ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

---

# Ares's adventure

**Đề tài: Project 1. Search**

---

**Môn học: CSC14003 - Cơ sở trí tuệ nhân tạo**

*Sinh viên thực hiện:*

Thái Văn Mạnh (22120206)

Song Đồng Gia Phúc (22120282)

Nguyễn Quang Thắng

(22120333)

*Giảng viên hướng dẫn:*

Thầy Nguyễn Ngọc Đức

Ngày 7 tháng 11 năm 2024

# Mục lục

**1 Yêu cầu đề bài** **1**

1.1 Tổng quan . . . 1

1.2 Luật chơi . . . 1

1.3 Mục tiêu . . . 2

1.4 Đầu vào . . . 2

1.5 Đầu ra . . . 3

1.6 Giao diện đồ họa . . . 3

**2 Thông tin nhóm** **4**

2.1 Danh sách thành viên . . . 4

2.2 Bảng phân công công việc . . . 4

2.3 Bảng tự đánh giá . . . 5

**3 Hướng dẫn cài đặt và khởi chạy ứng dụng** **5**

3.1 Yêu cầu . . . 5

3.2 Các bước cài đặt và khởi chạy . . . 5

3.3 Video demo . . . 6

**4 Cài đặt các thuật toán** **6**

4.1 DFS và BFS . . . 6

**5 Giao diện** **7**

5.1 Tổng quan . . . 7

5.2 Màn hình chính (MenuScreen) . . . 7

5.2.1 Tổng quan . . . 7

5.2.2 Các tác vụ . . . 8

5.3 Màn hình giải mê cung (MazeScreen) . . . 9

5.3.1 Tổng quan . . . 9

5.3.2 Các tác vụ . . . 10

**6 Tổ chức mã nguồn** **12**

6.1 Thư mục board/entities . . . 14
6.1.1 File entity_interface.py . . . 14
6.1.2 File ares.py . . . 14
6.1.3 File stone.py . . . 14
6.1.4 File __init__.py . . . 14
6.2 Thư mục board . . . 15
6.2.1 File __init__.py . . . 15
6.2.2 File board_interface.py . . . 15
6.2.3 File static_board.py . . . 15
6.2.4 File dynamic_board.py . . . 15
6.2.5 File board_symbol.py . . . 15
6.3 Thư mục extended_input . . . 15
6.4 Thư mục search . . . 15
6.4.1 File __init__.py . . . 15
6.4.2 File algorithm.py . . . 16
6.4.3 File search_frontier.py . . . 16
6.4.4 File search_result.py . . . 16
6.4.5 File search.py . . . 16
6.4.6 File heuristics.py . . . 16
6.5 File controller.py . . . 16
6.6 Thư mục GUI . . . 16
6.6.1 File general_graphic.py . . . 16
6.6.2 File Grid.py . . . 17
6.6.3 File Maze.py . . . 17
6.6.4 File MazeScreen.py . . . 18
6.7 File MenuScreen.py . . . 18
6.7.1 Các file còn lại . . . 19
6.8 File main.py . . . 19

**7 Phân tích kết quả cài đặt** **19**
7.1 Thí nghiệm 1 . . . 19

7.1.1 Sơ đồ mê cung . . . 19
7.1.2 Kết quả đầu ra . . . 20
7.2 Thí nghiệm 2 . . . 21
7.2.1 Sơ đồ mê cung . . . 21
7.2.2 Kết quả đầu ra . . . 21
7.3 Thí nghiệm 3 . . . 22
7.3.1 Sơ đồ mê cung . . . 22
7.3.2 Kết quả đầu ra . . . 23
7.4 Thí nghiệm 4 . . . 24
7.4.1 Sơ đồ mê cung . . . 24
7.4.2 Kết quả đầu ra . . . 24
7.5 Thí nghiệm 5 . . . 25
7.5.1 Sơ đồ mê cung . . . 25
7.5.2 Kết quả đầu ra . . . 26
7.6 Thí nghiệm 6 . . . 27
7.6.1 Sơ đồ mê cung . . . 27
7.6.2 Kết quả đầu ra . . . 27
7.7 Thí nghiệm 7 . . . 28
7.7.1 Sơ đồ mê cung . . . 28
7.7.2 Kết quả đầu ra . . . 29
7.8 Thí nghiệm 8 . . . 30
7.8.1 Sơ đồ mê cung . . . 30
7.8.2 Kết quả đầu ra . . . 30
7.9 Thí nghiệm 9 . . . 31
7.9.1 Sơ đồ mê cung . . . 31
7.9.2 Kết quả đầu ra . . . 32
7.10 Thí nghiệm 10 . . . 33
7.10.1 Sơ đồ mê cung . . . 33
7.10.2 Kết quả đầu ra . . . 33

# Danh sách bảng

1 Danh sách thành viên . . . . . 4
2 Phân công công việc . . . . . 4
3 Bảng đánh giá kết quả . . . . . 5

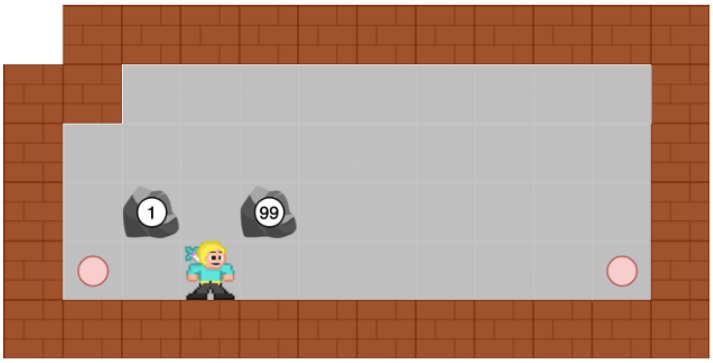
# Danh sách hình vẽ

1 Ví dụ về giao diện đồ họa và dữ liệu đầu vào tương ứng. . . . . 1
2 Hướng đi của Ares. . . . . 1
3 Ví dụ về cách đẩy đá. (a) Hợp lệ, (b) và (c) Không hợp lệ. . . . . 2
4 Màn hình chính khi bắt đầu . . . . . 8
5 Màn hình chính với mê cung và thuật toán khác cùng với con trỏ chuột ở nút bắt đầu 9
6 Màn hình tự phóng to/thu nhỏ theo cửa sổ đang chạy . . . . . 10
7 Màn hình giải mê cung . . . . . 11
8 Màn hình chờ trong quá trình tạo lại output . . . . . 12

# 1 Yêu cầu đề bài

## 1.1 Tổng quan

Sinh viên cần cài đặt các thuật toán tìm kiếm để giúp nhân vật chính Ares có thể đẩy những cục đá về vị trí các công tắc. Đầu vào của chương trình là một tệp tin chứa bản đồ mê cung, như hình 1.

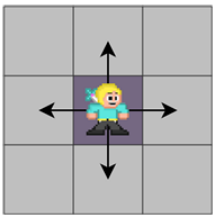



```
1 99
 ##########
##          #
#           #
# $ $       #
#.  @      .#
##########
```

Hình 1: Ví dụ về giao diện đồ họa và dữ liệu đầu vào tương ứng.

