

BÀI BÁO CÁO PROJECT 1

CƠ SỞ TRÍ TUỆ NHÂN TẠO



Lê Đại Hòa - 22120108

NGUYỄN TƯỜNG BÁCH HỖ - 22120455

LIÊU HẢI LƯU DANH - 22120459

Lê Hoàng Vũ - 22120461

Ngày 22 tháng 10 năm 2024

Mục lục

I	TÓM TẮT NỘI DUNG	3
1	Project 1: Search - Ares's Adventure	4
2	Giới thiệu sản phẩm	5
2.1	Về sản phẩm Ares: Holy Road to AI	5
2.2	Hướng dẫn	5
2.3	Bảng phân công công việc	6
2.4	Source code, Link Video và tài liệu tham khảo	6
II	CHI TIẾT ĐỒ ÁN	7
3	Graphical User Interface	8
4	Input	11
4.1	Quy ước chung về đầu vào	11
4.2	Xử lý đầu vào	13
5	Các thuật toán tìm kiếm	16
5.1	Breadth First Search (BFS)	16
5.2	Depth First Search (DFS)	18
5.3	Uniform Cost Search (UCS)	20
5.4	A* (A-star):	22
6	Kết quả kiểm thử	24

I

TÓM TẮT NỘI DUNG

1 Project 1: Search - Ares's Adventure

Ares's Adventure hay Cuộc phiêu lưu của Ares là một trò chơi thú vị yêu cầu bạn hướng dẫn nhà thám hiểm trẻ tên Ares vượt qua một mê cung phức tạp để mở khóa cánh cổng dẫn đến kho báu huyền thoại. Để làm được điều này, Ares phải di chuyển những tảng đá lớn đến các công tắc bí mật nằm trong mê cung. Việc di chuyển đòi hỏi sự tính toán tỉ mỉ, vì không gian hẹp và chỉ cần một sai lầm nhỏ cũng có thể khiến anh mắc kẹt vĩnh viễn. Nhiệm vụ của bạn là giúp Ares chinh phục mê cung và khám phá kho báu!

Trên thực tế ở trên chính là một phát biểu khác của trò chơi **Sokoban** cổ điển.

Trò chơi (Sokoban)

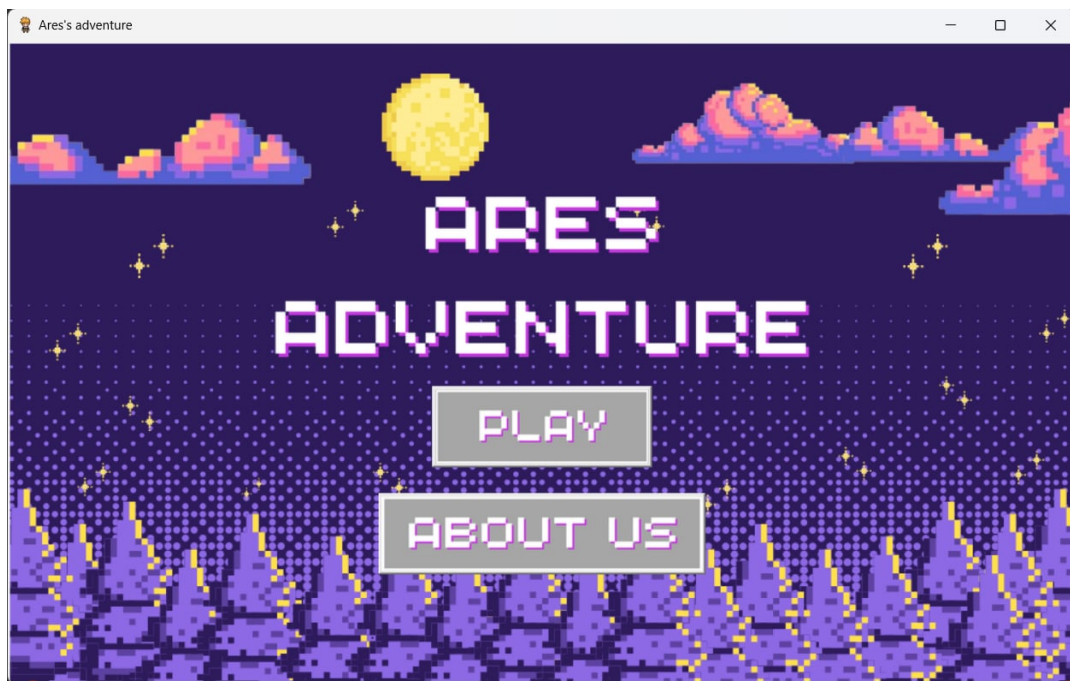
Sokoban là một trò chơi giải đố cổ điển được phát triển tại Nhật Bản vào năm **1981**. Tên gọi "*Sokoban*" trong tiếng Nhật có nghĩa là "người giữ kho". Trong trò chơi, người chơi vào vai một nhân vật người giữ kho với nhiệm vụ đẩy các thùng hàng đến đúng vị trí đích đã được đánh dấu trên bản đồ. Tuy luật chơi đơn giản - chỉ được đẩy một thùng mỗi lần và không thể kéo thùng - nhưng Sokoban lại là một thử thách trí tuệ thực sự. Người chơi phải tính toán cẩn thận từng bước đi trong không gian giới hạn, tránh đẩy thùng vào góc chết hay những vị trí không thể di chuyển tiếp. Mỗi màn chơi đòi hỏi khả năng tư duy logic và lập kế hoạch trước để tìm ra chuỗi di chuyển tối ưu nhất, biến trò chơi đơn giản này thành một bài toán phức tạp và hấp dẫn.

Tất nhiên ở Project này sẽ không chỉ đơn giản là **Sokoban** cổ điển. Điểm nhấn của Project này chính là thông qua trợ giúp của các **thuật toán tìm kiếm** trong Trí tuệ Nhân tạo, người chơi giúp cho Ares chinh phục kho báu và mê cung, thậm chí là với **những đường đi tối ưu nhất!**

2 Giới thiệu sản phẩm

§2.1 Về sản phẩm Ares: Holy Road to AI

Như đã nói, đây không còn là một phiên bản trò chơi Sokoban thông thường khi mà Ares không còn phải cặm cụi dò xét cả mê cung (map) chỉ để tìm ra con đường thích hợp đẩy các tảng đá vào vị trí các công tắc. Giờ đây, nhà thám hiểm này sẽ có thêm 'trang bị' là những kiến thức về thuật toán tìm kiếm mà anh có lẽ đã được học từ môn Trí tuệ Nhân tạo. Chính vì vậy, với sản phẩm cho đồ án lần này, nhóm quyết định sẽ đặt tên phần mềm giải quyết bài toán Ares's Adventure là Ares: Holy Road to AI.



§2.2 Hướng dẫn

Mê cung, hay còn được gọi là map của trò chơi này, sẽ được tạo ra trong một không gian kích thước $m \times n$, trong đó mỗi ô một là không gian trống, hai là sẽ chứa vật thể. Những vật thể ở đây có thể là tường, đá, hoặc công tắc. Nhân vật chính của chúng ta - Ares sẽ khởi đầu màn chơi ở một vị trí cụ thể trên map và có thể di chuyển từng ô một theo 4 hướng: Lên, Xuống, Trái, Phải. Anh ta không thể đi xuyên qua tường hoặc đá. Điều này có nghĩa là sẽ có trường hợp Ares có thể giẫm lên công tắc.

