BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

CƠ SỞ TRÍ TUỆ NHÂN TẠO

BÁO CÁO ĐỒ ÁN MÔN HỌC

ĐỀ TÀI: LAB 1- SEARCH

Giảng viên hướng dẫn :Lê Nhựt Nam

Lớp :CQ2022/4

Nhóm sinh viên thực hiện :Nguyễn Duy Lâm- 22120181

Huỳnh Tấn Lộc- 22120186

Đinh Viết Lợi- 22120188

Nguyễn Trần Lợi- 22120190

Hồ Chí Minh, ngày 29 tháng 10 năm 2024

MỤC LỤC

PHẦN 1: BÁO CÁO NHÓM

I. Thông tin thành viên

MSSV	Họ và tên	Email	Vai trò chính
22120181	Nguyễn Duy Lâm	22120181@student.hcmus.edu.vn	Lập trình viên
22120186	Huỳnh Tấn Lộc	22120186@student.hcmus.edu.vn	Nhóm phó
22120188	Đinh Viết Lợi	22120188@student.hcmus.edu.vn	Nhóm trưởng
22120190	Nguyễn Trần Lợi	22120190@student.hcmus.edu.vn	Lập trình viên

II. Phân chia công việc

Nội dung công việc	Người phụ trách	Thời gian bắt đầu	Thời gian kết thúc	Kết quả mong muốn	Ghi chú
Viết các hàm bổ trợ giải quyết thuật toán	Tất cả thành viên	15/10	17/10	Hiểu được cơ chế chung để giải quyết vấn đề, tổng quan vấn đề dưới góc nhìn của lập trình. Các hàm bổ trợ chung để các thuật toán ứng dụng.	Mục đích là biết cách ứng dụng thuật toán từ lý thuyết vào thực tế vấn đề.
Thiết kế giao diện /trải nghiệm người dùng	Đinh Viết Lợi	18/10	26/10	Giao diện hỗ trợ người dùng quan sát được cơ chế của trò chơi, kết quả áp dụng từ các thuật toán. Có cơ chế điều hướng hỗ trợ người dùng.	Trực quan kết quả của thuật toán.
Viết thuật toán giải quyết BFS	Huỳnh Tấn Lộc	18/10	27/10	Giải quyết bài toán theo ý tưởng BFS. Đảm bảo chạy đúng trước khi tối ưu tài nguyên.	

Viết thuật toán giải quyết DFS	Huỳnh Tấn Lộc	18/10	27/10	Giải quyết bài toán theo ý tưởng DFS. Đảm bảo chạy đúng trước khi tối ưu tài nguyên.	
Viết thuật toán giải quyết UCS	Nguyễn Duy Lâm	18/10	27/10	Giải quyết bài toán theo ý tưởng UCS. Đảm bảo chạy đúng trước khi tối ưu tài nguyên.	
Viết thuật toán giải quyết A*	Nguyễn Trần Lợi	18/10	27/10	Giải quyết bài toán theo ý tưởng A *. Đảm bảo chạy đúng trước khi tối ưu tài nguyên.	
Thiết kế các màn chơi	Nguyễn Trần Lợi	18/10	19/10	Tạo được lượng test case phù hợp để đánh giá thuật toán. Các test case phải đa dạng về vật thể, độ khó và vị trí vật thể.	
Trực quan hóa so sánh các thuật toán	Nguyễn Trần Lợi	27/10	29/10	Thu thập kết quả từ các tập tin output nhằm vẽ biểu đồ so sánh các thông số của bốn thuật toán thuộc từng test case.	
Quay video giới thiệu chương trình	Nguyễn Duy Lâm Huỳnh Tấn Lộc	27/10	29/10	Quay video báo cáo kết quả đồ án. Video thể hiện được dòng chảy code, UI và quá trình chạy.	Video được phụ đề theo nội dung soạn sẵn
Viết báo cáo nội dung thuật toán tìm kiếm	Huỳnh Tấn Lộc Nguyễn Duy Lâm Nguyễn Trần	28/10	29/10	Báo cáo mô tả nguyên lý của thuật toán phải đảm bảo đủ ý tưởng và quá trình thực thi giải quyết vấn đề theo góc nhìn của các thuật toán tìm kiếm.	Nội dung bao gồm: • Nội dung thuật toán • Ý tưởng áp

	Lợi				dụng vào vấn đề • Trình tự thực thi • Hạn chế của thuật toán
Viết báo cáo tổng hợp	Đinh Viết Lợi	30/10	31/10	Báo cáo tổng hợp đồ án phải thể hiện được toàn bộ quá trình và kết quả làm việc của nhóm Bao gồm các nội dung: báo cáo nhóm, quá trình làm việc, công nghệ sử dụng, nội dung, ý tưởng thuật toán, nội dung mã nguồn, tài liệu tham khảo, phản hồi của từng cá nhân.	

