỗi lần lặp thuật toán sẽ chọn mở rộng nút có giá trị heuristic nhỏ nhất. Biến thể được dùng phổ biến trong bài toán tìm đường là Greedy Best-First Search, với hàm đánh giá $f(n) = h(n)$ chỉ phụ thuộc vào ước lượng khoảng cách tới mục tiêu. Trong bối cảnh Maze Explorer, một trạng thái được mô hình hóa bởi bộ $(\mathrm{x}, \mathrm{y}, \mathrm{mask})$, trong đó (x, y) là tọa độ ô lưới còn $mask$ biểu diễn tập sao đã thu thập. Do yêu cầu bài toán là phải thu thập hết các ngôi sao trước khi đến đích G, heuristic cần không chỉ phản ánh mức "gần đích" mà còn tính đến các sao chưa được lấy. Thuật toán có ưu điểm nổi bật về tốc độ thực nghiệm nhờ ưu tiên những hướng đi có vẻ gần mục tiêu; tuy nhiên, do không xét chi phí thực tế đã đi $g(n)$, Greedy Best-First Search không đảm bảo tối ưu về độ dài đường đi và có thể dễ bị kẹt trong những vùng heuristic đánh giá chưa tốt.

3.4.2. Ý tưởng thuật toán

Ý tưởng cốt lõi của Best-First Search là sử dụng một hàng đợi ưu tiên để luôn rút ra trạng thái "hứa hẹn nhất" theo thứ tự tăng dần của $h(n)$. Tại mỗi bước, thuật toán lấy nút có h nhỏ nhất để mở rộng, sinh ra các trạng thái kề theo bốn hướng di chuyển (lên, xuống, trái, phải) trên lưới rời rạc. Khi đi qua một ô chứa sao, mặt nạ $mask$ được cập nhật để ghi nhận ngôi sao đã thu thập; điều kiện dừng chỉ thỏa khi vị trí hiện tại trùng với đích G và $mask$ bao phủ toàn bộ ngôi sao của màn chơi. Để phản ánh đúng ràng buộc "ăn sao rồi mới về đích", một heuristic đơn giản nhưng hiệu quả là tổng của khoảng cách Manhattan từ vị trí hiện tại tới ngôi sao gần nhất cộng với khoảng cách Manhattan từ ngôi sao đó tới đích; nếu không còn sao, heuristic suy biến còn khoảng cách Manhattan đến đích. Cách ước lượng này khuyến khích thuật toán ưu tiên tiếp cận ngôi sao lân

cận trước khi hướng về đích, qua đó giảm hiện tượng lao thẳng tới G khi nhiệm vụ thu thập chưa hoàn thành. Trong triển khai thực tế, tập đã thăm được sử dụng để tránh lặp trạng thái, còn quy tắc hòa (tie-break) có thể dựa trên thứ tự hướng hoặc chi phí g nhỏ hơn nhằm tăng tính tái lập kết quả.

3.4.3. Mô tả thuật toán

```
from collections import deque
from typing import Deque, Dict, Iterable, List, Optional, Set, Tuple
from heapq import heappush, heappop

Position = Tuple[int, int]
State = Tuple[int, int, int]  # (x, y, collected_mask)

def _parse_level(rows: List[str]) -> Tuple[Position, Position, List[Position],
int, int]:
    """Trích xuất S, G, danh sách sao từ ma trận ký tự.

    Ký hiệu lưới:
    - "1": tường (không đi xuyên)
    - "S": vị trí bắt đầu
    - "G": cửa/đích
    - "*": ngôi sao

    Trả về: (start, goal, stars, width, height)
    """
    height = len(rows)
    width = len(rows[0]) if height > 0 else 0
    start: Optional[Position] = None
    goal: Optional[Position] = None
    stars: List[Position] = []

    for y in range(height):
        for x in range(width):
            ch = rows[y][x]
            if ch == "S":
```

```
                start = (x, y)
            elif ch == "G":
                goal = (x, y)
            elif ch == "*":
                stars.append((x, y))

    if start is None or goal is None:
        raise ValueError("Level không hợp lệ: thiếu S hoặc G")

    return start, goal, stars, width, height

def _is_blocked(rows: List[str], x: int, y: int, width: int, height: int) -> bool:
    if x < 0 or y < 0 or x >= width or y >= height:
        return True
    return rows[y][x] == "1"

def _reconstruct_path(
    parents: Dict[State, Tuple[Optional[State], str]],
    end_state: State,
) -> Tuple[List[Position], List[str]]:
    """Lần ngược từ end_state để tạo path (danh sách tọa độ) và moves (UDLR)."""
    path_rev: List[Position] = []
    moves_rev: List[str] = []
    cur: Optional[State] = end_state
    while cur is not None:
        (x, y, _) = cur
        path_rev.append((x, y))
        prev, move = parents[cur]
        if move:
            moves_rev.append(move)
        cur = prev

    path_rev.reverse()
    moves_rev.reverse()
    return path_rev, moves_rev
```

```python
def _manhattan_distance(p1: Position, p2: Position) -> int:
    """Tính khoảng cách Manhattan giữa hai điểm."""
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

def greedy_collect_all_stars_with_trace(rows: List[str]) -> Dict[str, object]:
    """Tìm đường đi bằng Greedy Best-First Search: thu thập hết sao rồi tới cửa (G).

    - Input: rows (danh sách chuỗi ký tự của level)
    - Output: dict gồm:
        {
          "path": List[(x,y)],        # dãy ô đi qua từ S tới G
          "moves": List[str],          # 'U','D','L','R'
          "steps": int,                # số bước
          "stars_total": int,
          "found": bool,               # có tìm thấy hay không
          "expanded_order": List[(x,y)]  # thứ tự các ô được mở rộng
        }
    """
    start, goal, stars, width, height = _parse_level(rows)

    # Ánh xạ vị trí sao -> bit index
    star_index: Dict[Position, int] = {pos: i for i, pos in enumerate(stars)}
    all_mask = (1 << len(stars)) - 1

    # Nếu không có sao, chỉ cần đi tới G
    start_mask = 0

    # Hàng đợi ưu tiên cho Greedy (min-heap): (h_score, state)
    queue: List[Tuple[int, State]] = []
    parents: Dict[State, Tuple[Optional[State], str]] = {}
    visited: Set[State] = set()
    expanded_order: List[Position] = []
```

```
start_state: State = (start[0], start[1], start_mask)
# Heuristic: khoảng cách Manhattan đến G + số sao chưa thu thập
h_score = _manhattan_distance(start, goal) + len(stars)
heappush(queue, (h_score, start_state))
parents[start_state] = (None, "")
visited.add(start_state)

directions: List[Tuple[int, int, str]] = [
    (0, -1, "U"),
    (0, 1, "D"),
    (-1, 0, "L"),
    (1, 0, "R"),
]

end_state: Optional[State] = None

while queue:
    _, (x, y, mask) = heappop(queue)
    expanded_order.append((x, y))

    # Điều kiện thắng: đứng ở G và đã gom đủ sao
    if (x, y) == goal and mask == all_mask:
        end_state = (x, y, mask)
        break

    for dx, dy, move in directions:
        nx, ny = x + dx, y + dy
        if _is_blocked(rows, nx, ny, width, height):
            continue

        next_mask = mask
        pos = (nx, ny)
        if pos in star_index:
            next_mask = mask | (1 << star_index[pos])
```

```
            nxt: State = (nx, ny, next_mask)
            if nxt in visited:
                continue

            # Heuristic Greedy ưu tiên tiến về ngôi sao gần nhất, rồi tới G
            remaining_stars = [s for s in stars if not (next_mask & (1 <<
star_index[s]))]
            if remaining_stars:
                nearest_star_dist = min(_manhattan_distance((nx, ny), s) for s in
remaining_stars)
                h_score = nearest_star_dist + _manhattan_distance((nx, ny), goal)
            else:
                h_score = _manhattan_distance((nx, ny), goal)

            heappush(queue, (h_score, nxt))
            visited.add(nxt)
            parents[nxt] = ((x, y, mask), move)

    if end_state is None:
        return {
            "path": [],
            "moves": [],
            "steps": 0,
            "stars_total": len(stars),
            "found": False,
            "expanded_order": expanded_order,
            "nodes_expanded": len(expanded_order),
        }

    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),
    }
```

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.