Nếu ô liền kề chứa đá và ô phía sau đá là ô trống, anh ta có thể đẩy đá vào ô trống đó. Anh ta không thể kéo đá về phía mình, không thể tiếp tục đẩy đá về phía tường nếu đá tiếp xúc, và cuối cùng là không thể đẩy chồng hai tảng đá cùng một lúc.

Mỗi viên đá có trọng lượng (trọng số) khác nhau, điều này có nghĩa là Ares phải cân nhắc cách hiệu quả nhất để đẩy đá lên các công tắc. Việc đẩy những viên đá nặng

hơn tốn nhiều công sức hơn và hạn chế khả năng di chuyển của anh ta. Điều này đòi hỏi cần có một chiến lược tối ưu khi quyết định đẩy viên đá nào và khi nào đẩy. Mục tiêu là đẩy tất cả các viên đá lên công tắc. Mỗi công tắc có thể được kích hoạt bởi bất kỳ viên đá nào với bất kỳ trọng lượng nào, và chắc chắn số lượng đá luôn bằng số lượng công tắc.

Và để tìm ra chiến lược tối ưu - vừa đỡ mang nặng, vừa di chuyển quãng đường ngắn hơn, thì Ares sẽ phải vận dụng các thuật toán tìm kiếm sau đây:

- Tìm kiếm theo chiều rộng (BFS)
- Tìm kiếm theo chiều sâu (DFS)
- Tìm kiếm chi phí đồng nhất (UCS)
- Tìm kiếm A* với heuristic.

§2.3 Bảng phân công công việc

Thứ tự	Vai trò	Mô tả	Người thực hiện	Độ hoàn thành
1	Developer	Code Search	Lê Hoàng Vũ	100%
		Code xử lý map	Nguyễn Tường Bách Hỷ	100%
		Code GUI	Lê Đại Hòa	100%
		Code xử lý Deadlock	Liêu Hải Lưu Danh	100%
2	Tester	Kiểm lỗi	Liêu Hải Lưu Danh	100%
3	Designer	Thiết kế GUI	Lê Đại Hòa	100%
4	Báo cáo		Lê Hoàng Vũ	100%
			Lê Đại Hòa	100%
5	Video	Thuyết trình	Nguyễn Tường Bách Hỷ	100%
		Chỉnh sửa video	Lê Đại Hòa	100%

§2.4 Source code, Link Video và tài liệu tham khảo

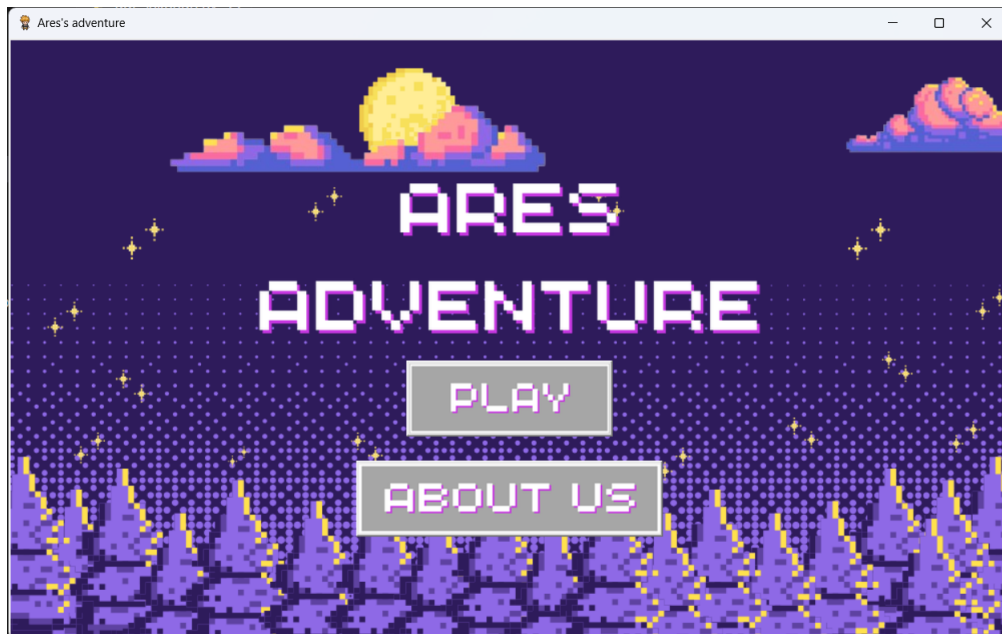
- [1] [Link video hướng dẫn](#)
 [2] [Source code](#)
 [3] [Link tham khảo](#)

II

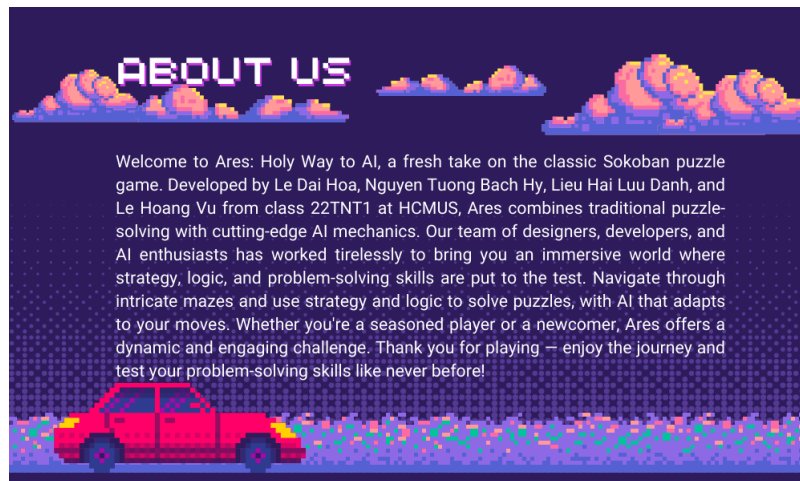
CHI TIẾT ĐỒ ÁN

3 Graphical User Interface

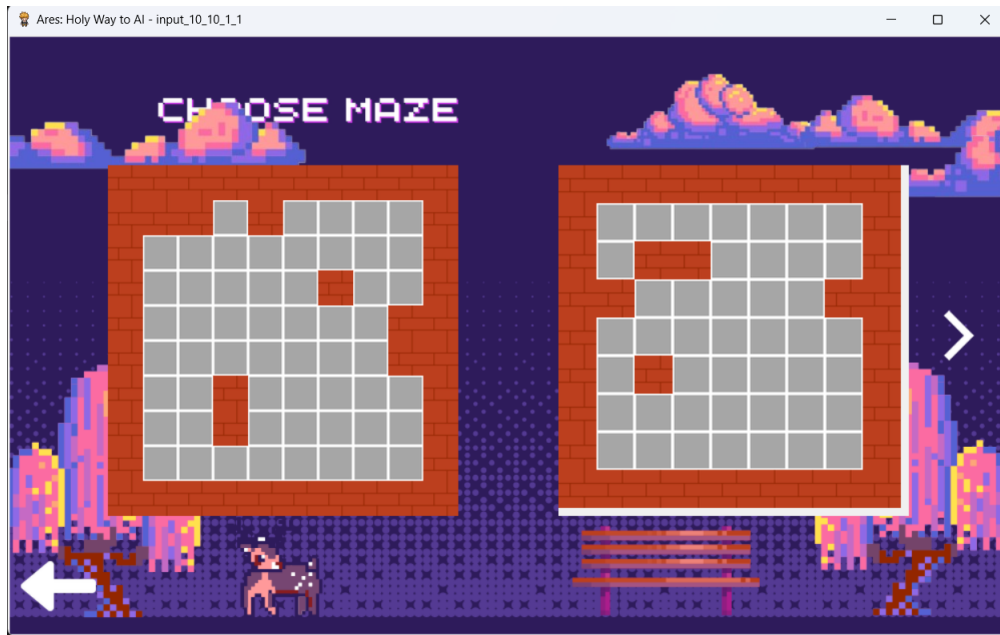
- ① Trang MENU: Gồm 2 nút là START (để bắt đầu chơi) và ABOUT US (giới thiệu về nhóm)