PHẦN 2: TỔNG QUAN ĐỒ ÁN

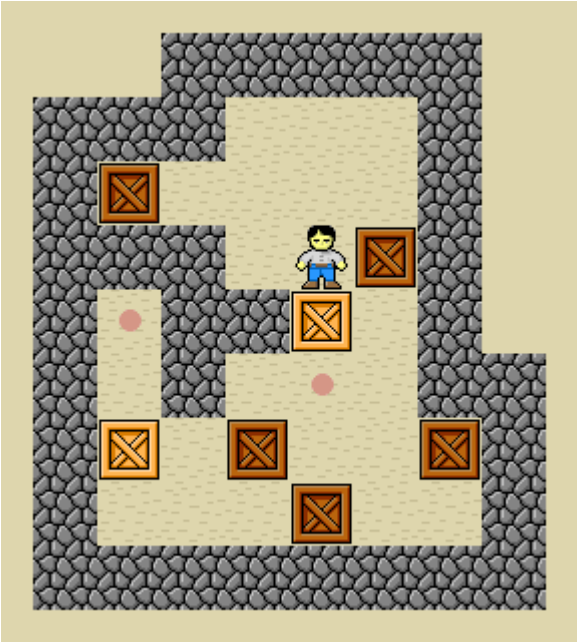
I. Yêu cầu đồ án

a. Giới thiệu đồ án

- Đồ án được thực hiện dựa trên yêu cầu của đồ án Lab 1-Search thuộc lớp "Cơ sở trí tuệ nhân tạo- CQ2022/4" trường đại học Khoa Học Tự Nhiên thuộc Đại học Quốc gia TP.HCM học kỳ I năm học 2024-2025.
- Đồ án nhằm ứng dụng kiến thức lý thuyết của một số thuật toán tìm kiếm phổ biến trong khoa học trí tuệ nhân tạo vào việc giải quyết vấn đề "Tìm đường đi giải quyết thử thách trong trò chơi Ares's Adventure". Sau đây xin được gọi là trò chơi Sokoban và Ares là người chơi.
- Kết quả của đồ án là một chương trình trò chơi gồm đồ họa của một trò chơi sokoban thông thường cùng với các hỗ trợ từ các thuật toán được nhóm phát triển và một video mô tả kết quả. Bốn thuật toán được nhóm ứng dụng vào trò chơi: Breadth-first search(BFS), Deep-frist search(DFS), Uniform-Cost Search (UCS) và A*.
- Cấu trúc của bài báo cáo bao gồm: thông tin về nhóm và quá trình làm việc; mô tả bối cảnh thực hiện; ý tưởng và cách thức áp dụng thuật toán vào giải quyết vấn đề bài toán; hướng dẫn sử dụng; tài liệu tham khảo.

b. Mô tả chi tiết

- **Sokoban** là một trò chơi giải đố kinh điển. Trong trò chơi này, người chơi điều khiển một nhân vật trong một bản đồ 2D, với mục tiêu là di chuyển các khối đá vào các vị trí được đánh dấu sẵn Tuy nhiên, có một số quy tắc và thách thức cụ thể:
  - Bản đồ là một ma trận có kích thước n x m, mỗi ô có thể là không gian trống, tường, đá hoặc điểm đích.
  - Người chơi có thể di chuyển theo 4 hướng lên, xuống, qua phải, qua trái và không thể đi xuyên qua tường hoặc đá.
  - Nếu ô liền chứa đá và ô liền kề đá là không gian trống thì người chơi có thể đẩy đá theo hướng đẩy. Người chơi không được phép kéo, đẩy đá vào tường hoặc đẩy nhiều viên đá cùng lúc.



