VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

BÁO CÁO LAB 01: SEARCH

Cơ sở trí tuệ nhân tạo

TOPIC: Ares's adventure

Giảng viên: Lê Hoài Bắc
Trợ giảng: Nguyễn Ngọc Đức
Giảng viên dự án: Nguyễn Ngọc Đức
Sinh viên: Nguyễn Kiều Đức Vĩnh Thiên - 22120340
Nguyễn Hưng - 22120122
Lớp: 22TN

HO CHI MINH CITY, DECEMBER 2023

Mục lục

1. Thông tin thành viên

Mã sinh viên: 22120122
Họ và tên: Nguyễn Hưng
Lớp: 22TN

Mã sinh viên: 22120340
Họ và tên: Nguyễn Kiều Đức Vĩnh Thiên
Lớp: 22TN

2. Bảng phân chia công việc

Tên nhiệm vụ	Thành viên phụ trách	Tình trạng	Phần trăm hoàn thành
Thiết kế thuật các lớp logic	Nguyễn Hưng	hoàn thành	100%
Thiết kế thuật toán	Nguyễn Kiều Đức Vĩnh Thiên	hoàn thành	100%
Cài đặt thuật toán	Nguyễn Kiều Đức Vĩnh Thiên	hoàn thành	100%
Cài đặt giao diện	Nguyễn Hưng	hoàn thành	100%
Thiết kế giao diện	Nguyễn Hưng	hoàn thành	100%
Viết báo cáo	Nguyễn Kiều Đức Vĩnh Thiên	hoàn thành	100%
Quay demo	Nguyễn Hưng	hoàn thành	100%
Thiết kế testcase	Nguyễn Kiều Đức Vĩnh Thiên	hoàn thành	100%
Thống kê kết quả	Nguyễn Hưng	hoàn thành	100%

Bảng 1: Bảng phân công công việc

3. Tự đánh giá

Sau khi hoàn thành dự án tìm kiếm cho Ares, nhóm chúng em đã học được nhiều bài học quan trọng về việc áp dụng các thuật toán tìm kiếm như Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS) và A* Search.

Việc hiện thực hóa các thuật toán này đã giúp chúng em nâng cao kỹ năng lập trình Python, đồng thời phải hiểu sâu về cách hoạt động và ưu nhược điểm của từng thuật toán. Điều này không chỉ hữu ích trong dự án này, mà còn là nền tảng quan trọng cho việc áp dụng các thuật toán tìm kiếm vào các bài toán thực tế khác trong tương lai.

Bên cạnh đó, việc thiết kế và thực hiện giao diện người dùng (GUI) cũng là một thách thức đáng kể. Chúng em đã phải tìm hiểu các thư viện hỗ trợ GUI, đồng thời cân nhắc cách thể hiện trực quan và sinh động các bước đi của Ares trong mê cung. Kết quả là một giao diện trực quan, dễ sử dụng và cung cấp đầy đủ thông tin về quá trình giải quyết bài toán.

Về mặt quản lý dự án, nhóm chúng em đã phân công công việc một cách hợp lý, đảm bảo mọi thành viên đều đóng góp và hoàn thành tốt nhiệm vụ được giao. Chúng em cũng thường xuyên trao đổi, chia sẻ ý tưởng và hỗ trợ lẫn nhau để đảm bảo chất lượng của sản phẩm cuối cùng.

Tuy nhiên, trong quá trình thực hiện, chúng em cũng gặp một số thách thức như việc lựa chọn *heuristic* phù hợp cho A* Search, hay tối ưu hóa hiệu suất của các thuật toán. Điều này đòi hỏi chúng em phải nghiên cứu sâu hơn, thử nghiệm nhiều phương án khác nhau để tìm ra giải pháp tối ưu nhất có thể.

Tất nhiên không thể tránh khỏi những thiếu sót, và còn nhiều những chức năng mà nhóm em dự định thêm vào nhưng chưa kịp. Trong tương lai, chương trình này có thể có thêm nhiều chức năng như cho phép người dùng nhập vào mê cung, chế độ cho phép người dùng tự chơi,...

Về hiệu năng, chương trình có thể tối ưu hơn trong tương lai ở những điểm sau: phát hiện deadlock và chọn hàm heuristic tốt hơn.

Tổng kết lại, dự án này đã giúp nhóm chúng em không chỉ nâng cao kỹ năng lập trình, mà còn rèn luyện khả năng làm việc nhóm, giải quyết vấn đề và tư duy sáng tạo. Đây là những kỹ năng vô cùng quan trọng không chỉ trong học tập, mà còn trong công việc trong tương lai.

4. Mô tả bài toán

4.1. Giới thiệu

Theo truyền thuyết, ẩn sâu trong một vương quốc xa xôi là một cánh cổng dẫn đến kho báu bí ẩn. Người ta nói rằng để mở được cánh cổng, phải giải một mê cung bằng cách di chuyển những tảng đá nặng lên các công tắc bí mật. Dù điều này nghe có vẻ đơn giản, nhưng đã có nhiều người thử và thất bại, vì mê cung rất phức tạp và việc di chuyển những tảng đá lớn qua các không gian chật hẹp rất khó khăn. Chỉ một sai lầm nhỏ cũng có thể khiến người thử thách bị mắc kẹt trong mê cung mãi mãi.

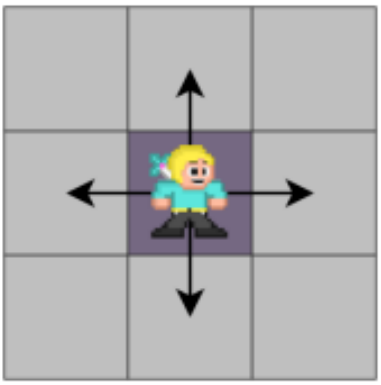
Một ngày nọ, một nhà thám hiểm trẻ tên là Ares quyết định đối mặt với thử thách này. Anh biết rằng để thành công, cần phải lập kế hoạch cẩn thận, chú ý đến từng bước đi, từng chi tiết và không bao giờ bỏ cuộc. Nhiệm vụ của bạn là giúp Ares trong cuộc phiêu lưu này. Sử dụng các thuật toán tìm kiếm mà bạn đã học trong khóa học Nhập môn Trí tuệ Nhân tạo, bạn phải hướng dẫn anh ấy vượt qua mê cung đầy thử thách, tìm ra con đường và đặt các tảng đá lên các công tắc để mở khóa cánh cổng kho báu.

4.2. Cấu trúc mê cung:

Mê cung được biểu diễn dưới dạng lưới ô, bao gồm các loại ô khác nhau như không gian trống, tường, đá, và công tắc. Các ô này tạo thành cấu trúc của mê cung và xác định phạm vi di chuyển cũng như vị trí các vật cản.