- ② Trang ABOUT US: giới thiệu chung về trò chơi Ares: Holy Way to AI và các thành viên trong nhóm.



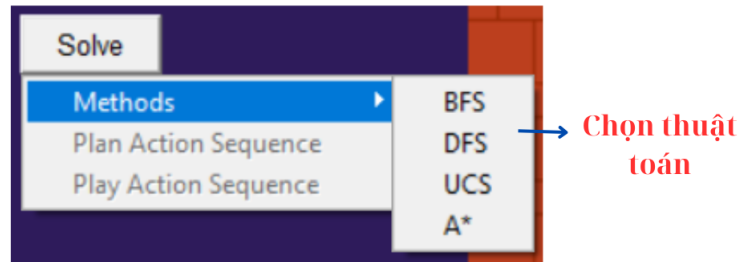
- ③ Trang CHOOSE MAZE: sau khi bấm START ở trang MENU, người chơi sẽ chọn mê cung mà mình muốn thử thách.



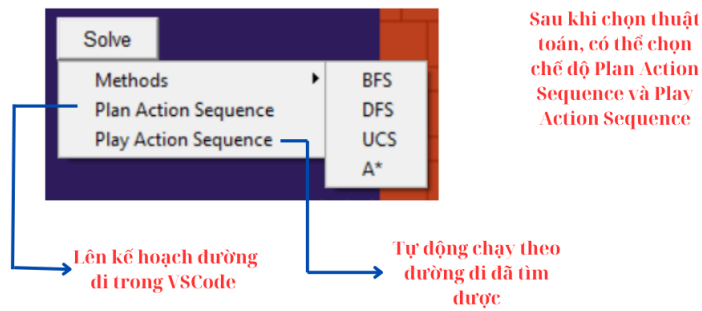
- ④ Trang chính khi chơi game: Gồm có mê cung trò chơi, nút Restart để bắt đầu lại, nút Solve để lựa chọn thuật toán, nút Quit để kết thúc.



- ⑤ Chọn thuật toán:



⑥ Chọn chế độ



⑦ Khi kết thúc, nếu hoàn thành thì sẽ có lời chúc mừng hiện ra. Nếu muốn tiếp tục, nhấn BACK.



4 Input

§4.1 Quy ước chung về đầu vào

Yêu cầu của đầu vào (input) là nên có các tham số khác nhau như số lượng đá trên mê cung, trọng lượng của từng viên đá, ... để dựa vào đó có thể hiểu được cấu trúc của mê cung cũng như vị trí của các vật thể, của người chơi **Ares**. Với sản phẩm **Ares: Holy Road to AI**, nhóm chúng em xin được giới thiệu đến độc giả hai kiểu đầu vào sau đây:

① **Đầu vào cơ bản:** Dòng đầu tiên chứa danh sách các số nguyên dương thể hiện trọng lượng của từng viên đá, theo thứ tự chúng xuất hiện trong map, từ trái sang phải và từ trên xuống dưới. Các dòng tiếp theo mô tả vị trí các vật thể trên bản đồ theo quy ước ký hiệu như dưới:

- # tượng trưng cho tường.
- 'whitespace' tượng trưng cho khoảng trống không có vật thể trên map.
- \$ tượng trưng cho đá.
- @ tượng trưng cho người chơi Ares.
- . tượng trưng cho công tắc.
- * tượng trưng cho đá đang đè lên công tắc sẵn rồi.
- + tượng trưng cho Ares đang giẫm lên công tắc.

Câu hỏi — Vấn đề được đặt ra là: Nếu phải tạo thủ công nhiều file input như trên, chẳng phải sẽ khá 'mệt' sao? Và nếu tồn tại một kiểu đầu vào khác tối ưu hơn thì sao?


Suy cho cùng, một đầu vào tiêu chuẩn là một đầu vào mà trước hết là có thể dễ dàng tạo ra số lượng lớn với hiệu quả tốt nhất. Mà quan trọng hơn hết là dựa vào đó, người cũng như máy có thể dễ dàng đọc và hiểu được những thuộc tính của map. Vì những trăn trở trên, nhóm chúng em đã thử tìm tòi và tham khảo một số cách để tạo ra đầu vào cho kiểu trò chơi này. Từ đó, xin được giới thiệu đến độc giả kiểu đầu vào sẽ được dùng chủ yếu trong **Ares: Holy Road to AI**.

② **Đầu vào số hóa:** Ngay từ cái tên, ta đã có thể hình dung rằng nội dung file đầu vào ở dạng này sẽ toàn là những con số. Cụ thể ta có định nghĩa đầu vào số hóa như sau:

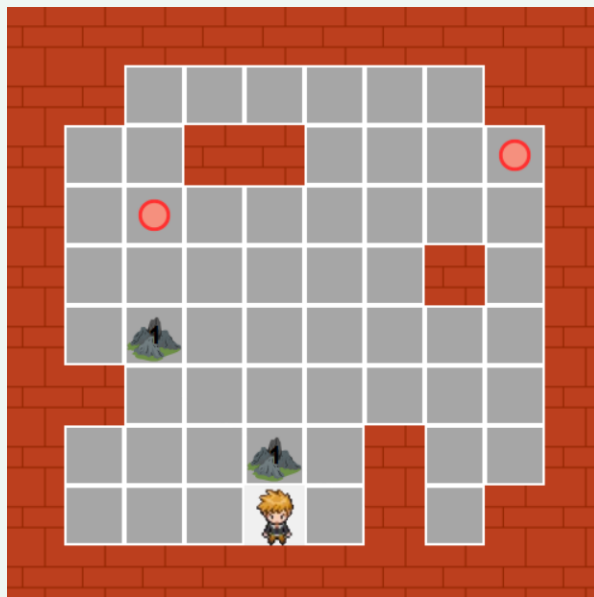
Định nghĩa 1 (Đầu vào số hóa)

Đầu vào số hóa về mặt cấu trúc gồm 6 dòng như sau:

1. **nRows, nCols**: Kích thước của map. ví dụ: **10 10** nghĩa là map có kích thước 10×10 .
2. **nWalls**: số lượng tường bao quanh map, đi kèm một danh sách tọa độ các ô tường, ví dụ: **42 10 10 8 10 ...** thì **42** là số lượng block tường, các block đó có tọa độ lần lượt là **(10, 10), (8, 10), ...**
3. **nStones**: số lượng đá đi kèm với danh sách tọa độ của các tảng đá này.
4. **nSwitches**: số lượng công tắc đi kèm với danh sách tọa độ của các công tắc.
5. **pos**: Vị trí ban đầu của Ares, ví dụ: **2 6** là Ares đang ở tọa độ **(2, 6)**.
6. **weights**: trọng lượng của từng viên đá, theo thứ tự chúng xuất hiện trong map, từ trái sang phải và từ trên xuống dưới.