Hình 1: Trò chơi Sokoban

- Mỗi viên đá có khối lượng riêng, nghĩa là người chơi phải sử dụng chiến thuật hợp lý để tối ưu số bước di chuyển và công sức bỏ ra. Mục đích là đẩy mọi viên đá tới điểm đích, mọi viên đá nằm ở đích đều được ghi nhận không quan trọng khối lượng.
- Trò chơi hỗ trợ người chơi với các cơ chế tự động tìm ra đường đi thông qua 4 thuật toán tìm kiếm : BFS, DFS, UCS và A*.

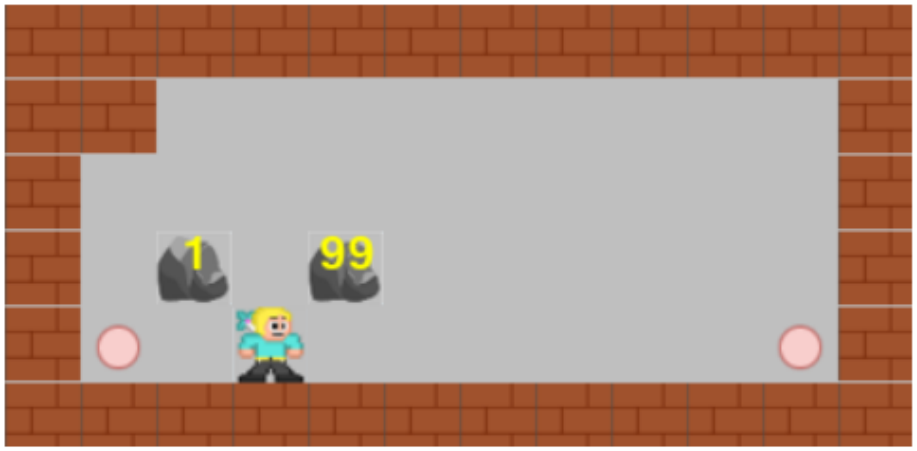
c. Yêu cầu

i. Tập tin đầu vào (Input)

Nhóm chuẩn bị 10 tập tin input theo định dạng input-01.txt, input-02.txt… nhằm chuẩn bị các test case thực hiển kiểm thử hiệu quả của từng thuật toán. Các tập tin này khác nhau ở khối lượng các viên đá, số lượng và vị trí các vật thể của mê cung. Cấu trúc các file này theo định dạng:

- Dòng đầu tiên là danh sách các khối lượng của các viên đá từ trên xuống dưới, từ trái qua phải.

- Các dòng tiếp theo mô tả cấu trúc của mê cung:
  - "#" : tường
  - " " (khoảng trống) : không gian có thể di chuyển đến.
  - "$" : viên đá
  - "@" : Ares
  - "." : điểm đích
  - "*" : đá nằm trùng điểm đích
  - "+" : Ares trùng điểm đích




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

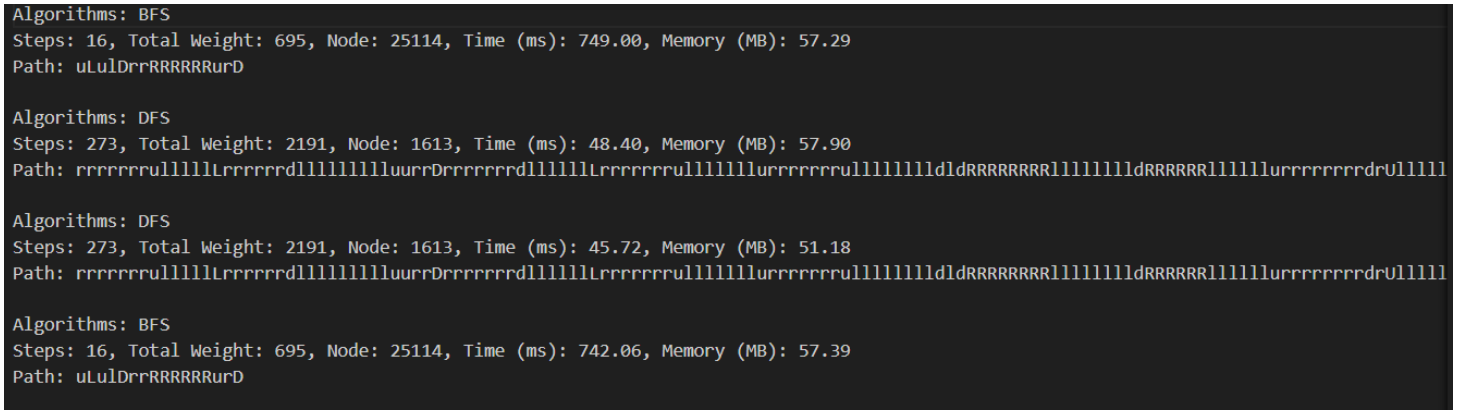
Hình 2: Hình ảnh mê cung và tập tin tương ứng

ii. Tập tin đầu ra (Output)