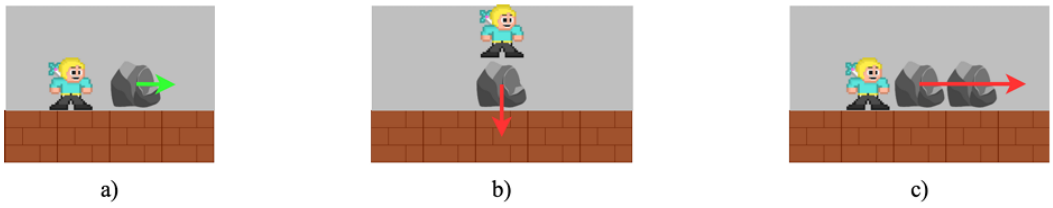
## 1.2 Luật chơi

Mê cung được mô tả dưới dạng lưới với kích thước $n \times m$, với mỗi ô trong lưới có thể là ô trống (free space), tường (wall), cục đá (stone) hoặc công tắc (switch). Nhân vật chính Ares có thể di chuyển 1 ô vuông một lượt, theo 4 hướng: **trên**, **dưới**, **trái**, **phải**, như hình 2. Ares không thể đi xuyên tường hay xuyên đá.



Hình 2: Hướng đi của Ares.

Ares có thể đẩy cục đá nếu đang đứng cạnh cục đá, và ô phía bên kia của cục đá là ô trống. Ares không thể kéo cục đá đi lùi, và cũng không thể đẩy đá xuyên tường hoặc đẩy hai cục đá cùng lúc. Hình 3 minh họa chi tiết cách đẩy.




Hình 3: Ví dụ về cách đẩy đá. (a) Hợp lệ, (b) và (c) Không hợp lệ.

Mỗi cục đá sẽ có khối lượng. Công để Ares đẩy cục đá sẽ tỉ lệ thuận với khối lượng cục đá được đẩy. Công để Ares di chuyển nếu không có đá là 0. Các cục đá có thể về với bất kỳ công tắc nào.

## 1.3 Mục tiêu

Sinh viên cần cài một số thuật toán tìm kiếm để giúp Ares tìm được cách đẩy tất cả cục đá về vị trí của công tắc. Các thuật toán cần cài đặt bao gồm:

- Tìm kiếm theo chiều rộng (Breadth-First Search, BFS).
- Tìm kiếm theo chiều sâu (Depth-First Search, DFS).
- Tìm kiếm chi phí đều (Uniform-Cost Search, UCS).
- Thuật toán A* có Heuristic.

## 1.4 Đầu vào

Định dạng tệp đầu vào như sau (như hình 1):

- Dòng đầu tiên chứa danh sách những số nguyên, biểu thị cho khối lượng mỗi cục đá có trong mê cung. Thứ tự các cục đá được xếp theo thứ tự chúng xuất hiện trong mê cung, từ trái sang phải và từ trên xuống dưới.
- Những dòng tiếp theo miêu tả từng ô trong mê cung. Mỗi ô được biểu diễn bởi một trong các ký tự dưới đây:
  - "#" cho tường.
  - " " (khoảng trắng) cho ô trống.

- "$" cho cục đá.
- "@" cho Ares.
- "." cho công tắc.
- "*" cho cục đá được đặt trên một công tắc.
- "+" cho Ares đang đứng trên công tắc.

## 1.5 Đầu ra

Tệp đầu ra có cấu trúc như sau:

- Dòng đầu tiên chứa tên thuật toán được dùng.
- Dòng thứ hai chứa những thông tin:
  - Số bước Ares cần thực hiện.
  - Tổng khối lượng Ares phải đẩy.
  - Tổng số Node sinh bởi thuật toán.
  - Thời gian để tìm kiếm đường đi.
  - Bộ nhớ cần dùng cho quá trình tìm kiếm.
- Dòng thứ ba chứa lộ trình Ares đi để đẩy đá. Ký tự thường (uldr) biểu thị bước đi không đẩy đá, ký tự in hoa (ULDR) biểu thị đẩy đá.

## 1.6 Giao diện đồ họa

Giao diện cần đảm bảo các tiêu chí tối thiểu dưới đây:

- Thể hiện được tất cả các thành phần của mê cung.
- Thể hiện từng bước quá trình Ares đưa các viên đá về đúng vị trí.
- Thể hiện các thông số ở từng bước đi, tối thiểu bao gồm số bước đã đi và khối lượng đã đẩy.
- Có một số nút điều khiển: bắt đầu hoặc dừng việc di chuyển; chạy lại từ đầu.

# 2 Thông tin nhóm

## 2.1 Danh sách thành viên

| STT | MSSV | Họ và tên | Phần trăm đóng góp |
|---|---|---|---|
| 1 | 22120206 | Thái Văn Mạnh | 33,33% |
| 2 | 22120282 | Song Đồng Gia Phúc | 33,33% |
| 3 | 22120333 | Nguyễn Quang Thắng | 33,34% |

Bảng 1: Danh sách thành viên

## 2.2 Bảng phân công công việc

| STT | Tên công việc | Thành viên thực hiện | Đánh giá |
|---|---|---|---|
| 1 | Cài đặt thuật toán DFS | Thái Văn Mạnh | Hoàn thành |
| 2 | Cài đặt thuật toán BFS | Thái Văn Mạnh | Hoàn thành |
| 3 | Cài đặt thuật toán UCS | Nguyễn Quang Thắng | Hoàn thành |
| 4 | Cài đặt thuật toán A* | Song Đồng Gia Phúc | Hoàn thành |
| 5 | Sinh test case 1-5 | Nguyễn Quang Thắng | Hoàn thành |
| 6 | Sinh test case 6-10 | Song Đồng Gia Phúc | Hoàn thành |
| 7 | Kết quả (tập tin đầu ra) | Nguyễn Quang Thắng | Hoàn thành |
| 8 | Code giao diện (GUI) | Thái Văn Mạnh | Hoàn thành |
| 9 | Quay video demo | Song Đồng Gia Phúc | Hoàn thành |
| 10 | Edit video demo | Song Đồng Gia Phúc | Hoàn thành |
| 11 | Viết báo cáo thuật DFS, BFS | Thái Văn Mạnh | Hoàn thành |
| 12 | Viết báo cáo thuật UCS, A* | Nguyễn Quang Thắng | Hoàn thành |
| 13 | Viết báo cáo kết quả thử nghiệm từng test case | Nguyễn Quang Thắng | Hoàn thành |
| 14 | Nhận xét kết quả thử nghiệm từng test case | Song Đồng Gia Phúc | Hoàn thành |
| 15 | Viết hướng dẫn chạy chương trình | Thái Văn Mạnh | Hoàn thành |

Bảng 2: Phân công công việc

### 2.3 Bảng tự đánh giá

| STT | Nội dung | Mức độ hoàn thành |
|---|---|---|
| 1 | Cài đặt BFS đúng | 100% |
| 2 | Cài đặt DFS đúng | 100% |
| 3 | Cài đặt UCS đúng | 100% |
| 4 | Cài đặt A* đúng | 100% |
| 5 | Tự sinh ra 10 bộ dữ liệu thử với các thuộc tính khác nhau | 100% |
| 6 | Kết quả (tệp đầu ra và giao diện đồ họa) | 100% |
| 7 | Video minh họa một vài thuật toán | 100% |
| 8 | Báo cáo | 100% |