3.5. Thuật toán A*

*3.5.1. Giới thiệu thuật toán A**

A* là thuật toán tìm kiếm có thông tin kết hợp cân bằng giữa "khám phá chi phí đã đi" và "ước lượng chi phí còn lại" thông qua hàm đánh giá $f(n) = g(n) + h(n)$, trong đó $g(n)$ là chi phí từ trạng thái bắt đầu tới nút n, còn $h(n)$ là heuristic ước lượng chi phí tối thiểu từ n tới mục tiêu. Với bài toán Maze Explorer, mỗi trạng thái được mô hình hóa bởi $(x, y, mask)$, trong đó $mask$ ghi nhận tập sao đã thu thập; điều kiện dừng chỉ thỏa khi nhân vật đứng ở đích G và $mask$ bao phủ toàn bộ các sao. Điểm mạnh của A* là đảm bảo tối ưu độ dài đường đi khi h là **admissible** (không vượt quá chi phí thật) và **consistent** (thỏa tam giác), đồng thời thường mở rộng rất ít nút so với các chiến lược mù nhờ

định hướng tìm kiếm vào những vùng "hứa hẹn".

3.5.2. Ý tưởng thuật toán

Cốt lõi của A* là duy trì một hàng đợi ưu tiên theo giá trị f, luôn chọn mở rộng trạng thái có f nhỏ nhất. Để phản ánh ràng buộc "ăn hết sao rồi mới về đích", heuristic cần ước lượng chi phí nhỏ nhất để: (i) đi từ vị trí hiện tại đến một sao còn lại, (ii) kết nối các sao còn lại với nhau một cách tiết kiệm, và (iii) từ một sao trong số đó đi đến G. Một xây dựng điển hình là heuristic dựa trên cây khung nhỏ nhất (MST) trên tập sao chưa thu thập:

$\boxed{h(n) = d(cur, R) + MST(R) + d(R, G)}$ với R là tập sao còn lại, d là khoảng cách ngắn nhất trên lưới (tiền xử lý bằng BFS giữa các điểm quan tâm S, G, và các sao). Heuristic này admissible vì $MST(R)$ là cận dưới cho mọi hành trình phải ghé qua tất cả các sao, còn hai đầu mút $d(cur, R)$ và $d(R, G)$ lần lượt là chi phí tối thiểu để "vào" và "ra" khỏi tập R. Nhờ đó, A* được định hướng tốt: vừa kiểm soát quãng đường đã đi (g), vừa "nhìn xa" qua h để tránh mở rộng dư thừa.