- Với mỗi màn chơi khác nhau, nhóm chuẩn bị số tập tin output-01.txt, output-02.txt tương ứng… Mỗi khi người dùng chọn thực hiện thuật toán với mỗi màn chơi, các thông số chi tiết của thuật toán đó sẽ được lưu vào file tương ứng.
- Cấu trúc của tập tin đầu ra:
  - Dòng đầu cung cấp tên thuật toán được sử dụng
  - Dòng thứ hai cung cấp các thông số bao gồm:
    - Tổng số bước Ares di chuyển.
    - Tổng khối lượng đá Ares đã đẩy.
    - Tổng số node được tạo ra bởi thuật toán.
    - Thời gian thực thi tìm kiếm.
    - Tổng dung lượng cần thiết cho quá trình tìm kiếm.
  - Dòng thứ ba mô tả trình tự di chuyển được thuật toán đề xuất với uldr là

bước di chuyển và ULDR là bước đẩy đá.

- Ví dụ:
  - Algorithms: BFS
  - Steps: 16, Total Weight: 695, Node: 25114, Time (ms): 1173.32, Memory (MB): 65.27
  - Path: uLulDrrRRRRRRurD



Hình 3: Cấu trúc tập tin đầu ra

iii. Giao diện đồ họa (GUI)

- Nhóm xây dựng giao diện đồ họa hỗ trợ cho trò chơi trở nên thú vị, bắt mắt đồng thời giúp trực quan kết quả của các thuật toán:
  - Giao diện cung cấp sự tồn tại của các tất cả vật thể trong mê cung.
  - Giao diện thể hiện từng bước di chuyển của Ares suốt quá trình di chuyển thông thường và đẩy đá.
  - Hỗ trợ thống kê quá trình thử thách thông qua số bước đi và tổng trọng lượng đá đã di chuyển.
  - Cung cấp một số điều hướng hỗ trợ người dùng tùy chỉnh các lựa chọn như lựa chọn thuật toán, chơi lại, dừng hoạt ảnh di chuyển….

Hình 4: Giao diện đồ họa

II. Quy trình thực hiện

- Giai đoạn chuẩn bị là khoảng thời gian nhóm tổ chức tập thể, tổng quan vấn đề, chuẩn bị tài liệu và lên kế hoạch thực hiện :
  - Nhóm tổ chức các kênh thông tin liên lạc, kho lưu trữ tài liệu, kết quả làm việc và ứng dụng quản lý phiên bản.
  - Họp nhóm thông báo các thông tin của đồ án, lên kế hoạch tạm thời và tìm hiểu rõ yêu cầu đồ án cũng như ý tưởng thực thi.
- Giai đoạn thực thi:
  - Nhóm trước tiên làm rõ yêu cầu cơ bản của chương trình chính sẽ gồm hai phần là giao diện đồ họa và tập trung vào mã nguồn thực thi thuật toán.
  - Đối với giao diện đồ họa, nhóm tham khảo một số giao diện phổ biến được công bố trên các tài liệu, website, trò chơi để vẽ bản thiết kế. Kết hợp với các yêu cầu có ở GUI.
  - Đối với các thuật toán, nhóm cho rằng tiên quyết cần nắm rõ ý tưởng của các thuật toán rồi từ đó ứng dụng vào việc giải quyết bài toán sokoban.
  - Vẽ thiết kế tổng quát của chương trình, các hàm chức năng, biến dữ liệu, hình

thức lưu trữ, hướng đi của luồng dữ liệu…

- Tham khảo các video hướng dẫn, kết quả làm việc của các tập thể khác trên các nguồn thông tin nhằm tinh chỉnh bản thiết kế. Đánh giá ưu, khuyết điểm của các chương trình này để cải thiện mã nguồn cho phù hợp.
- Kiểm thử kết quả của các chương trình, đánh giá ưu điểm, hạn chế của thuật toán, ghi lại các kết quả.