Bảng 3: Bảng đánh giá kết quả

## 3 Hướng dẫn cài đặt và khởi chạy ứng dụng

### 3.1 Yêu cầu

- Python 3.10.15 (cài đặt thông qua `Conda`)

### 3.2 Các bước cài đặt và khởi chạy

- Mở Command Prompt.
- Chuyển đến thư mục đồ án `22120206\_22120282\_22120333`.
- Chuyển đến thư mục `Source`:

  ```
  >> cd Source
  ```

- Tạo môi trường:

  ```
  >> conda create -n Lab01 python=3.10.15
  ```

- Kích hoạt môi trường:

  ```
  >> conda activate Lab01
  ```

- Cài đặt thư viện cần thiết:

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

- Khởi chạy ứng dụng:

  ```
  >> python main.py
  ```

### 3.3 Video demo

https://youtu.be/ciYdCXeMmRE

## 4 Cài đặt các thuật toán

### 4.1 DFS và BFS

Bảng 1 biểu thị mã giả của hai thuật toán DFS và BFS. Về cách cài đặt, hai thuật toán chỉ khác nhau cấu trúc dữ liệu dùng để lưu các bước đi, cụ thể DFS dùng ngăn xếp còn BFS dùng hàng đợi.

**Algorithm 1** Thuật toán tìm kiếm

```
function SEARCH(algorithm, maze)
    frontier ← getSearchFrontier(algorithm)
    entityPositions ← getAresAndStonesPositions(maze)
    frontier.append(entityPositions)
    while True do
        latestState := frontier.pop()
        if all stones is in switch then
            break
        end if
        for cell in Ares's adjacent cell do
            if is not valid move then
                continue
            end if
            newState ← state after Ares move to cell from latestState
            if newState is not visited then
                frontier.append(newState)
            end if
        end for
    end while
    trade back the path
    return result
end function
```

Đối với hai thuật toán UCS và A*, hàng đợi ưu tiên sẽ được sử dụng để lưu dữ liệu. Được đưa vào hàng đợi ưu tiên, bên cạnh trạng thái, một trọng số `weight` sẽ được gán với trạng thái và đưa vào cùng. Công thức tính trọng số như sau:

$$weight = g(state) + h(state)$$

trong đó $g(state)$ là khối lượng đá đã đẩy tính từ thời điểm khởi tạo đến khi có trạng thái đó; $h(state)$ là hàm Heuristic. Trong trường hợp thuật toán UCS, $h(state) = 0, \forall state$.

Đối với thuật toán A*, hàm $h$ nhận vào trạng thái hiện tại $state$ và tính toán ước lượng tổng trọng lượng phải đẩy tiếp theo bằng giải thuật chia cặp ghép **Hungarian**. Giả sử số lượng hòn đá là $cnt$ thì thuật toán này chạy trong độ phức tạp $O(cnt^3)$

Vì tính chất tương tự nhau của các thuật toán, chỉ có một hàm tìm kiếm duy nhất được cài đặt, và sử dụng

# 5 Giao diện

## 5.1 Tổng quan

Đồ họa của ứng dụng được lấy cảm hứng từ trò chơi *Shove It! ...The Warehouse Game* (trong tiếng Nhật gọi là *Shijo Saidai no Sokoban*). Các thành phần của mê cung được tải từ trang web *The Spriters Resource*.

## 5.2 Màn hình chính (MenuScreen)

### 5.2.1 Tổng quan

Đây là màn hình chính của ứng dụng (Hình 4), được quản lý bởi file `GUI/MenuScreen.py`. Màn hình chính của ứng dụng bao gồm:

- Tiêu đề của trò chơi: **Ares's adventure**
- 10 nút chọn mê cung ứng với 10 file input tương ứng: `input-01.txt`, `input-02.txt`, ..., `input-10.txt`. 10 mê cung đều được thể hiện trạng thái xem trước trong nút chọn. Mê cung đang được chọn sẽ có hình chữ nhật màu vàng bao quanh. Mê cung mặc định được chọn là `input-01.txt`.
- 4 nút chọn thuật toán ứng với 4 thuật toán tìm kiếm. Thuật toán đang được chọn sẽ có màu vàng. Thuật toán mặc định được chọn là `BFS`.
- 1 nút **Start** để bắt đầu giải mê cung.


Hình 4: Màn hình chính khi bắt đầu

### 5.2.2 Các tác vụ

Các tác vụ liên quan đến quản lý vị trí con trỏ chuột (hover) và nhấn chuột trái (click):

- Khi người dùng di chuyển chuột đến các vị trí có thể click được, gồm 10 nút chọn map, 4 nút chọn thuật toán tìm kiếm và nút bắt đầu, con trỏ chuột sẽ được đặt thành bàn tay biểu thị vùng này có thể click được. Ở những vùng khác, con trỏ chuột vẫn sẽ là mũi tên (mặc định).
- Khi người dùng di chuyển chuột đến vị trí các nút chọn mê cung, mê cung đó sẽ có hình chữ nhật bao quanh. Nếu người dùng click vào nút đó, mê cung đó sẽ được chọn.
- Khi người dùng di chuyển chuột đến vị trí các nút chọn thuật toán, nút đó sẽ có màu vàng. Nếu người dùng click vào nút đó, thuật toán đó sẽ được chọn.
- Khi người dùng di chuyển chuột đến vị trí nút bắt đầu, nút sẽ có màu xanh ngọc. Nếu người dùng click vào nút đó, ứng dụng sẽ chuyển đến màn hình giải mê cung với mê cung và thuật toán đã chọn cùng với file output đã có.

Đặc biệt, mọi màn hình trong ứng dụng đều có khả năng phóng to/thu nhỏ theo cửa sổ đang chạy (Hình 6).