3.5.3. Mô tả thuật toán

```
from collections import deque
from typing import Deque, Dict, Iterable, List, Optional, Set, Tuple
from heapq import heappush, heappop


Position = Tuple[int, int]
State = Tuple[int, int, int]  # (x, y, collected_mask)


def _parse_level(rows: List[str]) -> Tuple[Position, Position, List[Position],
int, int]:
    """Trích xuất S, G, danh sách sao từ ma trận ký tự.

    Ký hiệu lưới:
    - "1": tường (không đi xuyên)
    - "S": vị trí bắt đầu
    - "G": cửa/đích
    - "*": ngôi sao
```

```
    Trả về: (start, goal, stars, width, height)
    """
    height = len(rows)
    width = len(rows[0]) if height > 0 else 0
    start: Optional[Position] = None
    goal: Optional[Position] = None
    stars: List[Position] = []

    for y in range(height):
        for x in range(width):
            ch = rows[y][x]
            if ch == "S":
                start = (x, y)
            elif ch == "G":
                goal = (x, y)
            elif ch == "*":
                stars.append((x, y))

    if start is None or goal is None:
        raise ValueError("Level không hợp lệ: thiếu S hoặc G")

    return start, goal, stars, width, height

def _is_blocked(rows: List[str], x: int, y: int, width: int, height: int) -> bool:
    if x < 0 or y < 0 or x >= width or y >= height:
        return True
    return rows[y][x] == "1"

def _bfs_distance(rows: List[str], start: Position, width: int, height: int) ->
Dict[Position, int]:
    """Tính khoảng cách BFS từ start đến tất cả các ô trong lưới."""
    distances: Dict[Position, int] = {}
    queue = deque([(start, 0)])
    visited: Set[Position] = {start}
```

```
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    while queue:
        (x, y), dist = queue.popleft()
        distances[(x, y)] = dist

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if (nx, ny) not in visited and not _is_blocked(rows, nx, ny, width,
height):
                visited.add((nx, ny))
                queue.append(((nx, ny), dist + 1))

    return distances

def _precompute_distances(rows: List[str], start: Position, goal: Position, stars:
List[Position], width: int, height: int) -> Dict[Tuple[Position, Position], int]:
    """Tiền xử lý khoảng cách BFS giữa tất cả POI (S, G, stars)."""
    poi_list = [start, goal] + stars
    poi_to_index = {poi: i for i, poi in enumerate(poi_list)}
    distances: Dict[Tuple[Position, Position], int] = {}

    # Tính khoảng cách từ mỗi POI đến tất cả các ô
    poi_distances = {}
    for poi in poi_list:
        poi_distances[poi] = _bfs_distance(rows, poi, width, height)

    # Lưu khoảng cách giữa các POI
    for i, poi1 in enumerate(poi_list):
        for j, poi2 in enumerate(poi_list):
            if i != j:
                if poi2 in poi_distances[poi1]:
                    distances[(poi1, poi2)] = poi_distances[poi1][poi2]
                else:
```

```
                    # Nếu không thể đến được, dùng khoảng cách Manhattan làm upper
bound
                    distances[(poi1, poi2)] = abs(poi1[0] - poi2[0]) + abs(poi1[1]
- poi2[1])

    return distances

def _get_distance(distances: Dict[Tuple[Position, Position], int], pos1: Position,
pos2: Position) -> int:
    """Lấy khoảng cách giữa hai vị trí từ ma trận đã tính sẵn."""
    if (pos1, pos2) in distances:
        return distances[(pos1, pos2)]
    elif (pos2, pos1) in distances:
        return distances[(pos2, pos1)]
    else:
        # Fallback: Manhattan distance
        return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

def _compute_mst_weight(remaining_stars: List[Position], distances:
Dict[Tuple[Position, Position], int]) -> int:
    """Tính trọng số MST của các sao còn lại bằng thuật toán Prim."""
    if not remaining_stars:
        return 0
    if len(remaining_stars) == 1:
        return 0

    # Sử dụng Prim's algorithm
    mst_weight = 0
    visited: Set[Position] = {remaining_stars[0]}
    remaining = set(remaining_stars[1:])

    while remaining:
        min_dist = float('inf')
        min_star = None
```

```
        for visited_star in visited:
            for remaining_star in remaining:
                dist = _get_distance(distances, visited_star, remaining_star)
                if dist < min_dist:
                    min_dist = dist
                    min_star = remaining_star

        if min_star is not None:
            mst_weight += min_dist
            visited.add(min_star)
            remaining.remove(min_star)

    return mst_weight

def _compute_heuristic_mst(current_pos: Position, remaining_stars: List[Position],
goal: Position, distances: Dict[Tuple[Position, Position], int]) -> int:
    """Tính heuristic MST admissible: d(cur, R) + MST(R) + d(R, G)."""
    if not remaining_stars:
        return _get_distance(distances, current_pos, goal)

    # d(cur, R): khoảng cách ngắn nhất từ vị trí hiện tại đến một sao trong R
    min_dist_to_stars = min(_get_distance(distances, current_pos, star) for star
in remaining_stars)

    # MST(R): trọng số cây khung nhỏ nhất của các sao còn lại
    mst_weight = _compute_mst_weight(remaining_stars, distances)

    # d(R, G): khoảng cách ngắn nhất từ một sao trong R đến goal
    min_dist_stars_to_goal = min(_get_distance(distances, star, goal) for star in
remaining_stars)

    return min_dist_to_stars + mst_weight + min_dist_stars_to_goal

def _reconstruct_path(
    parents: Dict[State, Tuple[Optional[State], str]],
```

```
    end_state: State,
) -> Tuple[List[Position], List[str]]:
    """Lần ngược từ end_state để tạo path (danh sách tọa độ) và moves (UDLR)."""
    path_rev: List[Position] = []
    moves_rev: List[str] = []
    cur: Optional[State] = end_state
    while cur is not None:
        (x, y, _) = cur
        path_rev.append((x, y))
        prev, move = parents[cur]
        if move:
            moves_rev.append(move)
        cur = prev

    path_rev.reverse()
    moves_rev.reverse()
    return path_rev, moves_rev

def astar_collect_all_stars_with_trace(rows: List[str]) -> Dict[str, object]:
    """Tìm đường đi ngắn nhất bằng A* với heuristic MST admissible.

    - Input: rows (danh sách chuỗi ký tự của level)
    - Output: dict gồm:
        {
          "path": List[(x,y)],        # dãy ô đi qua từ S tới G
          "moves": List[str],          # 'U','D','L','R'
          "steps": int,                # số bước
          "stars_total": int,
          "found": bool,               # có tìm thấy hay không
          "expanded_order": List[(x,y)]  # thứ tự các ô được mở rộng
        }
    """
    start, goal, stars, width, height = _parse_level(rows)

    # Tiền xử lý khoảng cách BFS
```

```
    distances = _precompute_distances(rows, start, goal, stars, width, height)

    # Ánh xạ vị trí sao -> bit index
    star_index: Dict[Position, int] = {pos: i for i, pos in enumerate(stars)}
    all_mask = (1 << len(stars)) - 1

    # Nếu không có sao, chỉ cần đi tới G
    start_mask = 0

    # Hàng đợi ưu tiên cho A* (min-heap): (f_score, state)
    queue: List[Tuple[int, State]] = []
    parents: Dict[State, Tuple[Optional[State], str]] = {}
    g_scores: Dict[State, int] = {}  # Chi phí từ start đến state
    closed_set: Set[State] = set()  # Các state đã "đóng"
    expanded_order: List[Position] = []

    start_state: State = (start[0], start[1], start_mask)

    # Cache cho MST theo mask để tối ưu
    mst_cache: Dict[int, int] = {}

    def get_heuristic(state: State) -> int:
        """Tính heuristic MST cho state hiện tại."""
        x, y, mask = state
        current_pos = (x, y)

        # Lấy danh sách sao còn lại
        remaining_stars = [stars[i] for i in range(len(stars)) if not (mask & (1
<< i))]

        # Kiểm tra cache MST
        cache_key = mask
        if cache_key not in mst_cache:
            mst_cache[cache_key] = _compute_mst_weight(remaining_stars, distances)
```

```
        return _compute_heuristic_mst(current_pos, remaining_stars, goal,
distances)

    # Khởi tạo
    h_score = get_heuristic(start_state)
    heappush(queue, (h_score, start_state))
    parents[start_state] = (None, "")
    g_scores[start_state] = 0

    directions: List[Tuple[int, int, str]] = [
        (0, -1, "U"),
        (0, 1, "D"),
        (-1, 0, "L"),
        (1, 0, "R"),
    ]

    end_state: Optional[State] = None

    while queue:
        _, (x, y, mask) = heappop(queue)

        # Bỏ qua nếu đã xử lý state này
        if (x, y, mask) in closed_set:
            continue

        closed_set.add((x, y, mask))
        expanded_order.append((x, y))

        # Điều kiện thắng: đứng ở G và đã gom đủ sao
        if (x, y) == goal and mask == all_mask:
            end_state = (x, y, mask)
            break

        for dx, dy, move in directions:
            nx, ny = x + dx, y + dy
```

```
            if _is_blocked(rows, nx, ny, width, height):
                continue

            next_mask = mask
            pos = (nx, ny)
            if pos in star_index:
                next_mask = mask | (1 << star_index[pos])

            nxt: State = (nx, ny, next_mask)

            # Bỏ qua nếu đã đóng
            if nxt in closed_set:
                continue

            # Tính chi phí g (từ start đến nxt)
            tentative_g = g_scores[(x, y, mask)] + 1

            # Chỉ cập nhật nếu tìm được đường tốt hơn
            if tentative_g < g_scores.get(nxt, float('inf')):
                g_scores[nxt] = tentative_g
                parents[nxt] = ((x, y, mask), move)

                # Tính f_score với heuristic MST
                h_score = get_heuristic(nxt)
                f_score = tentative_g + h_score

                heappush(queue, (f_score, nxt))

    if end_state is None:
        return {
            "path": [],
            "moves": [],
            "steps": 0,
            "stars_total": len(stars),
            "found": False,
```

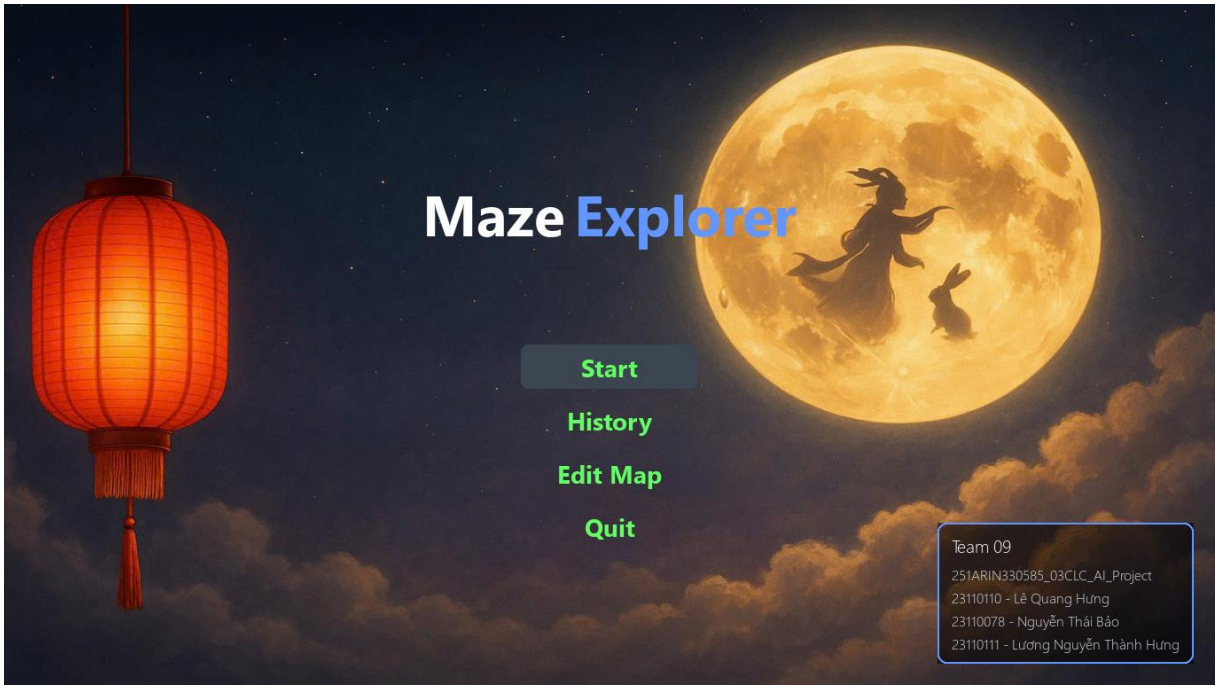
```
            "expanded_order": expanded_order,
            "nodes_expanded": len(expanded_order),
        }

    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),
    }
```

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.