 Hãy đến với một ví dụ để hiểu hơn về định nghĩa (1)

Ví dụ 1 — Đầu tiên hãy theo dõi một map sau đây:



Ứng với map trên, ta có đầu vào số hóa như bên dưới:

```

1 10 10
2 45 10 10 2 2 8 10 6 10 4 10 2 9 2 10 9 1 7 1 7 2 5 1 3 1 9 7 1 1 9 9
   9 10 1 2 3 4 1 3 3 5 7 10 1 4 5 8 1 5 5 10 1 6 1 7 3 10 1 8 1 9 1
   10 10 1 10 2 8 1 10 3 10 4 6 1 10 5 10 6 4 1 10 7 10 8 2 1 10 9 8
   7
3 2 6 3 8 5
4 2 3 9 4 3
5 9 5
6 1 1

```

§4.2 Xử lý đầu vào

Sau khi có một số quy ước về cú pháp như trên đã đề cập, mục tiêu tiếp theo chính là viết một chương trình để đọc những file đầu vào có định dạng kiểu số hóa, từ đó giúp phần mềm load được map.

Ý tưởng:

- ① Tạo lớp đọc và phân tích file:

Chẳng hạn ta gọi lớp này là `MapParser`. Đây sẽ là lớp chịu trách nhiệm đọc file đầu vào và phân tích để khởi tạo một bản đồ dưới dạng ma trận **2D** (map). Mỗi ô trong ma trận sẽ đại diện cho một vị trí trong bản đồ

Algorithm 4.2.1 Map Parser Algorithm

```

1: procedure MAP_PARSER(filename)
2:   Input: filename: string - path to input file
3:   Output: map: 2D array, weights: array of integers
4:   lines  $\leftarrow$  read all lines from file
5:   lines  $\leftarrow$  strip whitespace from each line
6:   [width, height]  $\leftarrow$  split lines[0] by space
7:   map  $\leftarrow$  2D array of ' ' with size width  $\times$  height
8:   player_coordinate  $\leftarrow$  split lines[-2] by space
9:   weights  $\leftarrow$  convert lines[-1] to array of integers
10:  px  $\leftarrow$  player_coordinate[0] as integer - 1
11:  py  $\leftarrow$  player_coordinate[1] as integer - 1
12:  if map[px][py] = ' ' then
13:    map[px][py]  $\leftarrow$  '@'
14:  else
15:    map[px][py]  $\leftarrow$  '+'
16:  parse_line(lines[1], map, '#', '#')
17:  parse_line(lines[2], map, '$', '*')
18:  parse_line(lines[3], map, '.', '*')
19:  return map, weights

```

Thêm phương thức dùng để gán các ký tự cho bản đồ theo các tọa độ lấy từ đầu vào. Chẳng hạn gọi là `parse_line`.

Algorithm 4.2.2 Parse Line Algorithm

```

1: procedure PARSE_LINE(line, map, char, replace_char)
2:   Input: line: string of coordinates, map: 2D game map, char, replace_char
3:   parsed_line  $\leftarrow$  split line by space, i  $\leftarrow$  1
4:   while i < length of parsed_line do
5:     (x, y)  $\leftarrow$  parsed_line[i], parsed_line[i + 1] as integers
6:     map[x-1][y-1]  $\leftarrow$  replace_char if map[x-1][y-1]  $\neq$  ' ' else char
7:     i  $\leftarrow$  i + 2

```

- ② Thêm lớp có nhiệm vụ tạo đối tượng bản đồ từ dữ liệu của lớp `MapParser`:

Chẳng hạn gọi lớp này là `Map`. Đây sẽ là lớp gọi `MapParser` để nhận được bản đồ dưới dạng danh sách hai chiều (`map`) và danh sách trọng số (`weights`).

Thêm các phương thức để nhận các nhiệm vụ sau:

- Chuyển đổi bản đồ dạng ma trận sang dạng chuỗi để dễ dàng lưu trữ và hiển thị. Chẳng hạn gọi là `from_lines`.
- Dò tìm các ký tự đặc biệt để xác định vị trí của người chơi. Chẳng hạn gọi là `extract_locations`.

Algorithm 4.2.3 Map Parsing and Location Extraction

```

1: procedure FROM_LINES(lines)
2:   Find first wall position: first_row_brick, first_col_brick
3:   if no walls found then
4:     raise ValueError("Maze with no walls!")
5:   canonical_lines ← lines from first_row_brick and first_col_brick
6:   ncols ← max column index of walls + 1
7:   nrows ← length of canonical_lines
8:   EXTRACT_LOCATIONS(canonical_lines)
9: procedure EXTRACT_LOCATIONS(lines)
10:  worker ← Find worker position ('@' or '+')
11:  boxes ← Find box positions ('')  targets ← Find target positions ('.')
12:  boxes ← boxes + targets with boxes ('*')
13:  walls ← Find wall positions ('')
14:  Sort boxes by coordinates
15:

```

Quan trọng hơn hết sẽ là cần một phương thức chịu trách nhiệm lấy dữ liệu từ file và chuyển nó thành một bản đồ có thể sử dụng được trong trò chơi. Chẳng hạn gọi phương thức này là `load_warehouse`.

Algorithm 4.2.4 Load Warehouse Algorithm

```

1: procedure LOAD_WAREHOUSE(filepath)
2:   Input: filepath: đường dẫn tới file bản đồ
3:   Output: Cập nhật các thuộc tính của warehouse (map, weights, etc.)
4:   map, weights ← MAPPARSER(filepath)  ▶ Sử dụng MapParser để đọc file
5:   lines ← join các dòng của map thành chuỗi
6:   FROM_LINES(lines) ▶ Gọi phương thức từ lines để xử lý bản đồ và xác định vị trí

```

Với cách tiếp cận này, `load_warehouse` hoạt động như cầu nối giữa dữ liệu thô từ file đầu vào và đối tượng bản đồ trong trò chơi. Phương thức này giúp:

- Đảm bảo rằng `Warehouse` có thể đọc file đầu vào với cấu trúc định sẵn.
- Tạo ra một bản đồ dễ thao tác và quản lý các yếu tố cần thiết trong trò chơi.

Quy trình xử lý:

- Bước 1: `Warehouse` gọi phương thức `load_warehouse` với đường dẫn file làm tham số. Phương thức này sẽ sử dụng `MapParser` để đọc và phân tích dữ liệu từ file.

- Bước 2: `MapParser` phân tích dữ liệu từ file và tạo ra ma trận bản đồ cùng danh sách trọng số (nếu có).
- Bước 3: `Warehouse` sử dụng ma trận bản đồ để xác định vị trí của các đối tượng, lưu trữ chúng vào các thuộc tính riêng, và xử lý logic về kiểm tra điều kiện trong trò chơi.