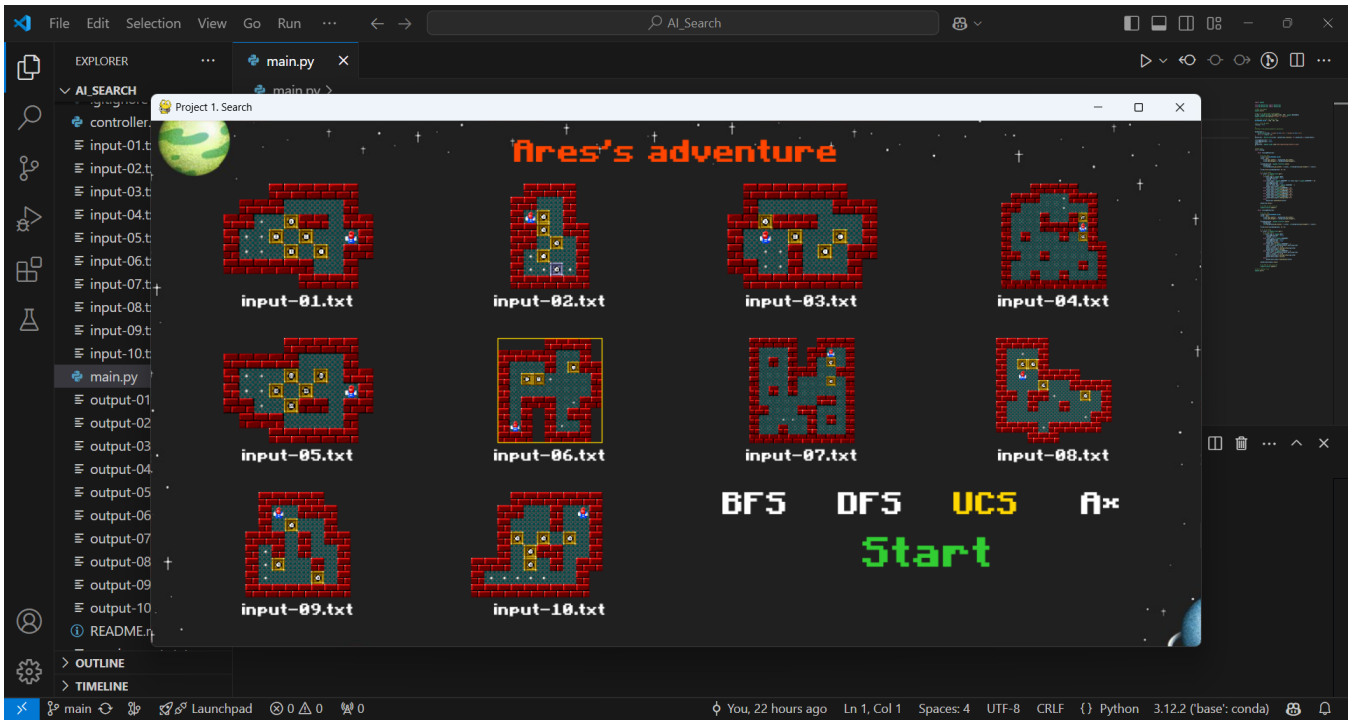

Hình 5: Màn hình chính với mê cung và thuật toán khác cùng với con trỏ chuột ở nút bắt đầu

## 5.3 Màn hình giải mê cung (MazeScreen)

### 5.3.1 Tổng quan

Đây là màn hình giải mê cung ứng với mê cung đã được chọn (Hình 7), được quản lý bởi file GUI/MazeScreen.py. Màn hình giải mê cung của ứng dụng bao gồm:

- Input của mê cung hiện tại. Ví dụ: **input-06.txt**
- Màn hình mê cung ứng với nội dung trong file input và yêu cầu đồ án.
- Bảng thông số của thuật toán hiện tại theo file output ứng với mê cung hiện tại, bao gồm: bước hiện tại/tổng số bước, tổng số khối lượng đã đẩy,
- 4 nút chọn thuật toán ứng với 4 thuật toán tìm kiếm. Thuật toán đang được chọn sẽ có màu vàng.
- Nút **Regenerate output** dùng để tạo lại output.
- 2 nút dùng để chuyển đến mê cung tiếp theo và trước đó.




Hình 6: Màn hình tự phóng to/thu nhỏ theo cửa sổ đang chạy

- Các nút chức năng **Start**, **Pause/Resume**, **Back**, **Next**, **Restart**, **Menu**, và nút chỉnh tốc độ (**1x**, **2x**, **4x**, **8x**, **16x**).

### 5.3.2 Các tác vụ

Các tác vụ liên quan đến quản lý vị trí con trỏ chuột (hover):

- Khi người dùng di chuyển chuột đến các vị trí có thể click được, gồm 4 nút chọn thuật toán tìm kiếm, nút tạo lại output, 2 nút chuyển mê cung và các nút chức năng (**Start**, **Pause/Resume**, **Back**, **Next**, **Restart**, **Menu**, và nút chỉnh tốc độ (**1x**, **2x**, **4x**, **8x**, **16x**)), con trỏ chuột sẽ được đặt thành bàn tay biểu thị vùng này có thể click được. Ở những vùng khác, con trỏ chuột vẫn sẽ là mũi tên (mặc định).
- Khi người dùng di chuyển chuột đến vị trí các nút chọn thuật toán và các nút chuyển mê cung, các nút đó sẽ có màu vàng.
- Khi người dùng di chuyển chuột đến vị trí nút tạo lại output, nút sẽ có màu đậm hơn.
- Khi người dùng di chuyển chuột đến vị trí các nút chức năng, các nút đó sẽ có màu xanh ngọc.


Hình 7: Màn hình giải mê cung

Các tác vụ liên quan đến quản lý việc nhấn chuột trái (click):

- Khi người dùng click vào nút **Next**, nếu bước hiện tại chưa phải bước cuối, tiến thêm một bước.
- Khi người dùng click vào nút **Back**, nếu hiện tại không phải trạng thái ban đầu, lùi một bước.
- Khi người dùng click vào nút **Start**, mê cung sẽ ở chế độ tự động chạy và sẽ tiến một bước sau một khoảng thời gian nhất định với tốc độ hiện tại.
- Khi người dùng click vào nút **Pause**, quá trình giải mê cung sẽ tạm dừng và nút trở thành **Resume**. Khi người dùng click vào nút **Resume**, quá trình giải mê cung sẽ tiếp tục chạy và nút trở thành **Pause**.
- Khi người dùng click vào nút **Restart**, mê cung sẽ được đặt lại trạng thái ban đầu.
- Khi người dùng click vào nút chỉnh tốc độ, tốc độ sẽ được thay đổi theo 1x, 2x, 4x, 8x, 16x và quay lại 1x.
- Khi người dùng click vào nút **Menu**, ứng dụng sẽ quay lại màn hình chính ban đầu.

- Khi người dùng click vào 2 nút chuyển mê cung, màn hình sẽ trở thành màn hình giải mê cung trước đó/kế tiếp với tên thuật toán là thuật toán hiện tại và đặt lại trạng thái ban đầu.
- Khi người dùng click vào nút chọn thuật toán, thuật toán đó sẽ được chọn và mê cũng sẽ đặt lại trạng thái ban đầu.
- Khi người dùng click vào nút tạo lại output, file output ứng với file input của mê cung hiện tại sẽ được tạo lại bằng cách gọi hàm tìm kiếm, ghi lại thông tin thời gian, bộ nhớ, số node,...

Khi ứng dụng đang trong quá trình tạo lại output, ứng dụng sẽ chuyển sang màn hình chờ (Hình 8). Màn hình chờ sẽ làm mờ màn hình mê cung hiện tại và hiện ra thông báo **Searching...**. Khi quá trình tạo lại output hoàn tất, ứng dụng sẽ tự động trở về màn hình mê cung hiện tại và cập nhật lại các thông số (thời gian tìm kiếm, bộ nhớ, số node,...) ứng với file output vừa được tạo lại.


Hình 8: Màn hình chờ trong quá trình tạo lại output

Như đã đề cập ở trên, mọi màn hình mê cung và màn hình chờ đều có khả năng phóng to/thu nhỏ theo cửa sổ đang chạy.