4.3. Quy tắc di chuyển

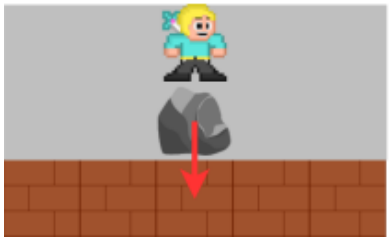
Ares có thể di chuyển theo 4 hướng cơ bản: Lên, Xuống, Trái, và Phải. Như thể hiện trong Hình 2, Ares không thể đi xuyên qua tường hoặc các tảng đá.



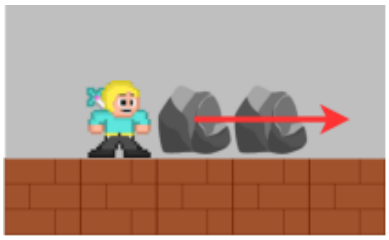
(a) Di chuyển hợp lệ

Nếu ô liền kề chứa một tảng đá và ô phía sau tảng đá là ô trống, Ares có thể đẩy tảng đá vào ô trống đó. Tuy nhiên, Ares không thể kéo các tảng đá, và các tảng đá cũng không thể được đẩy vào tường hoặc các tảng đá khác.

(a) Di chuyển không hợp lệ



(b) Di chuyển không hợp lệ



(c) Di chuyển không hợp lệ

Hình 2: Ares đẩy các tảng đá

4.4. Mục tiêu

Mục tiêu của bài toán là đẩy tất cả các tảng đá trong mê cung lên các ô công tắc đã chỉ định. Khi tất cả các tảng đá nằm đúng vị trí trên các công tắc, nhiệm vụ được xem là hoàn thành. Tất nhiên, số tảng đá trong mê cung luôn bằng với số công tắc.

Mỗi tảng đá có trọng lượng khác nhau, nghĩa là Ares phải xem xét cách hiệu quả nhất để đẩy các tảng đá lên các công tắc. Đẩy những tảng đá nặng hơn tốn nhiều sức lực hơn và giới hạn khả năng di chuyển của anh ấy. Điều này làm nảy sinh nhu cầu về một chiến lược tối ưu khi quyết định đẩy tảng đá nào và vào lúc nào.

5. Yêu cầu

5.1. Dữ liệu đầu vào

Thiết kế ít nhất 10 tệp đầu vào, mỗi tệp đại diện cho một cấu trúc mê cung khác nhau. Mỗi cấu trúc sẽ bao gồm các vị trí của ô trống, tường, đá và công tắc, nhằm tăng tính đa dạng và độ khó của các bài toán cần giải quyết.

5.2. Dữ liệu đầu ra

Lưu kết quả của mỗi lần chạy vào tệp văn bản, trong đó bao gồm các thông tin sau:
Dòng đầu in ra:
thuật toán sử dụng (BFS, DFS, UCS, hoặc A*)
Dòng thứ hai in ra:

- Số bước đi
- Tổng trọng lượng đẩy
- Số nút được sinh ra
- Thời gian chạy thuật toán
- Bộ nhớ sử dụng

Dòng thứ ba in ra:
giải pháp dưới dạng chuỗi biểu thị chuỗi hành động của Ares. Các chữ cái thường uldr được sử dụng cho di chuyển, trong khi các chữ cái hoa ULDR dùng để biểu thị hành động đẩy.

Ví dụ:
BFS
Steps: 16, Weight: 695, Node: 4321, Time (ms): 58.12, Memory (MB): 12.56
uLulDrrRRRRRRurD
DFS ...

5.3. Trình bày trực quan

Hiển thị trực quan quá trình giải quyết mê cung thông qua giao diện GUI, cho phép người dùng quan sát các bước đi và di chuyển của Ares khi thực hiện nhiệm vụ đẩy đá lên các công tắc.

5.4. Ngôn ngữ lập trình

Sử dụng Python để lập trình và triển khai các thuật toán tìm kiếm. Lưu ý rằng các thuật toán BFS, DFS, UCS, và A* sẽ được tự triển khai mà không dùng thư viện hỗ trợ có sẵn cho thuật toán tìm kiếm, nhằm mục đích nắm vững nguyên lý và chi tiết của các thuật toán.

6. Thuật toán và triển khai

6.1. Ý tưởng

6.1.1. Chi tiết

Để có thể áp dụng các thuật toán tìm kiếm vào việc giải quyết bài toán này, nhóm em đã mô hình hóa bài toán thành một đồ thị có hướng như sau:

- Đỉnh: Là một trạng thái của mê cung. Mỗi đỉnh chỉ cần lưu lại vị trí của Ares và vị trí của các tảng đá, bởi vì tường và không gian không thay đổi.
- Cạnh: Là một bước di chuyển hợp lệ của Ares.

- Trọng số cạnh: Là trọng lượng của tảng đá mà Ares phải đẩy khi chuyển từ trạng thái này sang trạng thái kế tiếp. Trọng số cạnh bằng 0 nếu Ares không đẩy tảng đá nào.

Như vậy, bài toán đã trở thành tìm kiếm đường đi từ một đỉnh bắt đầu nào đó đến một đỉnh đích trên đồ thị, với:

- Đỉnh bắt đầu: Là trạng thái ban đầu của mê cung.
- Đỉnh đích: Là trạng thái mà ở đó mỗi tảng đá đều nằm trên công tắc.

Tất nhiên, việc lưu toàn bộ đồ thị (hay không gian trạng thái) trong bộ nhớ trong quá trình giải là không tốt, bởi vì chúng có thể rất lớn, ngay cả đối với những bài toán đơn giản. Do đó, chúng ta chỉ lưu các trạng thái cần thiết nào đó trong bộ nhớ (bằng một cấu trúc dữ liệu thích hợp) để duyệt trước và mở rộng ra các trạng thái mới khi cần thiết. Các thuật toán tìm kiếm dưới đây áp dụng điều này.