5 Các thuật toán tìm kiếm

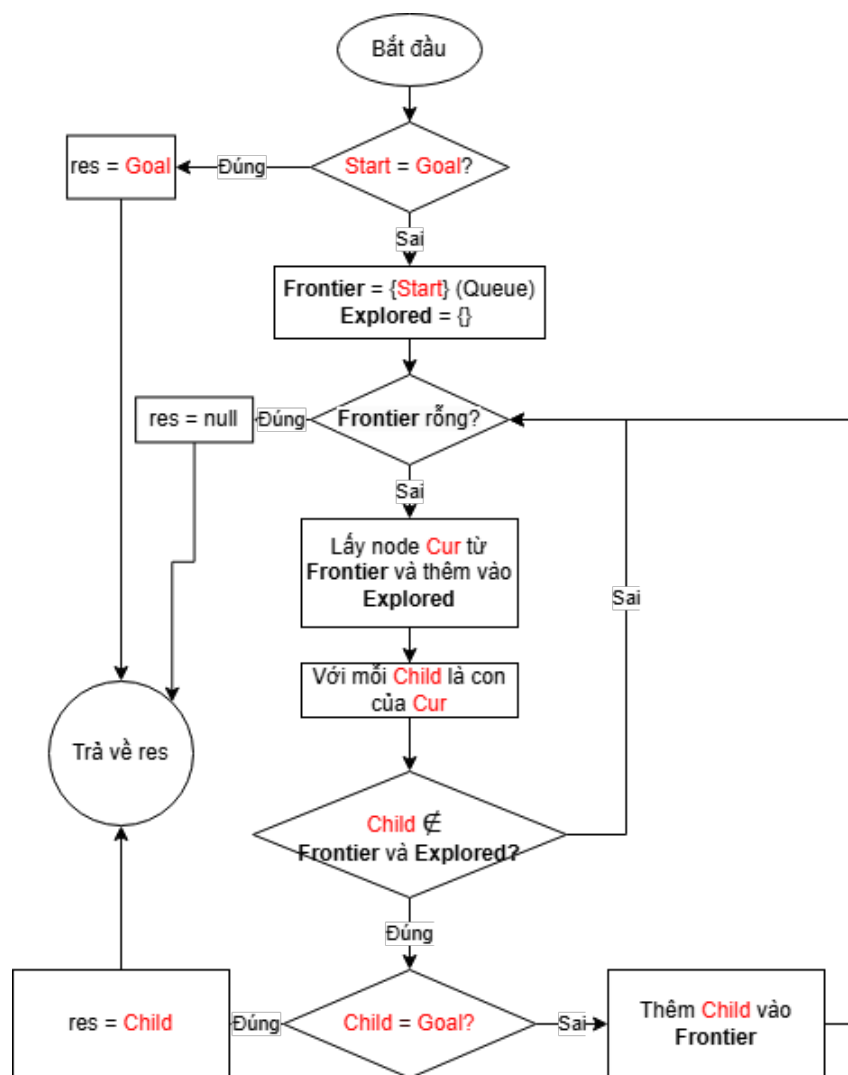
§5.1 Breadth First Search (BFS)

① **Cơ chế hoạt động:** là thuật toán duyệt đồ thị theo từng lớp, nghĩa là nó sẽ mở rộng các nút ở mức gần nhất trước khi tiếp tục sang các mức xa hơn. BFS đảm bảo tìm được đường đi ngắn nhất nếu mỗi bước di chuyển có cùng chi phí.

③ **Ý tưởng thực hiện:**

- Sử dụng hàng đợi (FIFO Queue) để lưu trữ các nút trong quá trình duyệt.
- Bắt đầu từ trạng thái ban đầu, đưa các trạng thái tiếp theo vào hàng đợi.
- Duyệt lần lượt các nút trong hàng đợi cho đến khi tìm thấy đích hoặc duyệt hết các trạng thái.

③ **Quy trình:**



④ Mã giả:

Algorithm 5.1.1 Breadth-First Search (BFS) Algorithm

```

1: procedure BFS(Start, Goal)
2:   if Start = Goal then
3:     return Goal                                ▶ If the start is the same as goal, return goal
4:   Frontier  $\leftarrow$  Queue()
5:   Explored  $\leftarrow \emptyset$                     ▶ Set of explored nodes
6:   res  $\leftarrow$  null                             ▶ Variable to store result
7:   Frontier.push(Start)                          ▶ Add start node to Frontier
8:   while Frontier is not empty do
9:     Cur  $\leftarrow$  Frontier.pop()                ▶ Get the first node from Frontier
10:    Explored.push(Cur)                          ▶ Add current node to Explored
11:    for each Child in Cur.child() do
12:      if Child not in Frontier and Child not in Explored then
13:        if Child = Goal then
14:          res  $\leftarrow$  Child
15:          return res                            ▶ Goal found, return the result
16:        Frontier.push(Child)                    ▶ Add Child to Frontier to explore later
17:   return res                                    ▶ Return null if no path found

```

⑤ **Độ phức tạp thời gian:** $O(b^d)$ với b là hệ số phân nhánh và d là độ sâu của nút đích gần nhất.

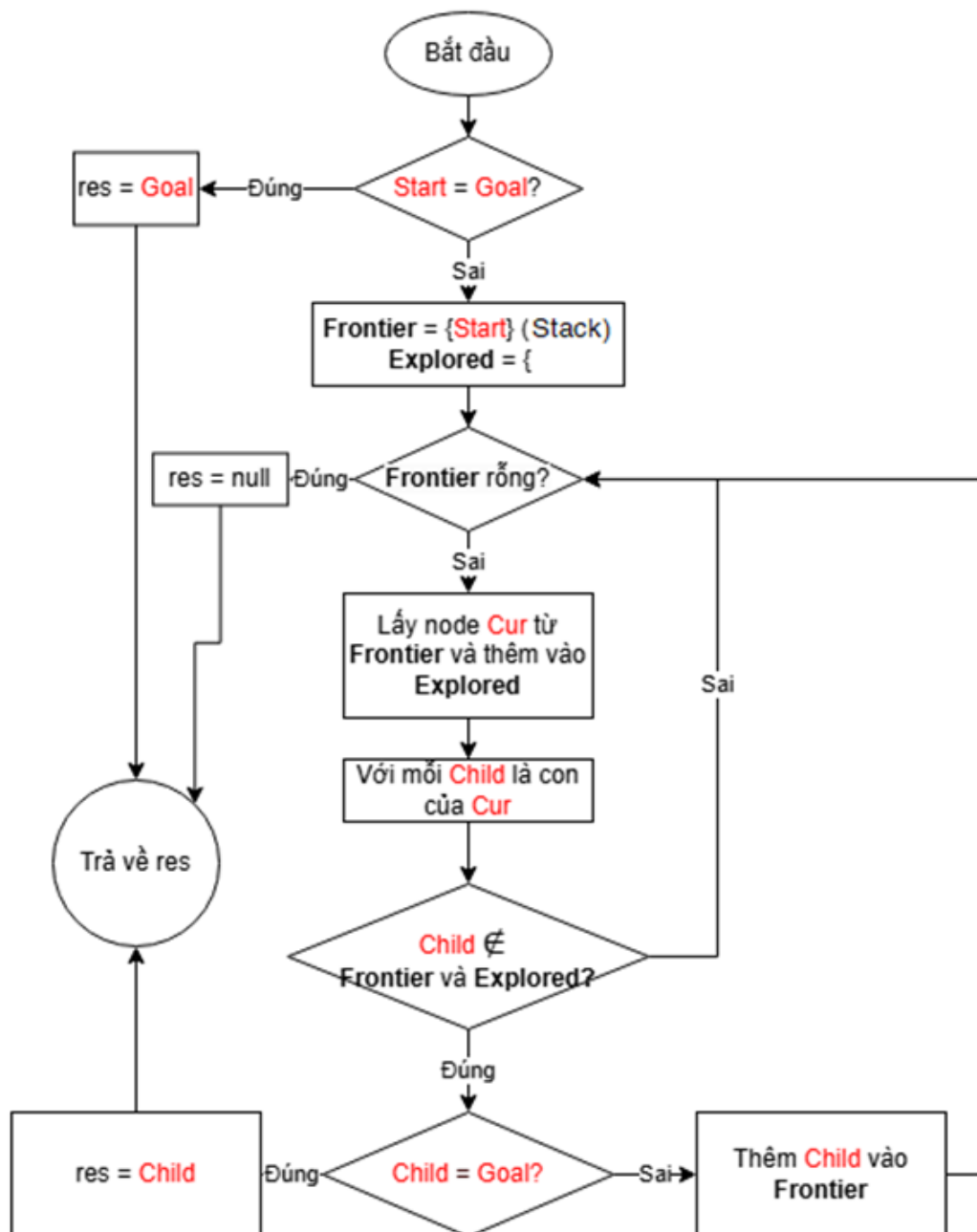
§5.2 Depth First Search (DFS)