# 6 Tổ chức mã nguồn

Đây là cấu trúc chi tiết về mã nguồn:

```
22120206_22120282_22120333
├── Source
│   ├── board
│   │   ├── entities
│   │   │   ├── __init__.py
│   │   │   ├── entity_interface.py
│   │   │   ├── ares.py
│   │   │   └── stone.py
│   │   ├── __init__.py
│   │   ├── board_interface.py
│   │   ├── board_symbol.py
│   │   ├── dynamic_board.py
│   │   └── static_board.py
│   ├── GUI
│   │   ├── general_graphic.py
│   │   ├── Grid.py
│   │   ├── Maze.py
│   │   ├── MazeScreen.py
│   │   ├── MenuScreen.py
│   │   └── ...
│   ├── search
│   │   ├── __init__.py
│   │   ├── algorithm.py
│   │   ├── heuristics.py
│   │   ├── search_frontier.py
│   │   └── search_result.py .4 search.py
│   ├── controller.py
│   ├── .gitignore
│   ├── input-01.txt
│   ├── ..
│   ├── input-10.txt
│   ├── output-01.txt
│   ├── ..
│   ├── output-10.txt
│   ├── main.py
│   ├── README.txt
│   ├── requirements.txt
│   └── ...
└── Report.pdf
```

Sau đây là giải thích về từng thư mục và các tệp bên trong:

22120206_22120282_22120333: Thư mục gốc của dự án, được đặt tên theo mã số sinh viên của các thành viên nhóm.

- `Source`: Thư mục chứa toàn bộ mã nguồn và tài nguyên của dự án.
  - `input-01.txt` đến `input-10.txt`: Tập hợp các tệp đầu vào mẫu được sử dụng để kiểm thử chương trình.
  - `output-01.txt` đến `output-10.txt`: Tập hợp các tệp đầu ra tương ứng với các tệp đầu vào.
  - `README.txt`: File hướng dẫn cách để chạy ứng dụng.
  - `requirements.txt`: File liệt kê các thư viện cần thiết để chạy ứng dụng.
- `Report.pdf`: Báo cáo về ứng dụng.

Sau đây là giải thích về các thư mục và file còn lại bên trong thư mục `Source`, là những thư mục cần giải thích sâu về mã nguồn:

## 6.1 Thư mục board/entities

Thư mục này chứa khai báo các thực thể có trong mê cung, bao gồm Ares và các viên đá.

### 6.1.1 File entity_interface.py

Chứa lớp trừu tượng cho các thực thể. Bao gồm các thuộc tính như có ở trên công tắc không `is\_on\_switch`, và `id`.

### 6.1.2 File ares.py

Khai báo lớp `Ares` kế thừa Interface.

### 6.1.3 File stone.py

Khai báo lớp `Stone` kế thừa Interface.

### 6.1.4 File __init__.py

Đóng gói thư mục `entities` thành một module để bên ngoài sử dụng.

## 6.2 Thư mục board

Thư mục này giúp xử lý các mê cung. Được chia thành hai loại bảng: bảng tĩnh lưu các thành phần không đổi (như vị trí tường, công tắc), và bảng động lưu các thành phần thay đổi (như đá và Ares).

### 6.2.1 File __init__.py

Đóng gói thư mục `board` thành một module để bên ngoài sử dụng.

### 6.2.2 File board_interface.py

Chứa lớp trừu tượng cho các loại bảng.

### 6.2.3 File static_board.py

Chứa lớp `StaticBoard` lưu các thành phần tĩnh.

### 6.2.4 File dynamic_board.py

Chứa lớp `DynamicBoard` lưu các thành phần động.

### 6.2.5 File board_symbol.py

Chứa lớp `BoardSymbol` kế thừa lớp `Enum` lưu các ký hiệu từ tệp đầu vào. Sử dụng lớp Enum để tránh bị thay đổi từ bên ngoài, cũng như dễ thống nhất kiểu dữ liệu.

## 6.3 Thư mục extended_input

## 6.4 Thư mục search

Nơi cài đặt thuật toán tìm kiếm.

### 6.4.1 File __init__.py

Đóng gói thư mục `search` thành một module để bên ngoài sử dụng.

### 6.4.2 File algorithm.py

Chứa lớp `Algorithm` kế thừa lớp `Enum` lưu các ký hiệu các thuật toán. Sử dụng lớp Enum để tránh bị thay đổi từ bên ngoài, cũng như dễ thống nhất kiểu dữ liệu.

### 6.4.3 File search_frontier.py

Chứa lớp `SearchFrontier` đại diện cho cấu trúc dữ liệu sử dụng trong thuật toán tìm kiếm. Bằng cách truyền vào hàm khởi tạo một kiểu dữ liệu từ lớp `Algorithm`, một cấu trúc dữ liệu tương ứng sẽ được sử dụng. Lớp này giúp đơn giản hóa và trừu tượng hóa thao tác ở việc tìm kiếm.

### 6.4.4 File search_result.py

Chứa lớp `SearchResult` lưu kết quả tìm kiếm để xuất ra tệp đầu ra. Có phương thức `save\_result` để lưu kết quả ra một output stream nào đấy.

### 6.4.5 File search.py

Chứa lớp `Search`. Đây là file quan trọng nhất, chứa cài đặt của thuật toán.

### 6.4.6 File heuristics.py

Chứa lớp `Heuristics` sử dụng trong thuật toan A*.

## 6.5 File controller.py

## 6.6 Thư mục GUI

Đây là thư mục quản lý việc hiển thị giao diện của ứng dụng, nên cách cài đặt các class trong thư mục này chỉ mang tính chất hiển thị giao diện, không liên quan tới việc cài đặt thuật toán.

### 6.6.1 File general_graphic.py

Chứa các thành phần cơ bản của mê cung được trích từ `sheet.png`.

### 6.6.2 File Grid.py

Chứa class `Grid`. Class `Grid` đại diện cho trạng thái hiện tại của mê cung. Các phương thức chính của class này:

- `__init__(self, input: list[str])`: kkhởi tạo đối tượng từ các dòng input.
- `setAlgorithm(self, algorithm: str)`: gán quá trình giải mê cung của đối tượng `Grid` hiện tại.
- `__getitem__(self, index: int)`: phương thức cho phép đối tượng có thể truy cập phần tử như một danh sách.
- `next(self)`: tiến một bước trong quá trình giải mê cung
- `back(self)`: quay lại bước đi trước
- `restart(self)`: đặt lại trạng thái của mê cung

### 6.6.3 File Maze.py

Chứa class `Maze`. Class `Maze` dùng để vẽ trạng thái hiện tại của mê cung lên màn hình. Các phương thức chính của class này:

- `__init__(self, input: list[str], output: list[str])`: khởi tạo đối tượng từ các dòng input và output.
- `set_output(self, output: list[str])`: đặt lại thông số và quá trình giải mê cung từ các dòng output. Phương thức này được dùng khi tạo lại output và hiển thị lại trên màn hình.
- `setAlgorithm(self, algorithm: str)`: đổi thuật toán mê cung của đối tượng hiện tại và đặt lại trạng thái ban đầu.
- `restart(self)`: đặt lại mê cung về trạng thái ban đầu.
- `get_size(self, screenWidth: int, screenHeight: int)`: tính toán kích thước của các thành phần cơ bản từ kích thước màn hình cho trước.
- `preview(self)`: trả về màn hình xem trước của mê cung hiện tại.