6.1.2. Cài đặt

MazeState:
Để thể hiện một trạng thái của mê cung, nhóm em xây dựng một lớp **MazeState**, gồm các thuộc tính sau:

- grid: một list 2 chiều chứa các kí tự tương ứng với từng ô của mê cung.
- height: chiều dài của mê cung (số dòng)
- width: chiều rộng của mê cung (số cột lớn nhất)
- ares_position: một tuple thể hiện vị trí (x, y) của Ares trên mê cung
- stones_weight: một dictionary chứa vị trí của các tảng đá và cân nặng tương ứng, với key là một tuple (x, y): vị trí của tảng đá và value là một số nguyên thể hiện cân nặng của tảng đá
- switches_position: một set chứa vị trí dưới dạng tuple (x, y) của các công tắc

```
class MazeState:
    def __init__(self,
                 grid,
                 height,
                 width,
                 ares_position,
                 stones_weight,
                 switches_position):
        self.grid = grid                          # a 2D list represents the maze
        self.height = height                      # the height of maze
        self.width = width                        # the maximum width of maze
        self.ares_position = ares_position        # ares's position in maze, as a tuple (i, j)
        self.stones_weight = stones_weight        # a dictionary storing the stones' positions and weights: {(i, j): w}
        self.switches_position = switches_position # A set of positions of the switches
```

Hình 3: class MazeState

Các phương thức chính của lớp **MazeState**:

- is_goal_state(): kiểm tra xem trạng thái hiện tại có phải trạng thái đích hay không.

- can_move_up(), can_move_down(), can_move_left(), can_move_right(): kiểm tra xem từ trạng thái hiện tại có thể chuyển sang trạng thái kế bằng cách di chuyển Ares theo hướng tương ứng hay không.
- move_up(), move_down(), move_left(), move_right(): trả về một MazeState tạo thành bằng cách di chuyển Ares theo hướng tương ứng. Nếu không thể di chuyển sẽ trả về trạng thái hiện tại.
- __eq__(): Kiểm tra hai trạng thái có bằng nhau hay không.
- __hash__(): Giá trị băm của trạng thái, để đưa vào các cấu trúc dữ liệu như set hay dictionary.

MazeStateCompress:
Lớp **MazeState** chứa đầy đủ các thông tin về một trạng thái của mê cung và có thể được sử dụng để thực hiện tìm kiếm hoặc các việc khác nếu cần thiết. Tuy nhiên, nhóm em nhận thấy trong quá trình tìm kiếm, chỉ có vị trí của Ares và vị trí của các tảng đá là thay đổi, còn các thuộc tính khác đều giữ nguyên. Do đó, nhóm em xây dựng một lớp **MazeStateCompress** chỉ chứa các thông tin cần thiết nói trên nhằm giúp cho các thuật toán tìm kiếm nhanh và ít tốn kém bộ nhớ hơn.
Các thuộc tính của lớp **MazeStateCompress**:

- ares_position: một tuple thể hiện vị trí (x, y) của Ares trên mê cung
- stones_weight: một dictionary chứa vị trí của các tảng đá và cân nặng tương ứng, với key là một tuple (x, y): vị trí của tảng đá và value là một số nguyên thể hiện cân nặng của tảng đá
- switches_position: một set chứa vị trí dưới dạng tuple (x, y) của các công tắc

Các phương thức chính của lớp **MazeStateCompress**:

- decompress(): trả về một **MazeState** suy ra từ trạng thái **MazeStateCompress** hiện tại.
- can_move_up(), can_move_down(), can_move_left(), can_move_right(), move_up(), move_down(), move_left(), move_right(), __eq__(), __hash__(): giống với lớp **MazeState**

```python
class MazeStateCompress:
    def __init__(self, ares_position, stones_weight, switches_position):
        self.ares_position = ares_position
        self.stones_weight = stones_weight
        self.switches_position = switches_position
```

Hình 4: class MazeStateCompress

MazeSolver:
MazeSolver là một lớp trừu tượng (abstract class), được thiết kế để giải mê cung và chứa các thông tin về lời giải sau khi giải xong. Với mỗi thuật toán tìm kiếm, nhóm em sẽ tạo một lớp kế thừa lớp **MazeSolver** này và cài đặt phương thức giải cho nó.
Các thuộc tính của lớp **MazeSolver**:

- initial_state: một đối tượng MazeState thể hiện trạng thái ban đầu của mê cung cần giải.
- compress_initial_state: một đối tượng MazeStateCompress tương ứng.
- time_consume: thời gian của quá trình giải mê cung.
- memory_consume: bộ nhớ tiêu thụ trong quá trình giải mê cung.
- path: một list các MazeState thể hiện đường đi tìm thấy sau khi giải từ trạng thái ban đầu đến trạng thái đích.
- cost: tổng trọng lượng mà Ares phải đẩy trên đường đi tìm thấy.
- state_visited: số trạng thái thăm trong quá trình tìm kiếm lời giải.
- str_path: thể hiện dưới dạng chuỗi của đường đi tìm thấy.

```python
# Abstract class
class MazeSolver(ABC):
    def __init__(self, initial_state: ms.MazeState):
        self.initial_state = initial_state
        self.compress_initial_state = ms.MazeStateCompress.from_original_maze_state(initial_state)
        self.time_consume = 0       # time consumed to solve the maze, in milliseconds (ms)
        self.memory_consume = 0     # peak memory consumed to solve the maze, in megabytes (MB)
        self.cost = 0               # the cost, in this case, the total weight pushed along the found path
        self.state_visited = 0      # number of states explored by the algorithm
        self.path = []              # list of states in the found path, in reverse order
        self.str_path = ''          # the string representation of the found path

    @classmethod
    def from_file(cls, file_path):
        maze_state = ms.MazeState.from_file(file_path)
        return cls(maze_state)

    @abstractmethod
    def solve_maze(self):
        pass

    def solve_and_track_memory(self):
        before = memory_usage()[0]
        res = memory_usage(self.solve_maze, interval=0.05, retval=True, max_usage=True, max_iterations=1)
        self.memory_consume = res[0] - before
        return res[1]
```

Hình 5: class MazeSolver

Các phương thức của lớp **MazeSolver**:

- solve_maze(): hàm ảo, nơi cài đặt thuật toán để giải mê cung. Các lớp con kế thừa lớp MazeSolver cần cài đặt thân hàm này.
- solve_and_track_memory(): giải mê cung và ghi lại bộ nhớ tiêu thụ.
- is_deadlock(): kiểm tra một trạng thái có rơi vào deadlock hay không.
- step_cost(): trả về trọng lượng mà Ares phải đẩy khi di chuyển từ trạng thái nào đó sang trạng thái kế.
- str_step(): trả về một kí tự đại diện cho hướng đi của Ares khi di chuyển từ trạng thái này sang trạng thái khác.