CHƯƠNG 4: THỰC NGHIỆM, ĐÁNH GIÁ, PHÂN TÍCH

4.1. Xây dựng giao diện

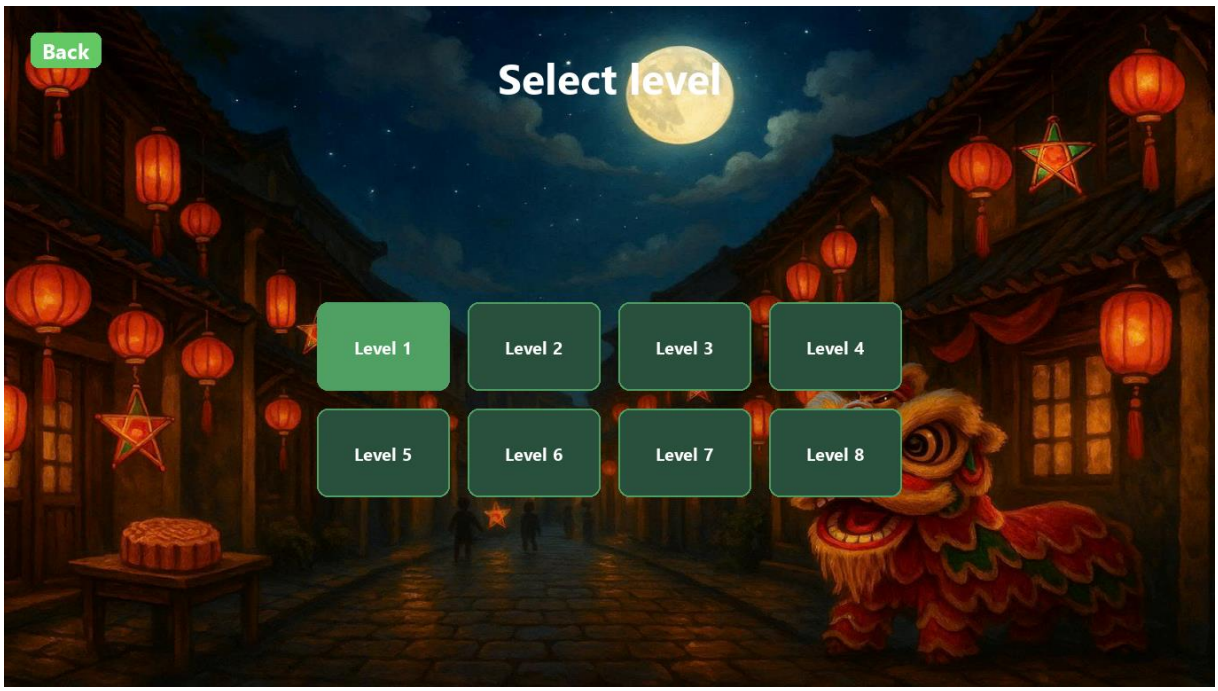
4.1.1. Giao diện màn hình chính



Hình 4. 1: Màn hình chính

Menu Scene đóng vai trò là điểm khởi đầu của trò chơi, cung cấp bốn chức năng chính là: start - khởi tạo quá trình chọn level để chơi game, history - hiển thị lịch sử các lần chơi trước đó, edit map - cho phép người chơi sửa các map hiện có hoặc tự tạo map mới theo sở thích của bản thân, quit - thoát khỏi ứng dụng. Giao diện sử dụng hệ thống điều hướng bằng mũi tên và chuột. Background của màn hình chính và một số giao diện khác được lấy cảm hứng từ Tết Trung thu.

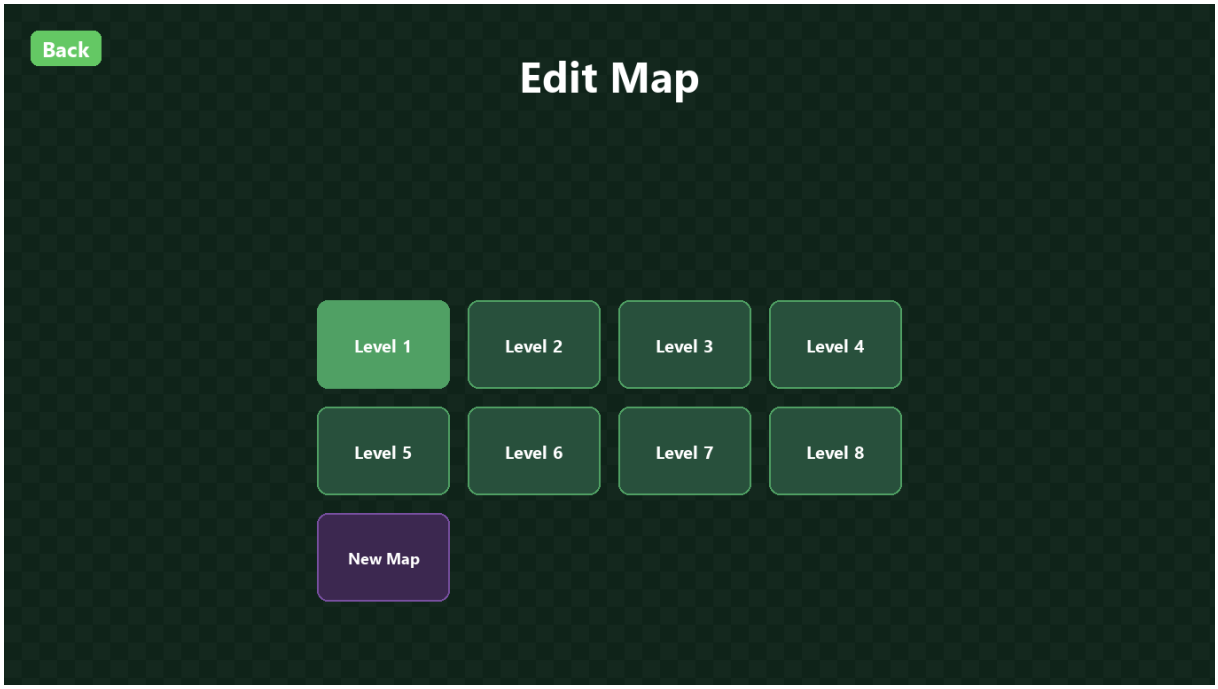
4.1.2. Giao diện chọn Level



Hình 4. 2: Màn hình chọn level

Màn hình chọn level cung cấp giao diện dạng lưới để người dùng lựa chọn level muốn chơi. Hiện tại game đang có 8 level do nhóm chúng em thiết kế (đọc từ file .txt). Hệ thống hiển thị các level dưới dạng card với hiệu ứng khi hover và chọn rõ ràng, background mang không khí của Tết Trung thu.