- Giai đoạn tổng kết:
  - Sau khi đảm bảo thuật toán trả về kết quả đúng, tiếp tục cải thiện tính tối ưu của thuật toán nhằm giảm tài nguyên của máy tính.
  - Thu thập các kết quả từ tập tin output, trực quan hóa các kết quả này thành các biểu đồ phù hợp để so sánh các thuật toán trong các trường hợp khác nhau.
  - Tinh chỉnh giao diện đồ họa phù hợp với người chơi, kiểm thử với các mê cung có kích thước lớn và đa dạng hơn.
  - Viết báo cáo tổng hợp đồ án.
  - Lên kịch bản quay video trình bày sản phẩm và thực hiện video.

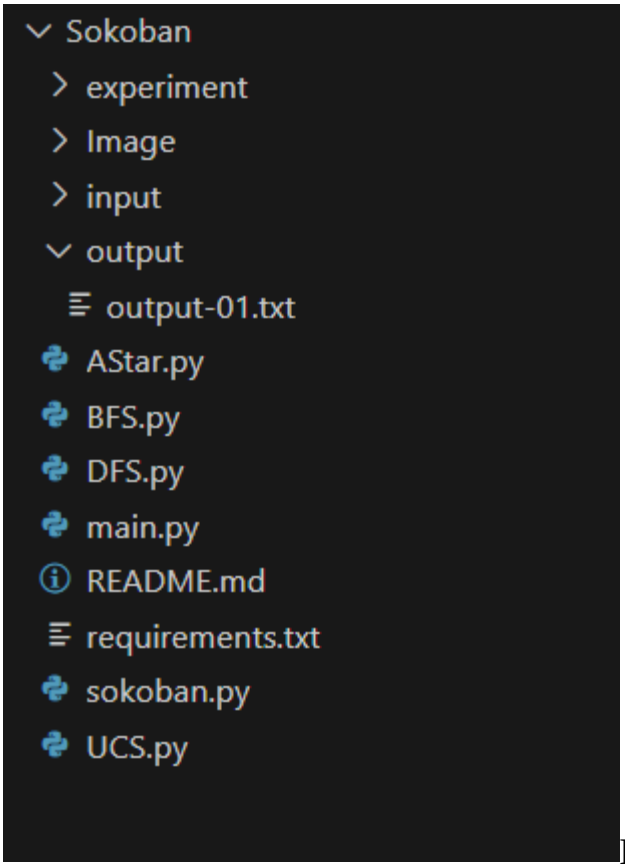
III. Đánh giá yêu cầu

- Tổng quan mức độ hoàn thành đồ án: Khá tốt (85-92%)
- Nhóm hoàn thành đồ án trong thời gian quy định.
- Các yêu cầu được nêu bên trên được hoàn thành đầy đủ:
  - Nhóm cung cấp 10 test case hỗ theo đúng định dạng quy định.Các tập tin được đặt tên theo cấu trúc input-01.txt…
  - 10 tập tin output đúng định dạng cung cấp kết quả thực thi 4 thuật toán cho mỗi màn chơi với đủ số lượng quy định.
  - Giao diện đồ họa cung cấp đầy đủ điều hướng hỗ trợ người chơi và thể hiện đầy đủ quá trình thực hiện thử thách. Giao diện dễ quan sát, gần gũi với người chơi
- Quá trình làm việc nhóm diễn ra thuận lợi, nhóm thực hiện đúng kế hoạch đã đề ra trong quy trình phát triển và phân công công việc; các vấn đề phát sinh đồng thời được giải quyết triệt để.
- Nhóm sử dụng ngôn ngữ python để thực hiện đồ án.
- Báo cáo thể hiện đầy đủ được quá trình và kết quả thực hiện đồ án.
- Có video trình bày đầy đủ các yêu cầu được giao.
- **Hạn chế**: do thời gian có hạn, chương trình có nhiều điểm chưa được tối ưu, một số thuật toán trong vài trường hợp tiêu tốn rất nhiều tài nguyên máy tính. Giao diện đồng thời chưa thể hiện tất cả thông báo hỗ trợ người dùng. Video trình bày thiếu chuyên nghiệp và chưa gây được chú ý cho người nghe.