- `draw(self)`: vẽ mê cung lên màn hình.
- `next(self)`: tiến một bước trong quá trình giải mê cung
- `back(self)`: quay lại bước đi trước

### 6.6.4 File MazeScreen.py

Chứa class `MazeScreen`. Class `MazeScreen` dùng để vẽ trạng thái hiện tại của mê cung lên màn hình cùng các nút điều khiển. Các phương thức chính của class này:

- `__init__(self, screen: pygame.Surface, inputFileName: str, outputFileName: str)`: khởi tạo đối tượng từ màn hình, file input và output.
- `resize(self)`: điều chỉnh giao diện tương ứng với kích thước cửa sổ hiện tại. Phương thức này được dùng khi khởi tạo và khi người dùng thay đổi kích thước cửa sổ.
- `setAlgorithm(self, algorithm: str)`: chỉnh thuật toán hiện tại
- `draw(self)`: vẽ tất cả nội dung lên màn hình.
- `start(self)`: bắt đầu chế độ tự động chạy.
- `pause(self)`: dừng chế độ tự động.
- `generate(self)`: tạo lại output thông qua `controller`.
- `handleEvent(self, event)`: xử lý sự kiện từ `pygame`, bao gồm các sự kiện về chuột và màn hình.
- `loading_screen(self)`: vẽ màn hình chờ lên màn hình.

## 6.7 File MenuScreen.py

Chứa class `MenuScreen`. Class `MenuScree` dùng để vẽ các bản xem trước của các mê cung lên màn hình và các nút chức năng. Các phương thức của class này:

- `__init__(self, screen: pygame.Surface, mazeScreens: list[pygame.Surface])`: khởi tạo đối tượng từ màn hình và mảng các màn hình xem trước của các mê cung.

- `resize(self)`: điều chỉnh giao diện tương ứng với kích thước cửa sổ hiện tại. Phương thức này được dùng khi khởi tạo và khi người dùng thay đổi kích thước cửa sổ.
- `draw(self)`: vẽ tất cả nội dung lên màn hình.
- `handleEvent(self, event: pygame.event.Event)`: xử lý sự kiện từ `pygame`, bao gồm các sự kiện về chuột và màn hình.

#### 6.7.1 Các file còn lại

Bao gồm các file tài nguyên của ứng dụng, gồm các hình ảnh được sử dụng trong ứng dụng và các phông chữ.

### 6.8 File main.py

Đây là file mã nguồn chính của chương trình, là điểm khởi chạy của ứng dụng. Trong file này, chương trình tạo của sổ giao diện bằng `pygame`, khai báo màn hình chính và các màn hình mê cung. Sau đó trong vòng lặp chương trình, xử lý các sự kiện từ người dùng và cập nhật màn hình.

## 7 Phân tích kết quả cài đặt

### 7.1 Thí nghiệm 1

#### 7.1.1 Sơ đồ mê cung

```
2 3 1 5 4
   ######
 ###    #
##. $## ##
#..$ $  @#
#.. $ $ ##
######  #
```

```
    ####
```

Mê cung này có vị trí các viên đá rất gần vị trí công tắc, và vị tri các viên đá phân bố khá đều theo chiều các công tắc. Tuy nhiên không gian để Ares đẩy đá bị hạn chế khá nhiều do đứng về một bên của tất cả các thùng, bên cạnh đó bị vướng tường và các thùng nằm gần nhau nên khó đẩy.

### 7.1.2 Kết quả đầu ra

```
BFS
Steps: 44, Weight: 47, Node: 55335, Time (ms): 3026.233,
   Memory (MB): 17.620
DFS
Steps: 47, Weight: 47, Node: 147993, Time (ms):
  7688.468, Memory (MB): 40.384
UCS
Steps: 44, Weight: 47, Node: 147360, Time (ms):
  7998.386, Memory (MB): 40.387
A_STAR
Steps: 44, Weight: 47, Node: 2966, Time (ms): 540.818,
  Memory (MB): 0.704
```

Cả 4 thuật toán đều tìm được lời giải tối ưu về trọng số do các viên đá nằm khá sát đích.

- **BFS.** Hoàn thành trong thời gian ngắn và sử dụng ít bộ nhớ. Do các viên đá nằm ở sát đích nên độ sâu cây tìm kiếm không cao, BFS sẽ phát huy được ưu thế.
- **DFS.** DFS cho một đường đi với số bước nhiều hơn, duyệt qua nhiều Node hơn dẫn đến tốn nhiều tài nguyên và thời gian hơn. Do cây tìm kiếm không cao nên việc đi sâu xuống sẽ có một số bước không cần thiết.
- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước. Do trạng thái đích gần với trạng thái bắt đầu, nên UCS vẫn tìm được đường đi với số bước tốt nhất.

- **A*.** Do có hàm đánh giá đủ tốt, A* tìm được đường đi tốt nhất với tất cả thông số, do đi thẳng đến hướng đích.

## 7.2 Thí nghiệm 2

### 7.2.1 Sơ đồ mê cung

```
2 3 1 5 4
 ####
##  #
#@$ #
##$ ##
## $ #
#.$  #
#..*.#
######
```

Mê cung này có các viên đá phân bố theo chiều vuông góc với chiều phân bố các công tắc, và không gian để đẩy theo phương pháp tuyến cũng nhỏ.

### 7.2.2 Kết quả đầu ra

```
BFS
Steps: 33, Weight: 35, Node: 985, Time (ms): 82.075,
  Memory (MB): 0.198
DFS
Steps: 35, Weight: 35, Node: 158, Time (ms): 10.549,
  Memory (MB): 0.031
UCS
Steps: 33, Weight: 35, Node: 1183, Time (ms): 79.837,
```

```
  Memory (MB): 0.133
A_STAR
Steps: 33, Weight: 35, Node: 535, Time (ms): 156.249,
  Memory (MB): 0.054
```

Trừ DFS, 3 thuật toán còn lại đều tìm được lời giải tối ưu về trọng số do các viên đá nằm khá sát đích.

- **BFS.** Do đặc thù phân bố nên BFS cần duyệt qua nhiều nút để tìm đường đi tốt nhất, dẫn đến các chi phí đều tăng cao.
- **DFS.** Không tìm được đường đi tối ưu về số bước do phải đi nhiều bước dư thừa. Tuy nhiên các thông số khác đều thấp do mê cung này có nút thắt cổ chai ở giữa, dẫn đến cây tìm kiếm không rộng.
- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước.
- **A*.** Với hàm đánh giá đủ tốt, A* đã tìm được đường đi với số nút cần duyệt qua và bộ nhớ tiêu tốn rất ít. Thời gian tìm kiếm tăng cao do độ phức tạp ở hàm đánh giá. Số nút duyệt qua ít nên không bù được nhược điểm này.

## 7.3 Thí nghiệm 3

### 7.3.1 Sơ đồ mê cung

```
5 2 4 3
 #######
 #     ###
##$###   #
# @ $  $ #
# ..# $ ##
##..#   #
```

```
########
```

Mê cung này có đường đi vào khu vực chứa công tắc rất hẹp, và Ares lại bị kẹt ở giữa những viên đá, nên phải mở đường cho chính mình ra ngoài trước. Mê cung này rất dễ dẫn đến trạng thái không thể tìm được lời giải.