4.1.3. Giao diện chọn level để chỉnh sửa



Hình 4. 3: Màn hình chọn level để chỉnh sửa

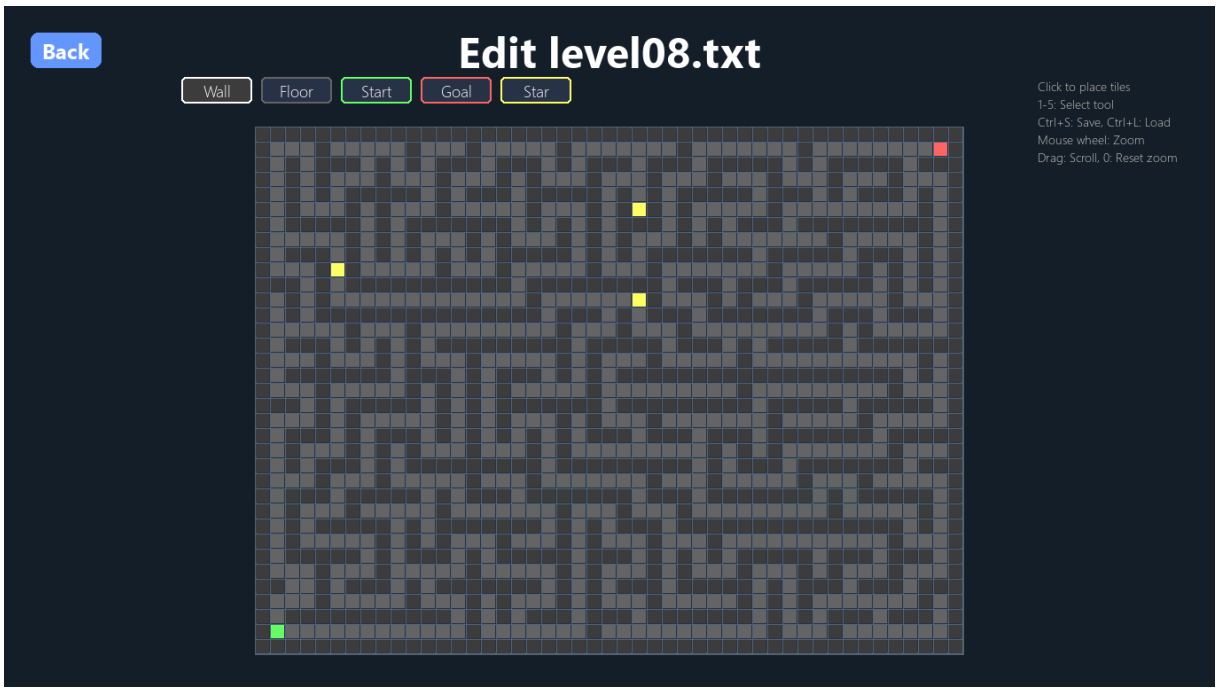
Giao diện chọn level để chỉnh sửa được thiết kế hoàn toàn tương tự như giao diện chọn level để chơi game nhưng bổ sung thêm: card "New Map" với màu tím (thể hiện sự nổi bật so với sửa map cũ) để tạo bản đồ mới theo ý của người chơi.

4.1.4. Giao diện nhập kích thước bản đồ

Hình 4. 4: Màn hình nhập kích thước bản đồ

Màn hình nhập kích thước bản đồ cung cấp form nhập liệu với: validation real-time dùng để kiểm tra tính hợp lệ của của dữ liệu đầu vào; kích thước được giới hạn từ 10x10 đến 50x50 ô để đảm bảo các thuật toán tìm kiếm được lời giải; giao diện form chuyên nghiệp với placeholder text và cursor blinking; có các hướng dẫn chi tiết ở góc trên bên phải.

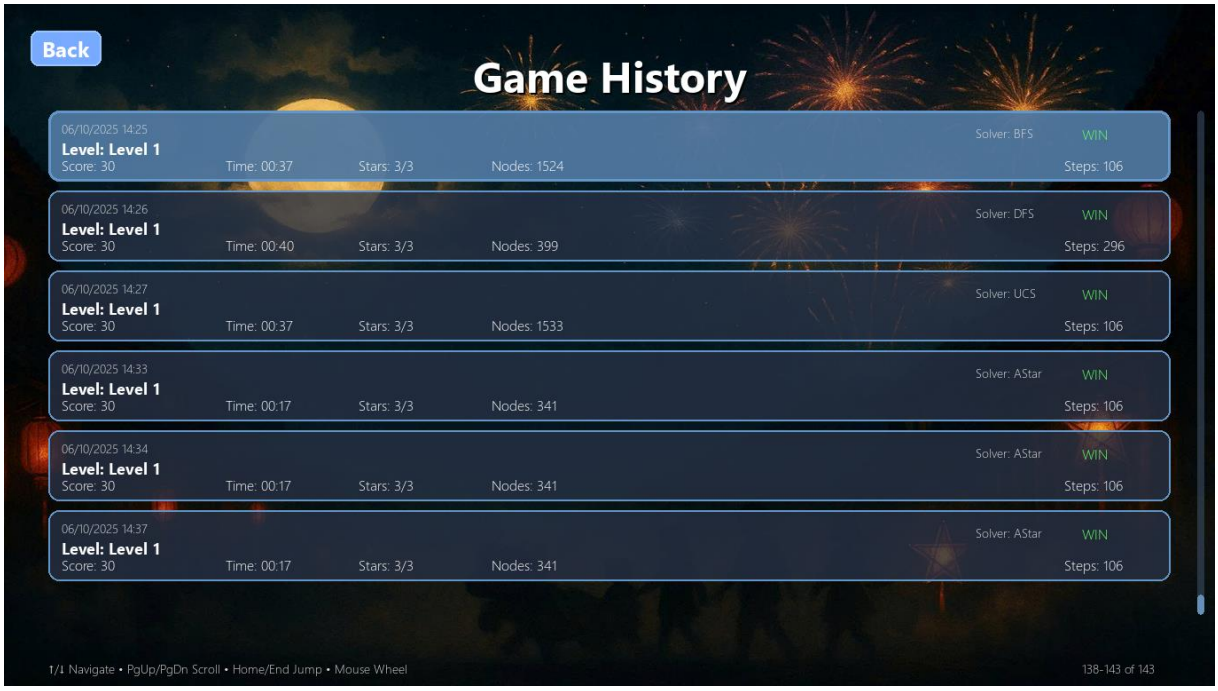
4.1.5. Giao diện chỉnh sửa bản đồ



Hình 4. 5: Màn hình chỉnh sửa bản đồ

Màn hình chỉnh sửa bản đồ là công cụ chỉnh sửa bản đồ với các tính năng: nó sẽ có 5 công cụ (Wall, Floor, Start, Goal, Star); hỗ trợ thu phóng và cuộn bản đồ để có thể tập trung chỉnh sửa một khu vực cụ thể trong map; chỉnh sửa trực tiếp bằng chuột; nhấn Ctrl + S để lưu lại map, phím số để chọn công cụ; tự động căn giữa bản đồ theo kích thước.

4.1.6. Giao diện lịch sử game



Hình 4. 6: Màn hình lịch sử game

Màn hình lịch sử game hiển thị thống kê chi tiết với 6 record mỗi trang, cuộn bằng chuột, kéo scroll bar hoặc điều hướng lên xuống bằng phím mũi tên, hiển thị thông tin chi tiết bao gồm ngày giờ, tên level, số điểm, thời gian, số bước, số sao thu thập, số nút đã duyệt khi chạy bằng thuật toán và tên thuật toán đã chơi (trường hợp người chơi thì hiển thị Solver: Human).

4.1.7. Giao diện kết thúc

Hình 4. 7: Cảnh cắt từ đoạn video kết thúc game

Khi người chơi chiến thắng ở map cuối cùng (Level 8) thì sẽ được xem 1 đoạn video về cảnh đoàn tụ của Hằng Nga và thỏ ngọc, giao diện kết thúc này sử dụng OpenCV để phát video MP4, tự động điều chỉnh kích thước theo màn hình; chương trình sẽ hiển thị thông báo lỗi nếu không thể phát video.

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. 8: Quá trình tìm kiếm lời giải của A ở Level 7*

Hình 4. 9: 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.