PHẦN 3: CHI TIẾT CHƯƠNG TRÌNH

I. Thư mục và tập tin

- Thư mục image: lưu trữ các ảnh sử dụng cho công đoạn thiết kế giao diện đồ họa
- Thư mục input: lưu trữ các test case được nhóm thiết kế cho người chơi hoặc được sử dụng để thử nghiệm kết quả của các thuật toán tìm kiếm.
- Thư mục output: lưu trữ các tập tin mang nội dung là kết quả và tài nguyên thực thi của các thuật toán.
- Thư mục experiment: lưu trữ các kết quả đã thực thi của 4 thuật toán trên 10 tập tin input thành 10 file output tương ứng. Hỗ trợ người dùng tiết kiệm thời gian chạy trực tiếp thuật toán.
- Tập tin main.py: chứa lệnh kiểm tra if _name__== “__main__” , đây là tập tin đầu tiên chạy khi thực thi chương trình, chứa các câu lệnh liên quan đến giao diện và trải nghiệm người dùng.
- Tập tin sokoban.py: chứa các hàm hỗ trợ và sử dụng chung cho các thuật toán tìm kiếm.
- Tập tin BFS.py, UCS.py, DFS.py, AStar.py : các tập tin chứa mã nguồn nội dung thuật toán tương ứng, thực thi việc tìm kiếm, lưu kết quả vào file output và trả về đường đi theo thuật toán.
- Tập tin README.md: Tập tin giới thiệu về dự án và giải thích nội dung dự án hỗ trợ mọi người cách cài đặt và sử dụng mã nguồn.
- Requirements.txt: ghi chú các thư viện cần cài đặt để chạy chương trình.