① **Cơ chế hoạt động:** DFS duyệt sâu vào các nhánh trước khi quay lại các nhánh khác. Tuy nhiên, nó không đảm bảo tìm đường đi ngắn nhất và có thể dẫn đến vòng lặp nếu không xử lý trạng thái đã duyệt.

② **Ý tưởng thực hiện:**

- Sử dụng ngăn xếp (LIFO Stack) để lưu trữ các nút.
- Bắt đầu từ trạng thái ban đầu và liên tục mở rộng sâu nhất có thể.
- Quay lại các nhánh khác khi đi vào ngõ cụt.

③ **Quy trình:**



④ Mã giả:

Algorithm 5.2.1 Depth-First Search (DFS) Algorithm

```

1: procedure DFS(Start, Goal)
2:   if Start = Goal then
3:     return Goal                                ▶ If the start is the same as goal, return goal
4:   Frontier  $\leftarrow$  Stack()                      ▶ Initialize the stack (LIFO)
5:   Explored  $\leftarrow \emptyset$                       ▶ Set of explored nodes
6:   res  $\leftarrow$  null                               ▶ Variable to store the result
7:   Frontier.push(Start)                          ▶ Add start node to Frontier
8:   while Frontier is not empty do
9:     Cur  $\leftarrow$  Frontier.pop()                  ▶ Pop the top node from the stack
10:    Explored.push(Cur)                            ▶ Add current node to Explored
11:    for each Child in Cur.child() do
12:      if Child not in Frontier and Child not in Explored then
13:        if Child = Goal then
14:          res  $\leftarrow$  Child
15:          return res                              ▶ Goal found, return the result
16:        Frontier.push(Child)                      ▶ Push Child to Frontier to explore later
17:   return res                                    ▶ Return null if no path found

```

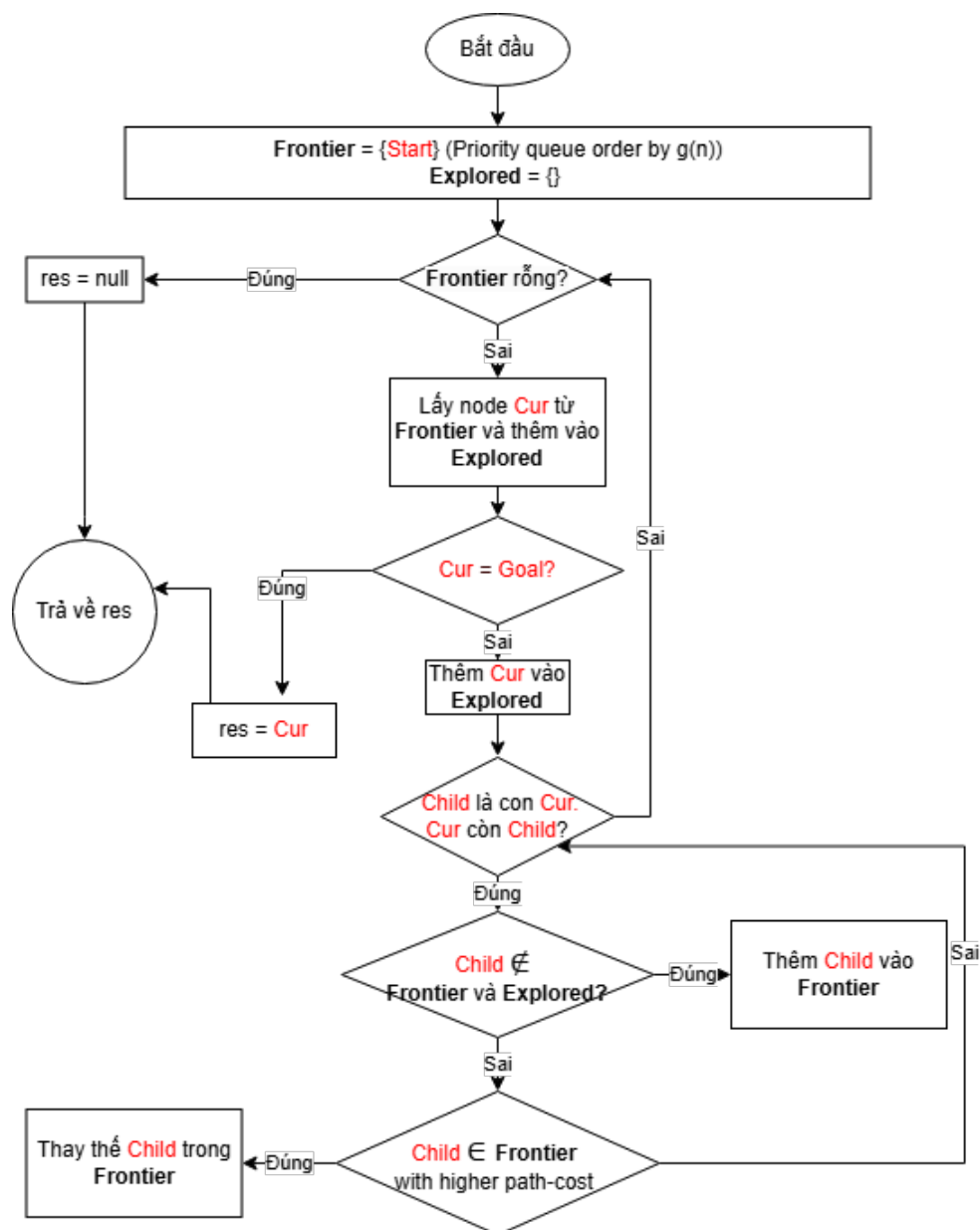
§5.3 Uniform Cost Search (UCS)

① **Cơ chế hoạt động:** UCS là một thuật toán tìm kiếm tối ưu chi phí, luôn mở rộng nút có chi phí thấp nhất từ trạng thái ban đầu đến nút đó. UCS thích hợp khi mỗi bước đi chuyển có trọng số khác nhau và cần tìm đường có tổng chi phí thấp nhất.