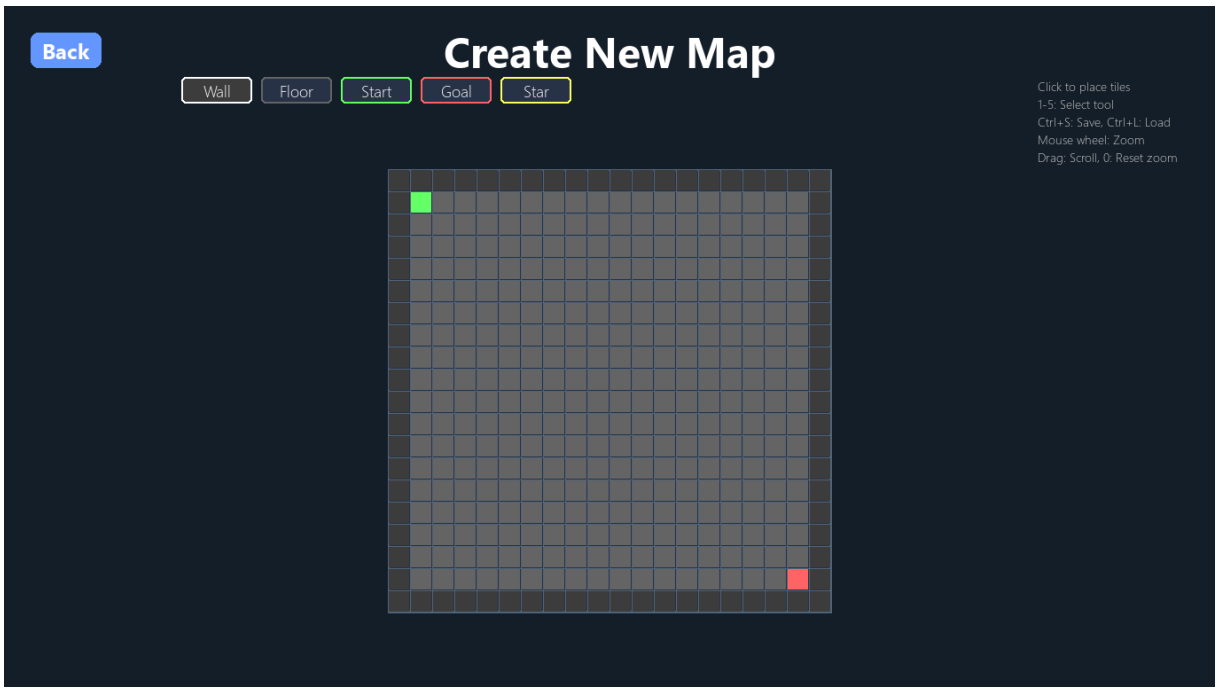
Kéo chuột: Vẽ liên tục.

Mouse wheel: Thu phóng.

Drag chuột giữa: Cuộn bản đồ.

Ctrl+S: Lưu bản đồ.

Hình 4. 10: Khởi tạo map mới với kích thước là 20x20



Hình 4. 11: Map mặc định khi khởi tạo map mới

4.2.5. Xem lịch sử

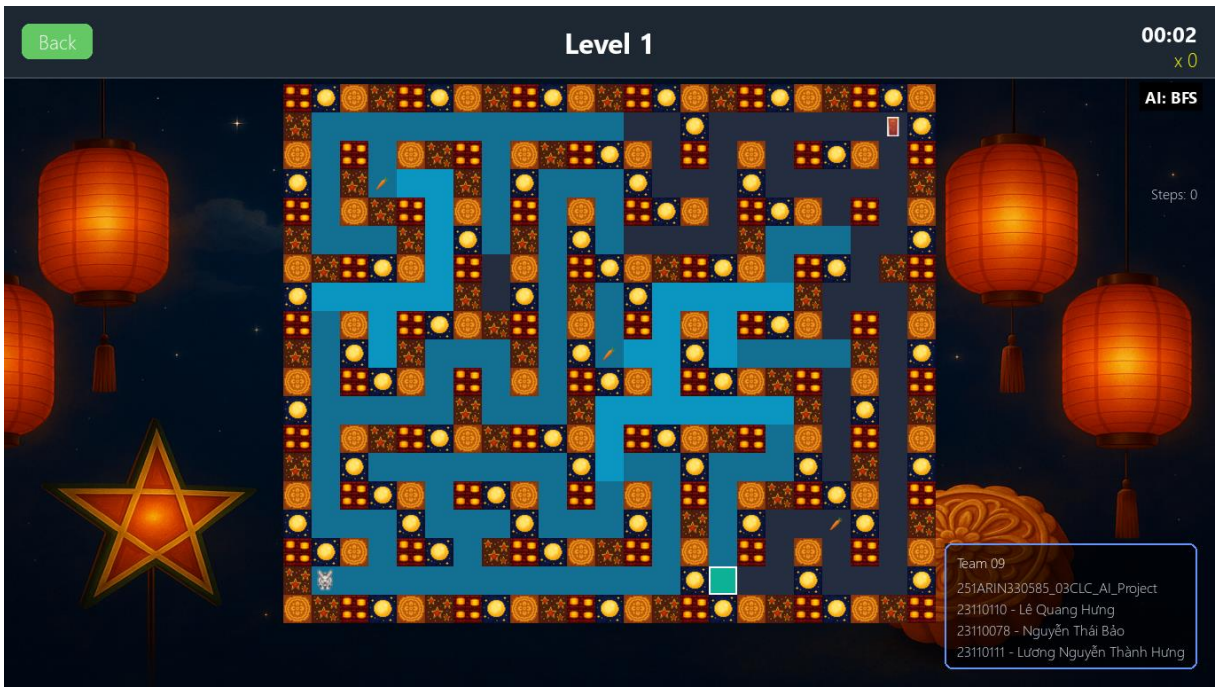
Chọn "History" từ menu chính

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.

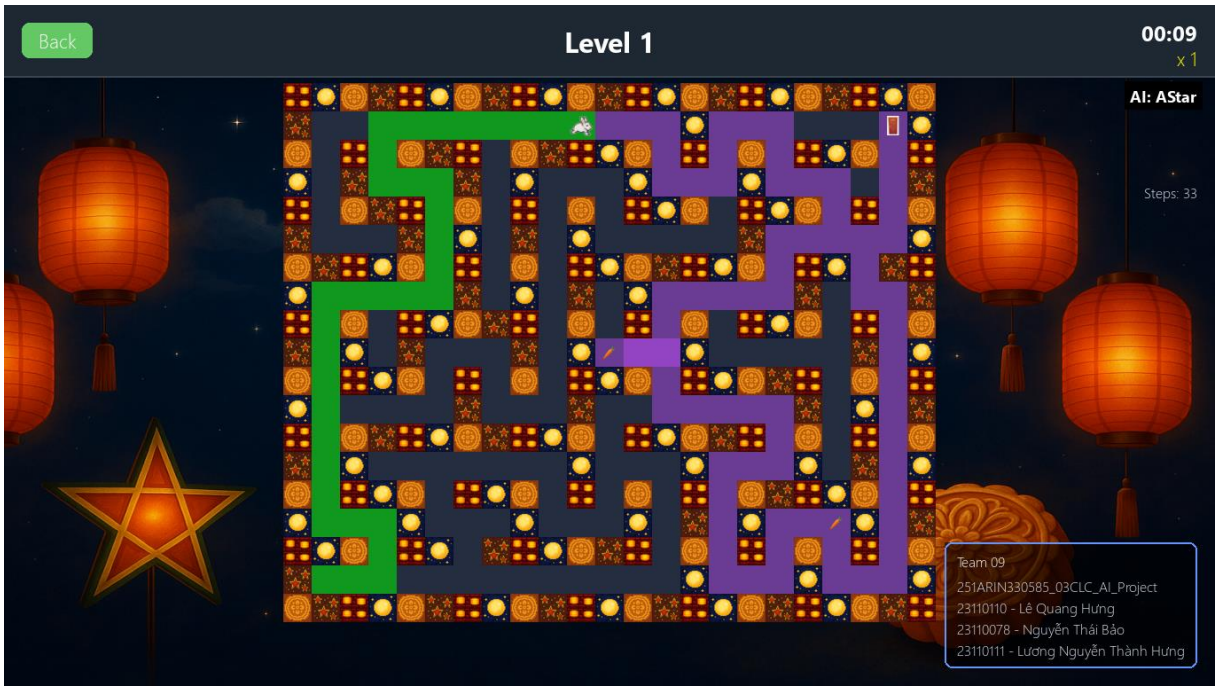
4.3. Kết quả sau khi chạy thuật toán

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. 12: Quá trình tìm kiếm lời giải khi chạy thuật toán