### 7.3.2 Kết quả đầu ra

```
BFS
Steps: 114, Weight: 87, Node: 299748, Time (ms):
  14261.488, Memory (MB): 76.842
DFS
Steps: 1234, Weight: 1116, Node: 24877, Time (ms):
  1102.836, Memory (MB): 7.472
UCS
Steps: 114, Weight: 86, Node: 291912, Time (ms):
  13806.584, Memory (MB): 75.009
A_STAR
Steps: 114, Weight: 86, Node: 26903, Time (ms):
  3955.251, Memory (MB): 7.808
```

Các thuật toán đều duyệt qua số lượng nút lớn do cần đảm bảo đi đường đúng để không đưa mê cung vào trạng thái không thể giải tiếp được. Chỉ có UCS và A* tìm thấy lời giải tối ưu. BFS tìm thấy lời giải rất gần với lời giải tối ưu, trong khi lời giải của DFS rất xa so với lời giải tối ưu.

- **BFS.** Tìm được đường đi ngắn hơn DFS nhưng vẫn chưa phải lời giải tối ưu. Do đặc trưng duyệt rộng cần lưu trữ cả cây tìm kiếm nên tiêu tốn nhiều tài nguyên. Mê cung ở thí nghiệm này đã lớn hơn, BFS bắt đầu bộc lộ khuyết điểm, và tỏ ra không phù hợp với bài toán có không gian tìm kiếm lớn.

- **DFS.** Tuy nhanh trong việc duyệt hết các nút và số nút duyệt qua ít, nhưng không đạt tối ưu về cả số bước lẫn trọng số. Do đặc trưng duyệt sâu, DFS có thể bỏ qua nhánh chứa lời

giải tốt nhất nên tìm được giải pháp ngắn nhất, dẫn đến số bước và trọng số cao hơn so với các thuật toán khác. Bộ nhớ tiêu tốn ít nhưng độ hiệu quả về giải thuật không cao.

- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước. Tuy nhiên, UCS đã tìm dược đường đi tối ưu.
- **A*.** Tìm được đường đi tối ưu về mọi mặt do hàm đánh giá đủ tốt với trường hợp này. A* thể hiện ưu thế tuyệt đối khi tìm được lời giải với các chi phí thấp hơn các thuật toán khác khoảng 10 lần.

## 7.4 Thí nghiệm 4

### 7.4.1 Sơ đồ mê cung

```
50 20
   #####
 ###   .###
##     .   #
#    #  #$#
#    ####@#
# #    # $#
#          ##
##  #       #
#         # #
#  #  #    #
##########
```

Mê cung này khá khó khi con đường dẫn tới lời giải là đi đường vòng chứ không thể đẩy thẳng đến đích, vì sẽ dẫn tới trạng thái không thể giải.

### 7.4.2 Kết quả đầu ra

```
BFS
Steps: 77, Weight: 1170, Node: 41034, Time (ms):
  1799.585, Memory (MB): 9.793
DFS
Steps: 1277, Weight: 4360, Node: 7048, Time (ms):
  294.095, Memory (MB): 1.892
UCS
Steps: 77, Weight: 1170, Node: 49198, Time (ms):
  2226.048, Memory (MB): 13.538
A_STAR
Steps: 77, Weight: 1170, Node: 31766, Time (ms):
  3318.282, Memory (MB): 7.808
```

Trừ DFS, 3 thuật toán còn lại đều tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.

- **BFS.** Do đặc trưng của mình nên BFS cần lưu trữ cấu trúc toàn bộ cây tìm kiếm, dẫn đến tiêu tốn bộ nhớ và thời gian.
- **DFS.** Do đặc trưng của mình nên DFS thực hiện rất nhiều bước dư thừa, đẩy các viên đá đi lòng vòng, và liên tục đổi đá trong quá trình di chuyển, dẫn tới cả số bước đi lẫn tổng khối lượng đều tăng cao.
- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước.
- **A*.** Với hàm đánh giá đủ tốt, A* đã tìm được đường đi với số nút cần duyệt và bộ nhớ giảm đáng kể, trong đó bộ nhớ giảm tới một nửa. Tuy nhiên, do độ phức tạp trong khi xử lý hàm đánh giá cao, nên thời gian bị tăng cao.

## 7.5 Thí nghiệm 5

### 7.5.1 Sơ đồ mê cung

```
150 75 29 47 189
    #####
#####   #
#.. $ $ ##
#..$ $  @#
##. $## ##
 ###    #
   ######
```

Mê cung này gần như giống mê cung ở thí nghiệm 1, nhưng thay đổi trọng số, với mục tiêu để bẫy thuật toán DFS.

#### 7.5.2 Kết quả đầu ra

```
BFS
Steps: 44, Weight: 1563, Node: 298609, Time (ms):
  15567.893, Memory (MB): 92.881
DFS
Steps: 78, Weight: 2379, Node: 967, Time (ms): 147.555,
  Memory (MB): 0.102
UCS
Steps: 44, Weight: 1563, Node: 1466487, Time (ms):
  83262.562, Memory (MB): 491.272
A_STAR
Steps: 44, Weight: 1563, Node: 20289, Time (ms):
  5170.543, Memory (MB): 5.597
```

Trừ DFS, 3 thuật toán còn lại đều tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.

- **DFS.** Do chiến lược tìm kiếm đi sâu vào một nhánh duy nhất, DFS bỏ qua những nhánh khác, nơi chứa lời giải tối ưu hơn. Tuy tiêu tốt ít tài nguyên và duyệt qua ít nút hơn tất cả các thuật toán khác, nhưng không đạt được kết quả tối ưu về bất kỳ chi phí nào.

## 7.6 Thí nghiệm 6

### 7.6.1 Sơ đồ mê cung

```
10 2 1
    ####
#####  #
#   # $##
# $$.   #
#       #
# ### .##
# # # ##
#@# #. #
### ####
```

Mê cung này được thiết lập để bẫy BFS, khi đặt viên có khối lượng nhỏ nằm giữa một công tắc và một viên có khối lượng lớn.

### 7.6.2 Kết quả đầu ra

```
BFS
Steps: 32, Weight: 45, Node: 29979, Time (ms): 3890.380,
   Memory (MB): 9.324
DFS
Steps: 354, Weight: 493, Node: 9895, Time (ms):
  1001.387, Memory (MB): 2.206
```

```
UCS
Steps: 36, Weight: 40, Node: 49134, Time (ms): 5812.577,
    Memory (MB): 14.154
A_STAR
Steps: 36, Weight: 40, Node: 1551, Time (ms): 460.474,
   Memory (MB): 0.272
```

Thuật toán UCS và A* tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng. BFS tìm được đường đi gần với tối ưu, còn DFS tìm được lời giải không tối ưu.

- **BFS.** Do đặc trưng của mình nên BFS cần lưu trữ cấu trúc toàn bộ cây tìm kiếm, dẫn đến tiêu tốn bộ nhớ và thời gian. Không gian mê cung thoáng nên cây tìm kiếm sẽ vừa rộng vừa sâu, dẫn đến BFS không thể tìm được đến lời giải tối ưu trước lời giải gần tối ưu.
- **DFS.** DFS vẫn tiếp tục thể hiện nhược điểm của mình như những thí nghiệm trước, khi đã đi sâu xuống một nhánh và bỏ qua nhánh tốt nhất. Do đường đi khá thoáng nên nhiều nhánh, DFS dễ dàng rơi vào bẫy.
- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước. Tuy nhiên, UCS đã tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.
- **A*.** Với hàm đánh giá đủ tốt, A* đã tìm được đường đi với số nút cần duyệt và bộ nhớ giảm hàng chục lần. Tỉ lệ giảm về thời gian ít hơn do độ phức tạp trong hàm đánh giá.