6.2. Breadth-First Search (BFS)

6.2.1. Chi tiết

- Thuật toán tìm kiếm theo chiều rộng (BFS) luôn chọn đỉnh có khoảng cách gần hơn với đỉnh bắt đầu để mở rộng trước. Để làm được điều đó, thứ tự thăm đỉnh phải đúng với thứ tự chèn vào bộ nhớ, do đó cấu trúc dữ liệu phù hợp là hàng đợi (queue), với đặc tính FIFO (First In First Out).
- BFS đảm bảo sẽ luôn tìm thấy một đường đi từ đỉnh bắt đầu đến đỉnh đích nếu tồn tại đường đi đó bởi vì thuật toán này duyệt tất cả các đỉnh, và đường đi này cũng là đường đi qua ít đỉnh nhất có thể.
- BFS không tính đến trọng số của cạnh, do đó đường đi tìm ra có thể không đảm bảo tính tối ưu (không ngắn nhất). BFS chỉ đảm bảo đường đi tối ưu trong trường hợp tất cả các cạnh có trọng số bằng nhau.
- Đối bài toán này, BFS sẽ tìm ra một đường đi giúp Ares di chuyển ít bước nhất, tuy nhiên không đảm bảo tổng trọng lượng mà anh ta phải đẩy là thấp nhất.
- Độ phức tạp:
  - Về thời gian: $O(b^s)$, với b là branching factor và s là độ sâu của đỉnh đích của cây tìm kiếm.
  - Về không gian: $O(b^s)$, với s là độ sâu của đỉnh đích của cây tìm kiếm.
  - Trong bài toán này $b = 4$, bởi vì từ một trạng thái có thể tạo ra tối đa bốn trạng thái mới bằng cách di chuyển Ares hợp lệ theo 4 hướng như ở phần mô tả.

6.2.2. Cài đặt

```
def solve_maze(self):
    start_time = time.time()
    self.cost = 0
    self.path = []
    self.str_path = ''
    self.state_visited = 1

    if self.initial_state.is_goal_state():
        self.path = [self.initial_state]
        self.time_consume = (time.time() - start_time) * 1000
        return True

    trace = {self.compress_initial_state: None}
    visited = {self.compress_initial_state}
    q = deque()
    q.append(self.compress_initial_state)

    def trace_back(state: ms.MazeStateCompress):
        while state in trace:
            self.path.append(state.decompress(self.initial_state))
            if trace[state] != None:
                self.str_path = MazeSolver.str_step(trace[state], state) + self.str_path
                self.cost = self.cost + MazeSolver.step_cost(trace[state], state)
            state = trace[state]

    while q:
        state = q.popleft()
        if self.is_deadlock(state):
            continue
        for (check, move) in [(state.can_move_up, state.move_up),
                              (state.can_move_left, state.move_left),
                              (state.can_move_down, state.move_down),
                              (state.can_move_right, state.move_right)]:
            if not check(self.initial_state):
                continue
            new_state = move()
            if new_state not in visited:
                self.state_visited += 1
                trace[new_state] = state
                if new_state.is_goal_state():
                    trace_back(new_state)
                    self.time_consume = (time.time() - start_time) * 1000
                    return True
                visited.add(new_state)
                q.append(new_state)

    self.time_consume = (time.time() - start_time) * 1000
    return False
```

Hình 6: Cài đặt thuật toán BFS

- Trước khi giải mê cung, chúng ta cần đặt lại các thuộc tính về ban đầu.
- Hàm trace_back() dùng để truy vết lại đường đi tìm được bởi thuật toán.

- Nhóm em sử dụng built-in deque của Python như một hàng đợi để cài đặt thuật toán BFS.
- Hàm solve_maze() trả về True nếu mê cung có lời giải, ngược lại trả về False.

6.3. Depth-Fisrt Search (DFS)

6.3.1. Chi tiết

- Thuật toán tìm kiếm theo chiều sâu (DFS) luôn mở rộng đỉnh sâu nhất trong cây tìm kiếm trước. Để làm được điều đó, thứ tự thăm đỉnh phải tuân theo thứ tự vào trước, ra sau, do đó cấu trúc dữ liệu phù hợp là ngăn xếp (stack), với đặc tính LIFO (Last In First Out).
- Đối với bài toán này, mỗi đỉnh chúng ta chỉ thăm một lần, vì vậy DFS đảm bảo sẽ luôn tìm thấy một đường đi từ đỉnh bắt đầu đến đỉnh đích nếu tồn tại đường đi, bởi vì thuật toán này duyệt qua tất cả các đỉnh, và số đỉnh trong không gian tìm kiếm là hữu hạn.
- DFS không đảm bảo tìm ra đường đi tối ưu, bởi vì thuật toán này không xét đến số lượng đỉnh hay trọng số các cạnh khi mở rộng các trạng thái.
- Với bài toán này, DFS có thể tìm ra một đường đi cho Ares từ vị trí ban đầu đến vị trí mục tiêu nếu có, nhưng không đảm bảo đường đi này là ít bước nhất hoặc có tổng trọng lượng thấp mà Ares phải đẩy là thấp nhất.
- Độ phức tạp:
  - Về thời gian: $O(b^s)$, với b là branching factor và s là độ sâu của đỉnh đích của cây tìm kiếm.
  - Về không gian: $O(b^s)$, với s là độ sâu của đỉnh đích của cây tìm kiếm.
  - Trong bài toán này $b = 4$, bởi vì từ một trạng thái có thể tạo ra tối đa bốn trạng thái mới bằng cách di chuyển Ares hợp lệ theo 4 hướng như ở phần mô tả.

6.3.2. Cài đặt

```
def solve_maze(self):
    start_time = time.time()
    self.cost = 0
    self.path = []
    self.str_path = ''
    self.state_visited = 1

    if self.initial_state.is_goal_state():
        self.path = [self.initial_state]
        self.time_consume = (time.time() - start_time) * 1000
        return True

    trace = {self.compress_initial_state: None}
    visited = {self.compress_initial_state}
    q = deque()
    q.append(self.compress_initial_state)

    def trace_back(state: ms.MazeStateCompress):···

    while q:
        state = q.pop()
        if self.is_deadlock(state):
            continue
        for (check, move) in [(state.can_move_up, state.move_up),
                              (state.can_move_left, state.move_left),
                              (state.can_move_down, state.move_down),
                              (state.can_move_right, state.move_right)]:
            if not check(self.initial_state):
                continue
            new_state = move()
            if new_state not in visited:
                self.state_visited += 1
                trace[new_state] = state
                if new_state.is_goal_state():
                    trace_back(new_state)
                    self.time_consume = (time.time() - start_time) * 1000
                    return True
                visited.add(new_state)
                q.append(new_state)

    self.time_consume = (time.time() - start_time) * 1000
    return False
```

Hình 7: Cài đặt thuật toán DFS

- Trước khi giải mê cung, chúng ta cần đặt lại các thuộc tính về ban đầu.

- Hàm trace_back() dùng để truy vết lại đường đi tìm được bởi thuật toán giống với ở thuật toán BFS.
- Nhóm em sử dụng built-in deque của Python như một ngăn xếp để cài đặt thuật toán DFS.
- Hàm solve_maze() trả về True nếu mê cung có lời giải, ngược lại trả về False.