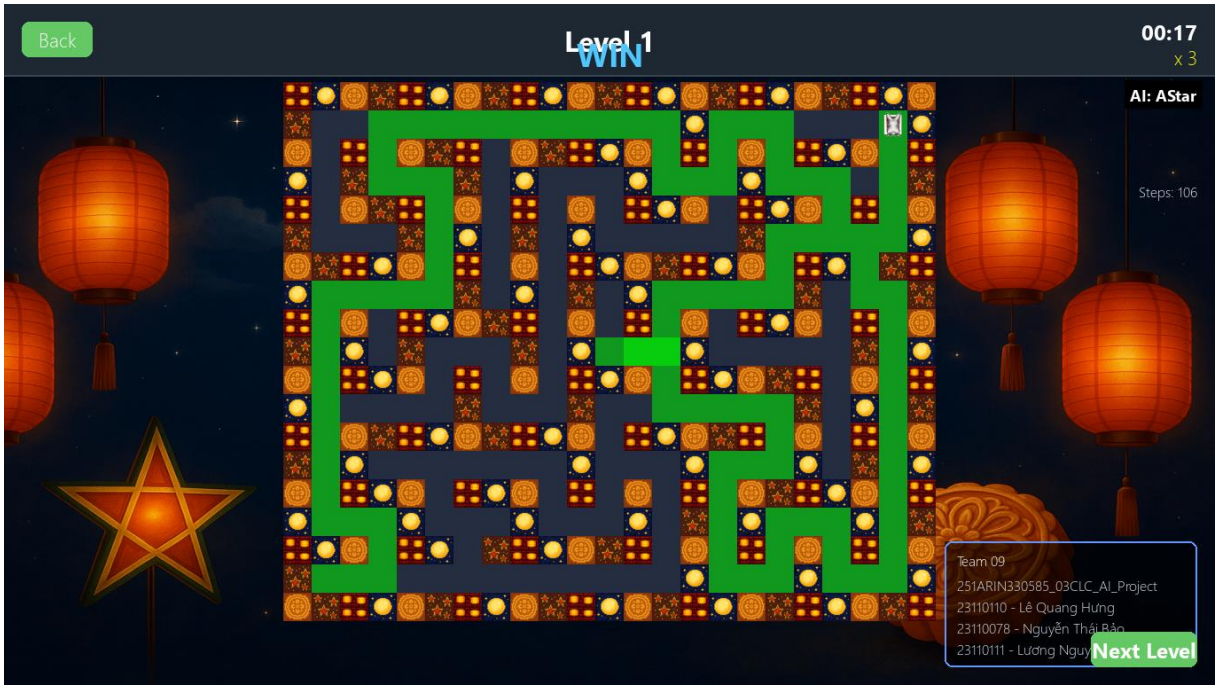
Trong quá trình tìm kiếm thì các màu được thể hiện như sau: các ô đã mở rộng thì dùng CYAN tươi (0, 200, 255, 110), alpha trung bình để nổi bật, còn ô hiện tại của tìm kiếm thì dùng teal đậm (0, 255, 200, 160) kèm viền trắng 2px.



Hình 4. 13: 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. 14: Kết quả sau khi chạy thuật toán

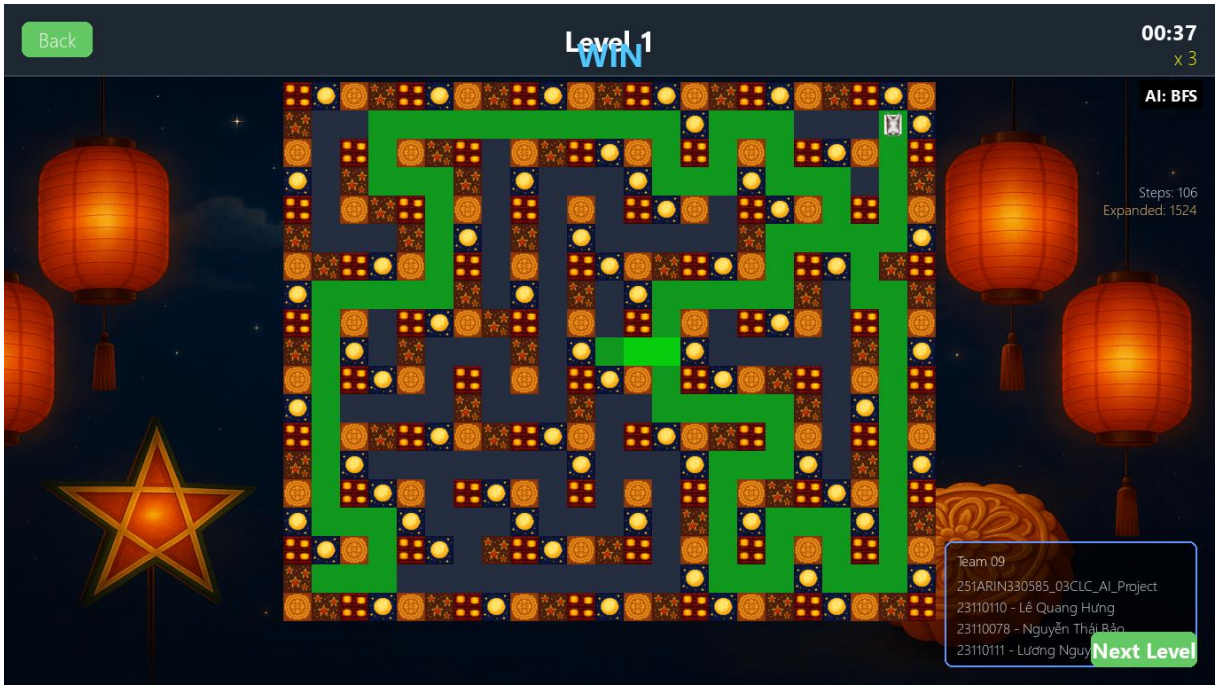
Với thuật toán A Star phía trên và map Level 1, ta thấy thuật toán mất thời gian hoàn thành là 17 giây, chi phí đường đi là 106 nút và số lượng nút đã mở rộng là 341.