## 7.7 Thí nghiệm 7

### 7.7.1 Sơ đồ mê cung

```
16 20
#####  ####
#   # ##@ #
# # # # $ #
#   ###  ##
```

```
#    #    $ #
# # #       #
#        ## #
##    ##    #
#   .. # # #
#   #   #    #
###########
```

Mê cung này được thiết lập với mục tiêu tương tự thí nghiệm 6, nhưng tạo một nút cổ chai để giảm bớt nhánh của cây tìm kiếm.

#### 7.7.2 Kết quả đầu ra

```
BFS
Steps: 87, Weight: 452, Node: 33022, Time (ms):
  3331.322, Memory (MB): 8.163
DFS
Steps: 213, Weight: 868, Node: 13174, Time (ms):
  1226.193, Memory (MB): 3.197
UCS
Steps: 87, Weight: 452, Node: 29291, Time (ms):
  2895.639, Memory (MB): 7.345
A_STAR
Steps: 87, Weight: 452, Node: 8374, Time (ms): 1796.899,
   Memory (MB): 1.811
```

Trừ DFS, 3 thuật toán còn lại đều tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.

- **BFS.** Do có một nút cổ chai nên số nhánh tìm kiếm bị giảm đi đáng kể, do đó dù mê cung có kích thước lớn hơn, BFS vẫn tìm thấy lời giải tối ưu. Chi phí lớn vẫn là nhược điểm của

BFS.

- **DFS.** DFS vẫn tiếp tục thể hiện nhược điểm của mình như những thí nghiệm trước, khi đã đi sâu xuống một nhánh và bỏ qua nhánh tốt nhất.
- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước. Tuy nhiên, UCS đã tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.
- **A*.** Với hàm đánh giá đủ tốt, A* đã tìm được đường đi với số nút cần duyệt và bộ nhớ giảm nhiều lần.

## 7.8 Thí nghiệm 8

### 7.8.1 Sơ đồ mê cung

```
44 1 50 5
#####
# . #
# $$#
# @ ####
#   $  ####
## ##.  $ #
#  #  #   #
#       .##
####.#####
   ###
```

Mê cung này đặt nhiều hòn đảo nổi nhằm làm rối quá trình tìm kiếm đường đi, tạo ra nhiều nhánh hơn. Ngoài ra các công tắc nằm xa nhau cũng làm tăng số lượng nhánh.

### 7.8.2 Kết quả đầu ra

```
BFS
```

```
Steps: 68, Weight: 618, Node: 563905, Time (ms):
  73297.810, Memory (MB): 157.862
Steps: 437, Weight: 1859, Node: 110299, Time (ms):
  16074.784, Memory (MB): 32.538
UCS
Steps: 80, Weight: 471, Node: 1017700, Time (ms):
  141247.024, Memory (MB): 279.616
A_STAR
Steps: 80, Weight: 471, Node: 42276, Time (ms):
  19066.068, Memory (MB): 14.588
```

Thuật toán UCS và A* đều tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.

- **BFS.** Do có nhiều nhánh trong cây hơn nên BFS đã không tìm thấy lời giải tối ưu trước những lời giải khác.
- **DFS.** DFS vẫn tiếp tục thể hiện nhược điểm của mình như những thí nghiệm trước, khi đã đi sâu xuống một nhánh và bỏ qua nhánh tốt nhất.
- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước. Tuy nhiên, UCS đã tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.
- **A*.** Với hàm đánh giá đủ tốt, A* đã tìm được đường đi với số nút cần duyệt và bộ nhớ giảm nhiều lần.

## 7.9 Thí nghiệm 9

### 7.9.1 Sơ đồ mê cung

```
5 2 4
 #####
 #@ ###
```

```
 # $  #
### # ##
#.# #  #
#.$  # #
#.   $ #
########
```

Mê cung này đặt các công tắc trong góc, dẫn tới dễ dẫn đến trường hợp đặt sai vị trí.

#### 7.9.2 Kết quả đầu ra

```
BFS
Steps: 50, Weight: 45, Node: 1690, Time (ms): 84.258,
  Memory (MB): 0.478
DFS
Steps: 52, Weight: 45, Node: 1259, Time (ms): 51.075,
  Memory (MB): 0.117
UCS
Steps: 50, Weight: 45, Node: 1703, Time (ms): 84.312,
  Memory (MB): 0.233
A_STAR
Steps: 50, Weight: 45, Node: 716, Time (ms): 92.948,
  Memory (MB): 0.098
```

Cả 4 thuật toán đều tìm được đường đi tối ưu cả về tổng khối lượng.

- **BFS.** Tuy tìm dược lời giải tối ưu, nhưng chi phí để BFS nuôi cây tìm kiếm vẫn lớn.
- **DFS.** DFS gặp nhiều rắc rối khi rơi nhiều vào trường hợp không thể giải, dẫn tới số nút được duyệt khá cao. Tuy tìm được lời giải tối ưu về tổng khối lượng nhưng chưa tìm được lời giải tối ưu về số bước.

- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước. Tuy nhiên, UCS đã tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.
- **A*.** Với hàm đánh giá đủ tốt, A* đã tìm được đường đi với số nút cần duyệt và bộ nhớ giảm nhiều lần.

## 7.10 Thí nghiệm 10

### 7.10.1 Sơ đồ mê cung

```
5 2 4 3 6
   #######
  ##   # @#
  #    #   #
  #$ $ $ #
  # $##   #
### $ # ##
#.....   #
#########
```

Mê cung này cùng ý tưởng với thí nghiệm 9, nhưng tăng kích thước mê cung và số đá để tăng không gian tìm kiếm.

### 7.10.2 Kết quả đầu ra

```
BFS
Steps: 79, Weight: 106, Node: 337357, Time (ms):
  34984.126, Memory (MB): 120.815
DFS
Steps: 688, Weight: 367, Node: 14394, Time (ms):
  630.462, Memory (MB): 3.978
```

```
UCS
Steps: 91, Weight: 94, Node: 821038, Time (ms):
  51709.332, Memory (MB): 243.928
A_STAR
Steps: 91, Weight: 94, Node: 46315, Time (ms):
  10608.564, Memory (MB): 14.801
```

Chỉ có UCS và A* tìm được đường đi tối ưu.

- **BFS.** Do không gian tìm kiếm lớn, BFS tìm thấy một lời giải khác trước khi tìm được lời giải tối ưu.
- **DFS.** Do không gian nhiều nhánh, DFS tiếp tục đi vào nhánh không chứa lời giải tối ưu trước, dẫn tới tìm được lời giải không tối ưu.
- **UCS.** Sử dụng thời gian và bộ nhớ nhiều do phải đánh giá chí phí di chuyển cho từng bước. Tuy nhiên, UCS đã tìm được đường đi tối ưu cả về số bước lẫn tổng khối lượng.
- **A*.** Với hàm đánh giá đủ tốt, A* đã tìm được đường đi với số nút cần duyệt và bộ nhớ giảm nhiều lần.