6.4. Uniform Cost Search (UCS)

6.4.1. Chi tiết

- Thuật toán tìm kiếm theo chi phí đồng nhất (Uniform Cost Search - UCS) mở rộng đỉnh có chi phí đường đi từ đỉnh bắt đầu nhỏ nhất trước. Để làm được điều này, UCS sử dụng một hàng đợi ưu tiên (priority queue) làm cấu trúc dữ liệu để lưu trữ các đỉnh, trong đó mỗi đỉnh được ưu tiên dựa trên chi phí đường đi từ đỉnh bắt đầu đến đỉnh đó.
- UCS đảm bảo luôn tìm thấy đường đi tối ưu từ đỉnh bắt đầu đến đỉnh đích nếu tồn tại, với điều kiện chi phí của các cạnh không âm. Đây là đặc điểm nổi bật khiến UCS khác biệt so với BFS.
- Trong bài toán này, nhóm em chọn hàm chi phí tại mỗi đỉnh là một bộ gồm (cost, step) với cost là tổng trọng lượng mà Ares phải đẩy từ đỉnh bắt đầu đến đỉnh hiện tại, và step là số bước mà Ares đi. Chi phí tại một đỉnh là ít hơn nếu cost tại đỉnh đó là ít hơn, nếu cost bằng nhau thì chi phí là ít hơn nếu step là ít hơn.
- Như vậy, trong bài toán này, UCS đảm bảo tìm ra một đường đi với ít bước đi nhất trong số các đường đi với tổng trọng lượng phải đẩy là ít nhất.
- Độ phức tạp:
  - Gọi chi phí đường đi tối ưu là C, chi phí tối thiểu giữa hai nút trong đồ thị không gian trạng thái là ϵ và branching factor là b.
  - Trong bài toán này, branching factor $b = 4$, bởi vì Ares có thể di chuyển theo tối đa 4 hướng hợp lệ từ một trạng thái.
  - Về thời gian: $O(b^{C/\epsilon})$, vì chúng ta phải thăm tất cả các đỉnh ở các độ sâu từ 1 đến $b^{C/\epsilon}$.
  - Về không gian: Ước tính là $O(b^{C/\epsilon})$

6.4.2. Cài đặt

```python
class HeapNode:
    def __init__(self, cost, step, current_state, prev_state):
        self.cost = cost
        self.step = step
        self.current_state = current_state
        self.prev_state = prev_state

    def __lt__(self, other):
        if self.cost == other.cost:
            return self.step < other.step
        return self.cost < other.cost
```

Hình 8: Heap node

```
def solve_maze(self):
    start_time = time.time()
    self.cost = 0
    self.path = []
    self.str_path = ''
    self.state_visited = 0

    trace = {}
    visited = set()
    pq = []

    def trace_back(state: ms.MazeStateCompress):···

    class HeapNode:···

    heapq.heappush(pq, HeapNode(0, 0, self.compress_initial_state, None))
    while pq:
        heap_node = heapq.heappop(pq)
        cost = heap_node.cost
        step = heap_node.step
        state = heap_node.current_state
        prev_state = heap_node.prev_state

        if state in visited:
            continue

        self.state_visited += 1

        if self.is_deadlock(state):
            continue

        trace[state] = prev_state
        visited.add(state)

        if state.is_goal_state():
            trace_back(state)
            self.time_consume = (time.time() - start_time) * 1000
            return True

        for (check, move) in [(state.can_move_up, state.move_up),
                              (state.can_move_left, state.move_left),
                              (state.can_move_down, state.move_down),
                              (state.can_move_right, state.move_right)]:
            if not check(self.initial_state):
                continue
            new_state = move()
            if new_state not in visited:
                (i, j) = new_state.ares_position
                heapq.heappush(pq,
                               HeapNode(cost + MazeSolver.step_cost(state, new_state),
                                        step + 1, new_state, state))

    self.time_consume = (time.time() - start_time) * 1000
    return False
```

Hình 9: Thuật toán UCS

- Trước khi giải mê cung, chúng ta cần đặt lại các thuộc tính về ban đầu.
- Hàm trace_back() dùng để truy vết lại đường đi tìm được bởi thuật toán giống với ở thuật toán BFS.
- Nhóm em sử dụng heapq của Python để hỗ trợ cài đặt hàng đợi ưu tiên. HeapNode là một cấu trúc dữ liệu được định nghĩa để chứa thông tin về đỉnh khi đưa vào hàng đợi ưu tiên.
- Hàm solve_maze() trả về True nếu mê cung có lời giải, ngược lại trả về False.

6.5. A*

6.5.1. Chi tiết

- Thuật toán A* mở rộng đỉnh có tổng chi phí ước lượng từ đỉnh bắt đầu đến đỉnh đích nhỏ nhất trước. Để làm được điều này, A* sử dụng một hàng đợi ưu tiên (priority queue) làm cấu trúc dữ liệu để lưu trữ các đỉnh, trong đó mỗi đỉnh được ưu tiên dựa trên hàm $f(n) = g(n) + h(n)$, với $g(n)$ là tổng chi phí từ đỉnh bắt đầu tới đỉnh đích, và $h(n)$ là hàm *heuristic* dự đoán chi phí từ đỉnh n tới đỉnh đích.
- Ở bài toán này, nhóm em chọn hàm *heuristic* $h(n)$ là một hàm trả về hai giá trị gọi là *hcost* và *hstep*, với:
  - *hcost* là tổng trọng lượng phải đẩy dự đoán trên đường đi từ trạng thái hiện tại tới trạng thái đích, được tính bằng tổng của khoảng cách Manhattan từ mỗi tảng đá tới công tắc gần nhất nhân với trọng lượng tương ứng của tảng đá.
  - *hstep* là số bước đi dự đoán mà Ares phải đi để đến trạng thái đích từ trạng thái hiện tại, được tính bằng khoảng cách Mahattan từ Ares đến tảng đá gần nhất.
- $g(n)$ cũng là một bộ hai giá trị (cost, step), như vậy f(n) cũng là một bộ (cost, step). Định nghĩa chi phí nào là ít hơn giống với UCS.
- Dễ thấy hàm *heuristic* $h(n)$ thiết kế trên có tính **admissible**, bởi vì đường đi của mỗi tảng đá tới công tắc không bao giờ nhỏ hơn khoảng cách Manhattan của nó tới công tắc đó, và do đó tổng chi phí ước lượng theo hàm $h(n)$ cũng sẽ không bao giờ vượt quá tổng chi phí thật sự.
- Do nhóm em cài đặt A* graph search thay vì A* tree search, tính **admissible** nói trên không giúp thuật toán luôn tìm được đường đi tối ưu, tuy nhiên, một đường đi tới đích vẫn sẽ luôn được tìm ra.
- Độ phức tạp:
  - Gọi chi phí đường đi tối ưu là C, chi phí tối thiểu giữa hai nút trong đồ thị không gian trạng thái là ϵ và branching factor là b.
  - Trong bài toán này, branching factor $b = 4$, bởi vì Ares có thể di chuyển theo tối đa 4 hướng hợp lệ từ một trạng thái.
  - Về thời gian: $O(b^{C/\epsilon}$, vì chúng ta phải thăm tất cả các đỉnh ở các độ sâu từ 1 đến $b^{C/\epsilon}$.
  - Về không gian: Ước tính là $O(b^{C/\epsilon}$