4.4. So sánh các thuật toán

4.4.1. Sử dụng một Level 1

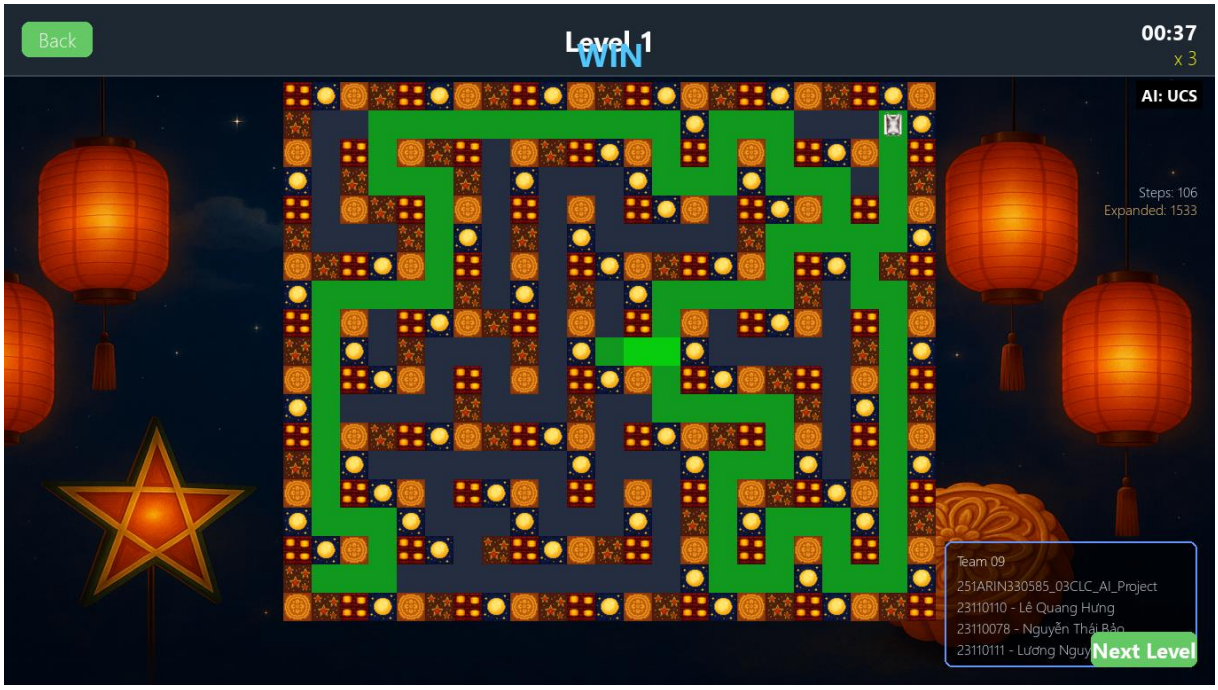
Nhóm chúng em chọn level_01.txt làm bản đồ ma trận để thể hiện 5 thuật toán được cài đặt.

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:

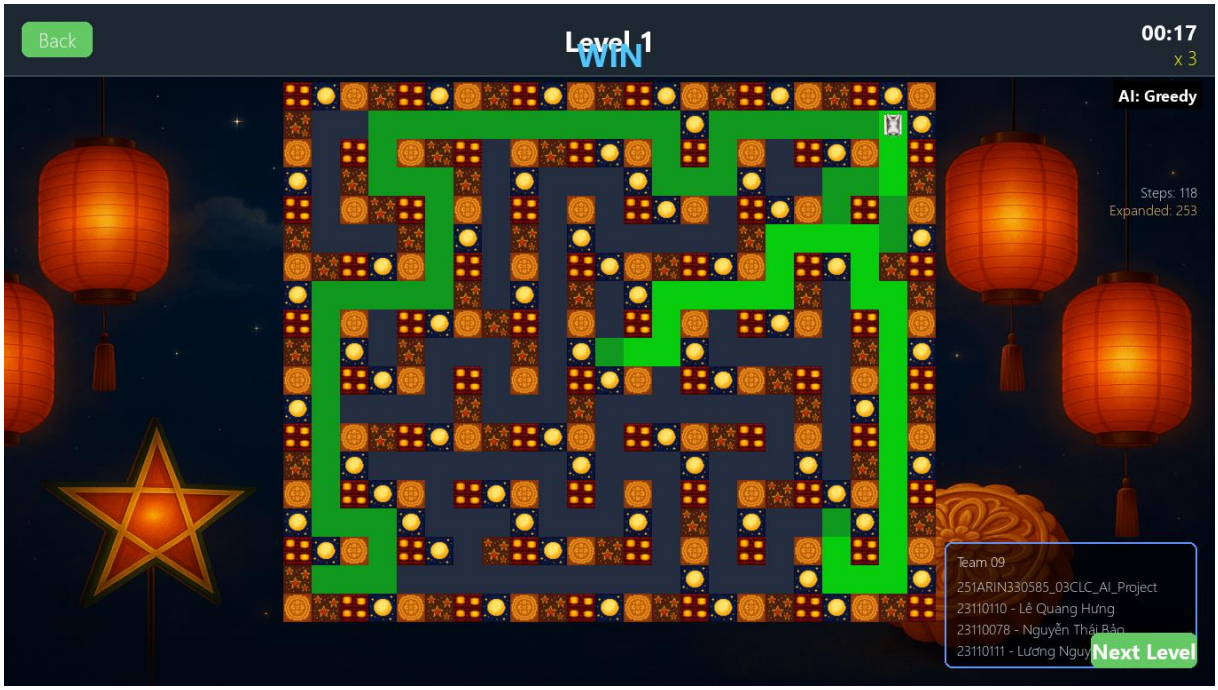


Hình 4. 15: Thuật toán BFS trên Level 1

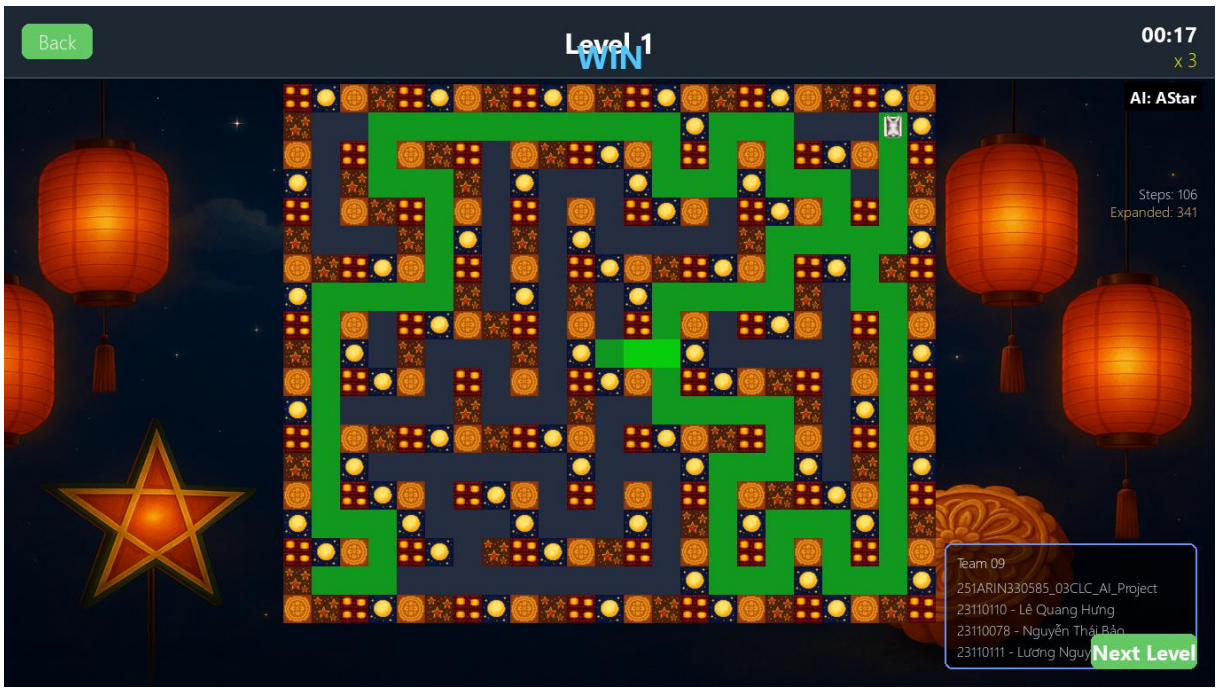
Hình 4. 16: Thuật toán DFS trên Level 1



Hình 4. 17: Thuật toán UCS trên Level 1



Hình 4. 18: Thuật toán Best First Search trên Level 1



Hình 4. 19: 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	41	399
UCS	106	37	1533
Best First Search	118	17	253
A*	106	18	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 (17–18 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 (41 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	41	37	17	18
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