② **Ý tưởng thực hiện:**

- Sử dụng hàng đợi ưu tiên (Priority Queue) để lưu các nút dựa trên chi phí đường đi từ trạng thái ban đầu.
- Mở rộng nút có chi phí thấp nhất trong hàng đợi.
- Tiếp tục mở rộng cho đến khi tìm thấy đích.

③ **Quy trình:**



④ Mã giả:

Algorithm 5.3.1 Uniform Cost Search (UCS) Algorithm

```

1: procedure UCS(Start, Goal)
2:   Initialize Frontier  $\leftarrow$  priority_queue()      ▶ Priority queue based on  $g(n)$ 
3:   Initialize Explored  $\leftarrow \emptyset$                 ▶ Set of explored nodes
4:    $g(\text{Start}) \leftarrow 0$                             ▶ Set the cost of the start node
5:   Frontier.push(Start)                                ▶ Add Start to the Frontier with cost  $g(\text{Start})$ 
6:   while Frontier is not empty do
7:     Cur  $\leftarrow$  Frontier.pop()    ▶ Pop the node with the lowest  $g(n)$  from Frontier
8:     if Cur = Goal then
9:       return Cur                                ▶ Goal found, return the path
10:    Explored.push(Cur)                            ▶ Add the current node to Explored
11:    for each Child in Cur.child() do
12:      newG  $\leftarrow g(\text{Cur}) + \text{cost}(\text{Cur}, \text{Child})$  ▶ Calculate the new cost to reach
      Child
13:      if Child not in Frontier and Child not in Explored then
14:         $g(\text{Child}) \leftarrow \text{newG}$ 
15:        Frontier.push(Child)    ▶ Push Child into Frontier with new cost
16:      else if Child in Frontier and newG <  $g(\text{Child})$  then
17:         $g(\text{Child}) \leftarrow \text{newG}$  ▶ Update the cost of Child in Frontier if the new
      cost is lower
18:        Frontier.replace(Child) ▶ Replace Child in Frontier with the updated
      cost
19:   return failure                                ▶ Return failure if no path found

```

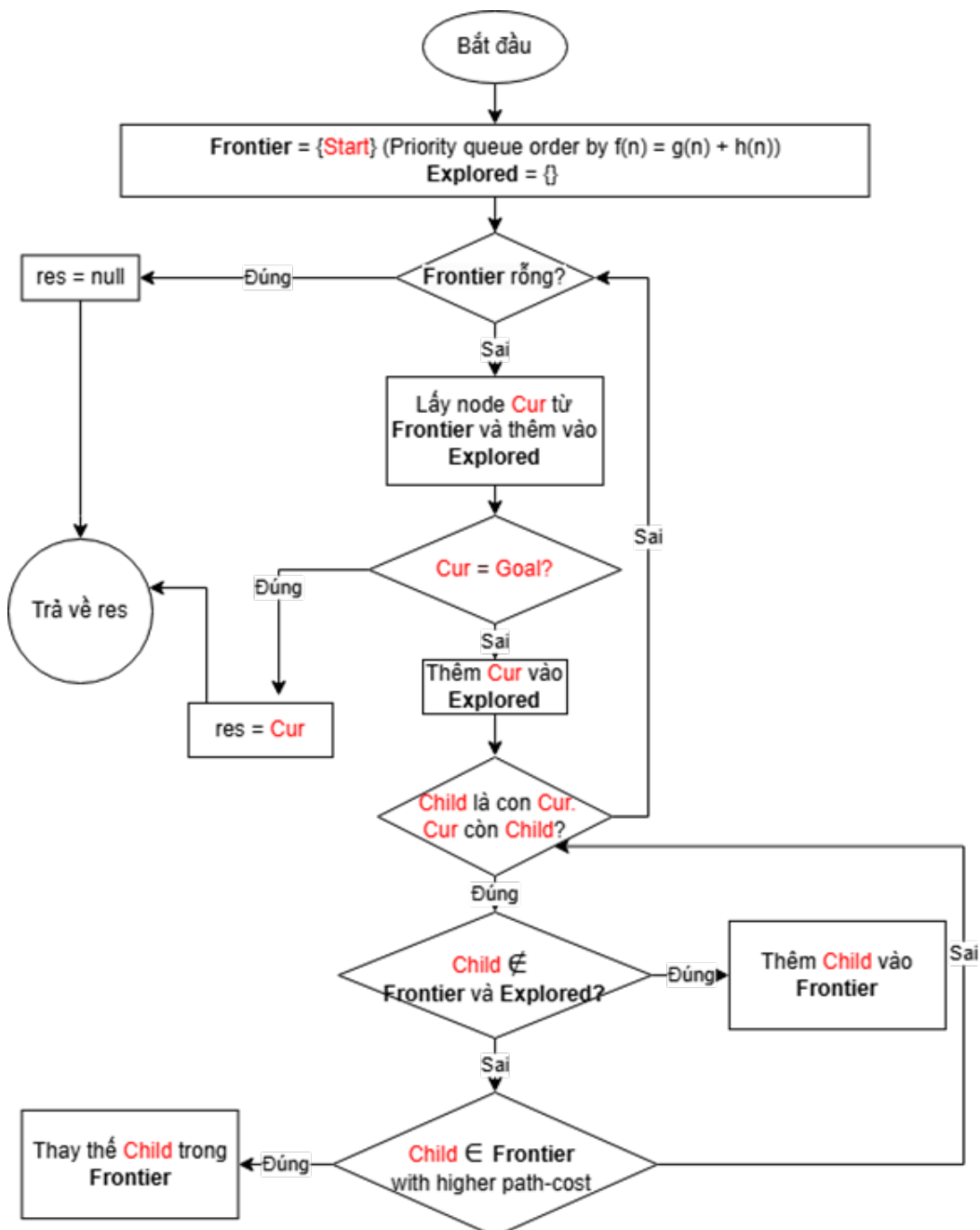
§5.4 A* (A-star):

① **Cơ chế hoạt động:** A* là một thuật toán tìm kiếm *heuristic*. Nó sử dụng hàm đánh giá tổng chi phí đến hiện tại và ước lượng chi phí từ điểm hiện tại đến đích để tối ưu việc tìm kiếm. A* hoạt động hiệu quả khi có hàm *heuristic* tốt, giúp giảm thiểu số lượng nút cần duyệt để đạt đến đích.

② **Ý tưởng thực hiện:**

- Sử dụng hàng đợi ưu tiên, nơi mỗi nút được sắp xếp theo tổng chi phí đi từ trạng thái ban đầu đến nút đó cộng với ước lượng chi phí còn lại.
- Luôn mở rộng nút có giá trị hàm đánh giá thấp nhất cho đến khi tìm thấy đích.