6.5.2. Cài đặt

```python
def heuristic_value(self, state):
    # sum of distance from each stone to the closest switch, multiply the weight of the stone
    hcost = 0
    for (i, j) in state.stones_weight:
        min_distance = 2000000000
        for (x, y) in self.initial_state.switches_position:
            distance = abs(i - x) + abs(j - y)
            min_distance = min(min_distance, distance)
        hcost += min_distance * state.stones_weight[(i, j)]

    # distance from Ares to the closest stone
    (x, y) = state.ares_position
    hstep = 2000000000
    for (i, j) in state.stones_weight:
        distance = abs(i - x) + abs(j - y)
        hstep = min(min_distance, distance)

    return hcost, hstep
```

Hình 10: Hàm heuristic

```python
class HeapNode:
    def __init__(self, heuristic_value, cost, step, current_state, prev_state):
        self.heuristic_value = heuristic_value
        self.cost = cost
        self.step = step
        self.current_state = current_state
        self.prev_state = prev_state

    def __lt__(self, other):
        self_cost = self.heuristic_value[0] + self.cost
        other_cost = other.heuristic_value[0] + other.cost
        if self_cost == other_cost: # same hcost
            return self.heuristic_value[1] + self.step < other.heuristic_value[1] + other.step
        return self_cost < other_cost
```

Hình 11: Heap node

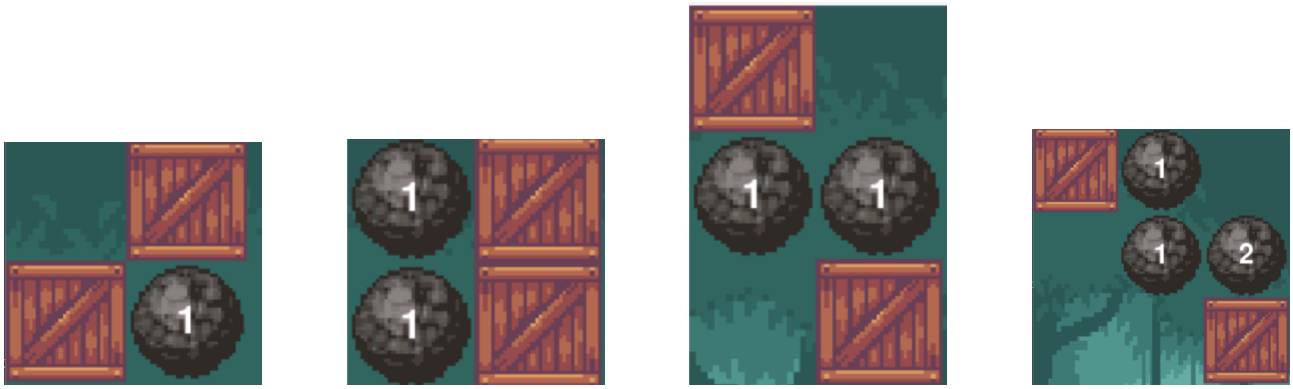
```
def solve_maze(self):
    start_time = time.time()
    self.cost = 0
    self.path = []
    self.str_path = ''
    self.state_visited = 0

    trace = {}
    visited = set()
    pq = []

    def trace_back(state: ms.MazeStateCompress):···

    class HeapNode:···

    heapq.heappush(pq, HeapNode(self.heuristic_value(self.compress_initial_state),
                                0, 0, self.compress_initial_state, None))
    while pq:
        heap_node = heapq.heappop(pq)
        cost = heap_node.cost
        step = heap_node.step
        state = heap_node.current_state
        prev_state = heap_node.prev_state

        if state in visited:
            continue

        self.state_visited += 1

        if self.is_deadlock(state):
            continue

        trace[state] = prev_state
        visited.add(state)

        if state.is_goal_state():
            trace_back(state)
            self.time_consume = (time.time() - start_time) * 1000
            return True

        for (check, move) in [(state.can_move_up, state.move_up),
                              (state.can_move_left, state.move_left),
                              (state.can_move_down, state.move_down),
                              (state.can_move_right, state.move_right)]:
            if not check(self.initial_state):
                continue
            new_state = move()
            if new_state not in visited:
                heapq.heappush(pq,
                               HeapNode(self.heuristic_value(new_state),
                                        cost + MazeSolver.step_cost(state, new_state),
                                        step + 1,
                                        new_state, state))

    self.time_consume = (time.time() - start_time) * 1000
    return False
```

Hình 12: Thuật toán A*

- Trước khi giải mê cung, chúng ta cần đặt lại các thuộc tính về ban đầu.
- Hàm trace_back() dùng để truy vết lại đường đi tìm được bởi thuật toán giống với ở thuật toán BFS.
- Nhóm em sử dụng heapq của Python để hỗ trợ cài đặt hàng đợi ưu tiên. HeapNode là một cấu trúc dữ liệu được định nghĩa để chứa thông tin về đỉnh khi đưa vào hàng đợi ưu tiên.
- Hàm solve_maze() trả về True nếu mê cung có lời giải, ngược lại trả về False.

6.6. Tối ưu

Trong lúc tìm kiếm lời giải, rất có thể sẽ xuất hiện các trạng thái mà tại đó có ít nhất một tảng đá bị kẹt (Ares không thể di chuyển tảng đó) khi chưa nằm ở trên công tắc nào. Việc tiếp tục tìm kiếm từ một trạng thái như vậy là lãng phí, bởi vì chắc chắn một đường đi tới đích sẽ không qua trạng thái đó.
Chính vì vậy, nhóm em đã tối ưu các thuật toán trên để có thể nhận diện hầu hết các trạng thái như trên và ngừng tìm kiếm ở nhánh đó. Với mỗi trạng thái đang được xét trong quá trình tìm kiếm, nhóm em thực hiện kiểm tra trạng thái đó với các deadlock patterns đã thiết kế từ trước:



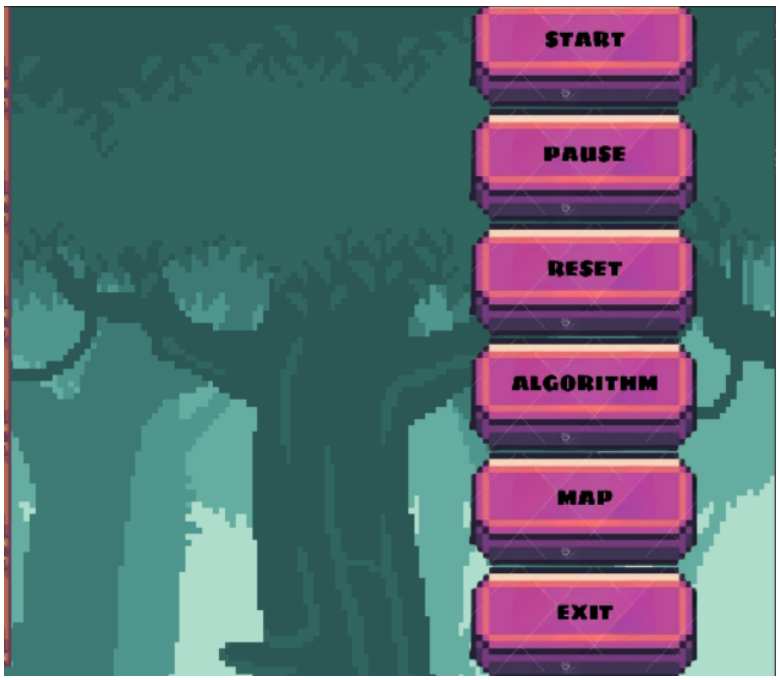
Hình 13: Một số deadlock patterns

7. Chi tiết về Giao diện Người Dùng (UI) của MazeVisualizer

MazeVisualizer là một chương trình trực quan hóa việc giải đố mê cung, sử dụng thư viện Pygame. Dưới đây là các thành phần và chức năng chính trong giao diện của ứng dụng.

7.1. Nút Bấm (Buttons)

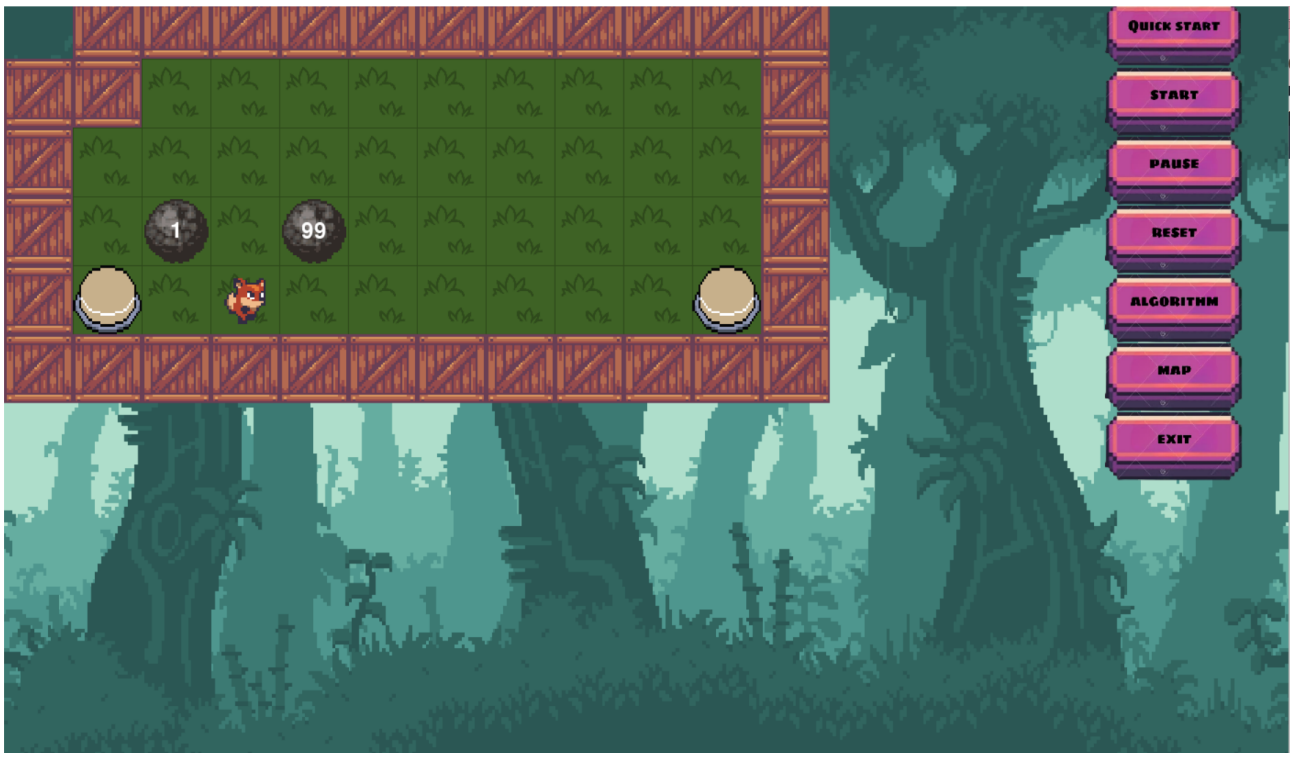
- **Nút Bắt Đầu (Start)**: Khởi động quá trình giải đố mê cung. Khi nhấn nút này, chương trình sẽ bắt đầu giải mê cung bằng 4 thuật toán BFS, DFS, UCS và A*, sau đó tiến hành ghi kết quả ra file output tương ứng, rồi bắt đầu thể hiện đường đi đúng tìm được lên màn hình.
- **Nút (QuickStart)**: là lựa chọn để giải mê cung mà không ghi ra file, sử dụng thuật toán a* , để có thể đưa ra lời giải nhanh hơn trong trường hợp người dùng không có nhu cầu lấy kết quả ở file output.
- **Nút Tạm Dừng (Pause)**: Cho phép tạm dừng quá trình giải. Người dùng có thể nhấn lại để tiếp tục.
- **Nút Đặt Lại (Reset)**: Khôi phục lại trạng thái ban đầu của mê cung, cho phép người dùng bắt đầu lại từ đầu.
- **Nút Thoát (Exit)**: Đóng ứng dụng và thoát khỏi chương trình.
- **Nút Chọn Thuật Toán**: Người dùng có thể chọn giữa các thuật toán khác nhau như BFS, DFS, UCS, và A* để giải mê cung. Mỗi thuật toán có một nút riêng với hình ảnh đại diện.
- **Nút Chọn Bản Đồ**: Có các nút cho phép người dùng chọn từ bản đồ 1 đến 10. Mỗi nút sẽ tải một bản đồ khác nhau từ tệp dữ liệu.



Hình 14: Các nút chức năng

7.2. Màn Hình Vẽ Mê Cung

- **Khu Vực Chính**: Đây là khu vực lớn nhất trong giao diện, nơi hiển thị mê cung và các đối tượng bên trong nó.
- **Hình Ảnh Các Khối**:
  - **Tường (Wall)**: Được đại diện bởi hình ảnh, thường là các khối màu tối.
  - **Không Gian (Space)**: Khu vực có thể đi lại, thể hiện bằng hình ảnh rõ hơn.
  - **Người Chơi (Player)**: Hình ảnh đại diện cho người chơi trong mê cung.
  - **Công Tắc (Switches) và Đá (Stones)**: Các đối tượng tương tác khác, có thể có trọng lượng và ảnh hưởng đến quá trình giải.



Hình 15: Màn hình game

7.3. Thông tin màn chơi

- **Bước Hiện Tại**: Hiển thị số bước đã thực hiện trong quá trình giải.
- **Trọng Lượng Đã Đẩy**: Thông tin về trọng lượng mà người chơi đã đẩy trong quá trình giải đố.
- **Trạng thái giải**: Hiển thị trạng thái map đã được giải hay chưa.

7.4. Chức Năng Tương Tác

- **Sự Kiện Chuột**: Người dùng có thể nhấp chuột để tương tác với các nút bấm. Nếu nhấn vào nút nào, ứng dụng sẽ thực hiện hành động tương ứng.
- **Tạm Dừng và Tiếp Tục**: Trong quá trình giải, người dùng có thể tạm dừng để xem xét tình hình, sau đó tiếp tục khi sẵn sàng.
- **Thông Báo Lỗi**: Nếu mê cung không thể giải được, ứng dụng sẽ hiển thị thông báo lỗi cho người dùng.

7.5. Thiết Kế và Giao Diện

- **Bố Cục Rõ Ràng**: Các nút bấm được thiết kế để dễ dàng nhìn thấy và truy cập. Phía bên phải có các nút chức năng, trong khi khu vực chính dành cho mê cung.
- **Hình Ảnh và Màu Sắc**: Sử dụng hình ảnh sắc nét và màu sắc tương phản để người dùng dễ dàng phân biệt các thành phần khác nhau trong mê cung.
- **Đáp Ứng**: Giao diện thiết kế để hoạt động mượt mà trên các màn hình có kích thước khác nhau, từ máy tính để bàn đến laptop.

Hình 16: Thông tin màn chơi

8. Kết quả chạy thử và nhận xét

8.1. Bảng thống kê

	BFS				
	Step	Weight	Node	Time (ms)	Memory (MB)
input-01.txt	16	695	10019	306.13	2.46
input-02.txt	97	880	1778	50.91	0.00
input-03.txt	110	227	9383	349.29	2.77
input-04.txt	92	96	234133	12670.16	78.59
input-05.txt	113	1601	3745	530.69	1.69
input-06.txt	248	770	230057	44067.93	102.96
input-07.txt	164	854	71212	3004.62	4.51
input-08.txt	39	619	2572879	144184.27	1194.59
input-09.txt	156	5940	378636	20954.74	116.76
input-10.txt	44	47	40460	1966.63	9.01

Hình 17: Kết quả thuật toán BFS

	DFS				
	Step	Weight	Node	Time (ms)	Memory (MB)
input-01.txt	49	695	132	4.52	0.02
input-02.txt	167	1130	1013	29.13	0.00
input-03.txt	241	369	4554	200.69	0.45
input-04.txt	318	309	581754	38564.60	218.94
input-05.txt	177	2005	2734	416.38	1.41
input-06.txt	3890	10780	453497	93237.93	208.53
input-07.txt	1322	4340	27608	1131.35	3.75
input-08.txt	1703	8337	2799873	175084.69	1376.90
input-09.txt	6470	186508	296723	16750.06	23.22
input-10.txt	101	93	18111	933.01	1.71

Hình 18: Kết quả thuật toán DFS

	UCS				
	Step	Weight	Node	Time (ms)	Memory (MB)
input-01.txt	24	405	31314	1300.47	13.47
input-02.txt	97	880	1864	54.11	0.00
input-03.txt	110	227	7611	338.58	0.45
input-04.txt	92	96	229037	15807.81	18.21
input-05.txt	117	1501	3779	665.85	1.62
input-06.txt	248	770	191056	45503.54	87.56
input-07.txt	208	722	159059	9145.23	29.81
input-08.txt	45	599	4134564	356888.80	1754.12
input-09.txt	156	5940	378731	23164.06	44.09
input-10.txt	44	47	110337	6813.22	38.09

Hình 19: Kết quả thuật toán UCS

	A*				
	Step	Weight	Node	Time (ms)	Memory (MB)
input-01.txt	24	405	4872	242.99	2.97
input-02.txt	97	880	1443	57.13	0.00
input-03.txt	110	227	4267	217.05	0.80
input-04.txt	92	96	109051	7663.74	51.64
input-05.txt	117	1501	3308	662.85	1.48
input-06.txt	248	770	151321	43367.62	61.39
input-07.txt	208	722	100295	6449.74	9.10
input-08.txt	45	599	185829	14663.92	55.25
input-09.txt	156	5940	373487	28025.99	167.47
input-10.txt	44	47	4549	382.56	0.01

Hình 20: Kết quả thuật toán A*

8.2. So sánh các thuật toán

- Thuật toán BFS luôn tìm ra đường đi có ít step nhất.
- Thuật toán DFS thường tìm ra đường đi có nhiều step nhất do tính chất tìm kiếm theo chiều sâu của nó.
- Thuật toán UCS thường là chậm nhất bởi vì nó xét nhiều trạng thái để tìm ra đường đi tối ưu nhất.
- Thuật toán A* tìm ra kết quả nhanh nhất vả điều đó càng được thể hiện rõ hơn đối với những mê cung phức tạp.
- Đối với những mê cung phức tạp, các thuật toán BFS, DFS, UCS chạy trong thời gian rất lớn, chỉ có thuật toán A* chạy trong thời gian ngắn chấp nhận được. Đó cũng là lí do để nhóm em thiết kế thêm chức năng **QUICK START** cho chương trình - chức năng chỉ sử dụng thuật toán A* để giải mê cung.

8.3. Kết luận

- Các thuật toán BFS, DFS, UCS chỉ phù hợp để giải các mê cung đơn giản.
- Đối với các trò chơi mê cung trong thực tế, nơi mà các mê cung thường phức tạp và rộng lớn, thuật toán A* nên là thuật toán được chọn để sử dụng.

9. Demo

Link video demo: `https://youtu.be/-NDkl0KQVKo?si=rAU6vBST6vt6kJal`