③ **Quy trình:**



④ Mã giả:

Algorithm 5.4.1 A* Algorithm for Pathfinding

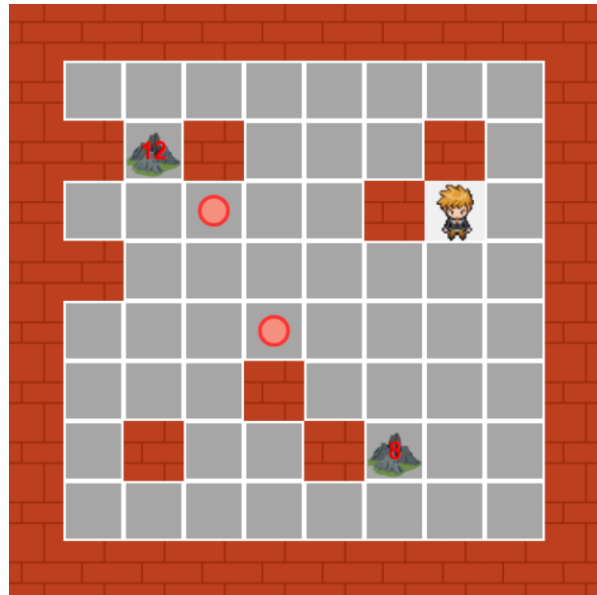
```

1: procedure A_STAR(Start, Goal)
2:   Frontier  $\leftarrow$  priority_queue() ▶  $f(n) = g(n) + \text{heuristic}(n)$ 
3:   Explored  $\leftarrow \emptyset$  ▶ Set of explored nodes
4:    $g(\text{Start}) \leftarrow 0$ 
5:    $f(\text{Start}) \leftarrow g(\text{Start}) + \text{heuristic}(\text{Start})$ 
6:   Frontier.push(Start)
7:   while Frontier is not empty do
8:     Cur  $\leftarrow$  Frontier.pop()
9:     if Cur = Goal then
10:      return Cur ▶ Found the path
11:     Explored.push(Cur)
12:     for each Child in Cur.child() do
13:       newG  $\leftarrow g(\text{Cur}) + \text{cost}(\text{Cur}, \text{Child})$ 
14:       newF  $\leftarrow \text{newG} + \text{heuristic}(\text{Child})$ 
15:       if Child not in Frontier and Child not in Explored then
16:          $g(\text{Child}) \leftarrow \text{newG}$ 
17:          $f(\text{Child}) \leftarrow \text{newF}$ 
18:         Frontier.push(Child)
19:       else if Child in Frontier and newG < g(Child) then
20:          $g(\text{Child}) \leftarrow \text{newG}$ 
21:          $f(\text{Child}) \leftarrow \text{newF}$ 
22:         Frontier.replace(Child)
23:
24:
25:       return failure ▶ No path found
26:

```

6 Kết quả kiểm thử

❑ **Test case:** Sau khi hoàn tất cài đặt các lớp, phương thức đầy đủ, nhóm đã kiểm thử lại kết quả của các thuật toán tìm kiếm với nhiều input khác nhau, với một kiểu định dạng tên gọi: `input-01.txt`, `input-02.txt`, ..., `input-10.txt`. Với phần kiểm thử sẽ giới thiệu sau đây, nhóm xin được sử dụng một map như bên dưới:



	1	2	3	4	5	6	7	8	9	10
BFS	0.75 (s)	0.75 (s)	0.75 (s)	0.75 (s)	0.98 (s)	1.01 (s)	0.77 (s)	0.80 (s)	0.81 (s)	0.90 (s)
DFS	7.03 (s)	7.77 (s)	7.72 (s)	7.67 (s)	7.56 (s)	7.65 (s)	9.74 (s)	9.16 (s)	7.73 (s)	10.01 (s)
UCS	2.27 (s)	2.29 (s)	2.23 (s)	2.32 (s)	2.27 (s)	2.26 (s)	2.39 (s)	2.26 (s)	2.25 (s)	2.29 (s)
A*	0.57 (s)	0.58 (s)	0.56 (s)	0.57 (s)	0.56 (s)	0.56 (s)	0.57 (s)	0.60 (s)	0.59 (s)	0.59 (s)

Bảng. So sánh thời gian thực thi 4 thuật toán Search với 10 lần kiểm thử.

❑ **Nhận xét:**

① A* là nhanh nhất:

- A* sử dụng hàm heuristic để ước lượng chi phí đến mục tiêu, giúp nó tìm được lời giải ngắn hơn.
- Việc sử dụng heuristic giúp A* tập trung tìm kiếm theo hướng đúng, tránh được những nhánh không hứa hẹn, do đó giảm được số nút cần xử lý.

② BFS nhanh hơn UCS:

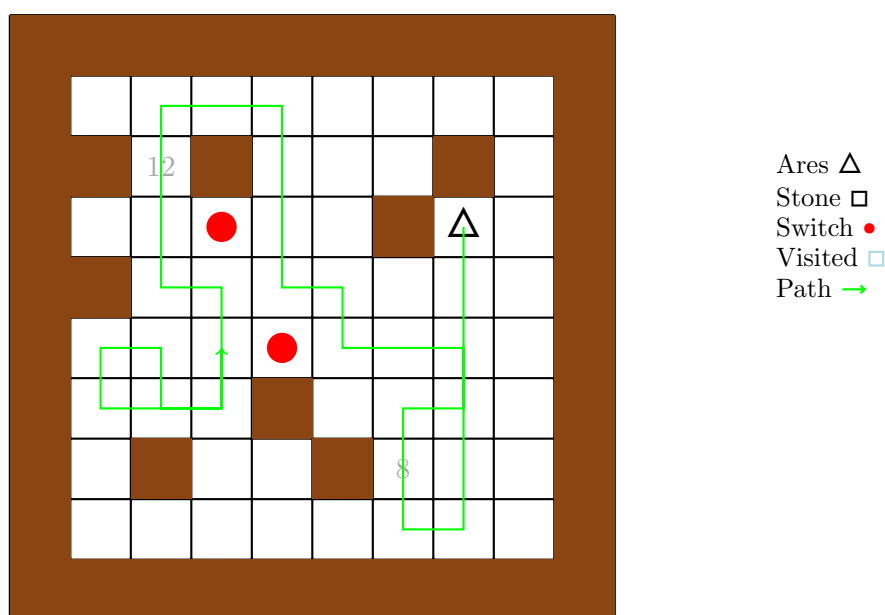
- BFS là thuật toán tìm kiếm theo chiều rộng, nên nó sẽ truy cập các nút theo thứ tự ưu tiên (mức sâu).
- UCS là thuật toán tìm kiếm theo chi phí tối thiểu, nên phải xem xét toàn bộ các nút trong mỗi mức sâu, do đó tốn nhiều thời gian hơn.

③ UCS chậm hơn A* và BFS:

- UCS không sử dụng như A*, nên phải xem xét toàn bộ các nút trong mỗi mức sâu.
- Điều này dẫn đến UCS phải xử lý nhiều nút hơn so với A* và BFS, do đó thời gian chạy sẽ lâu hơn.

④ DFS chậm nhất:

- DFS là thuật toán tìm kiếm theo chiều sâu, nên nó sẽ đi theo từng nhánh sâu nhất có thể trước khi quay trở lại.
- Điều này dẫn đến DFS phải xử lý nhiều nút hơn so với các thuật toán khác, đặc biệt là ở các bước cuối cùng của trò chơi, khi không gian trạng thái lớn.



Hình 6.1: Minh họa cho thuật toán A* với map kiểm thử