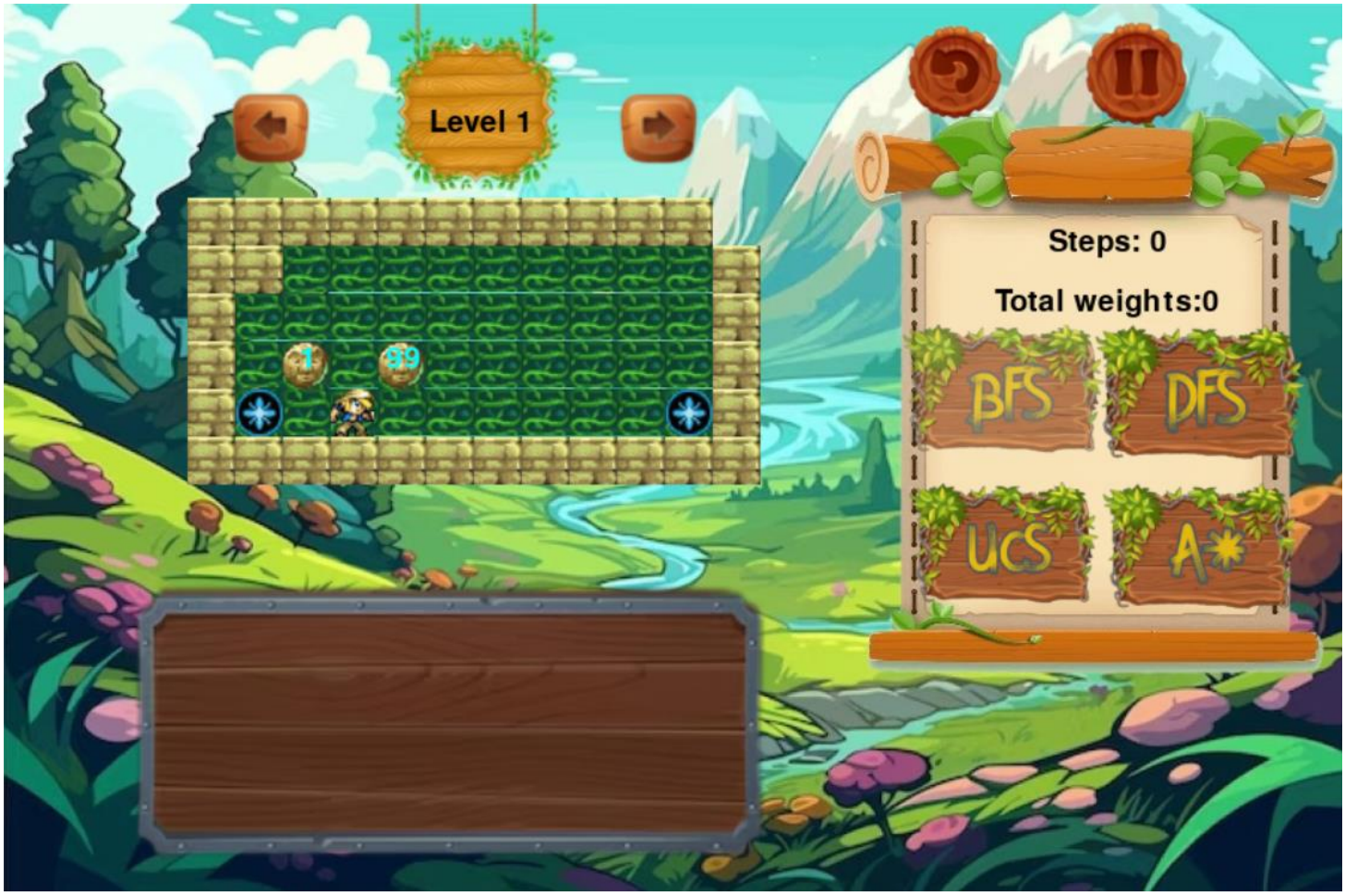
Hình 5: *Cấu trúc tập tin thư mục*

II. Giao diện đồ họa

- Đồ họa của trò chơi được thiết kế bằng thư viện pygame, đây là một thư viện rất phổ biến trong việc thiết kế trò chơi 2D.
- Trước tiên, ta sử dụng các biến để lưu trữ màu dùng lại về sau mà không cần phải nhập mã RBG thủ công.
- Các hình ảnh sử dụng được lưu vào một cấu trúc dữ liệu dict images {tên ảnh : đường dẫn ảnh}.
- Các biến toàn cục được sử dụng:
  - Board: ma trận hiện hành
  - map_directory: tên thư mục lưu trữ input
  -
  - map_filepath: danh sách các thư mục input
  - selected_level: màn chơi hiện tại
  - step_count : số bước di chuyển của Ares
  - instruct_step: bước di chuyển trả về bởi thuật toán
  - player_x, player_y: tọa độ Ares
  - check_point_list: danh sách các tọa độ của check point
  - stone_point_list: danh sách các tọa độ đá

  - stone_weights: danh sách trọng lượng các khối đá
  - stone_weights_dict: dict lưu { tọa độ: khối lượng tương ứng} của các khối đá
  - total_weights: tổng khối lượng đá di chuyển
  - stop_moving: biến xác định có được phép tiếp tục di chuyển theo chỉ dẫn.
- read_map: hàm này đọc cấu trúc của một file input để lưu vào một ma trận có các giá trị tương ứng với file input, dòng đầu tiên sẽ lưu vào lần lượt các khối lượng đá vào stone_weights.
- Change_level: hàm này lấy tham số đầu vào là một số nguyên để thực hiện việc chuyển level chơi bằng cách thay đổi giá trị selected level. Từ đó lựa chọn index tương ứng trong map_filepath để readmap và thay đổi board. Hàm này sẽ làm mới lại tất cả giá trị của tất cả biến toàn cục.
- Find_position: hàm này tìm tọa độ của tất cả giá trị trong ma trận hiện hành giống với tham số truyền vào và trả về tập vị trí của các phần tử này.
- Draw_button : hàm này vẽ hình ảnh của một nút bấm (button) lên trên màn hình với tham số hình ảnh và tọa độ truyền vào.
- Draw_interface: hàm này vẽ toàn bộ giao diện của màn hình ngoại trừ ma trận, bao gồm các nội dung (text) của các nút bấm, cấp độ đang chơi (selected_level), số bước di chuyển, tổng khối lượng đá đã đẩy và các nút bấm.
- Render_map: hàm này đọc nội dung ma trận (board) và vẽ lên giao diện hiện tại của trò chơi. Đối với các viên đá, ta so sánh tọa độ của viên đá trong stone_weights_dict và lấy ra giá trị tương ứng để hiển thị lên vị trí viên đá.
- Check_for_completion: hàm này thực hiện việc kiểm tra ma trận, nếu tọa độ của tất cả các check_point đều chứa giá trị * thì xác định đã hoàn thành màn chơi.
- Move_player: hàm này chịu trách nhiệm **cập nhật giá trị ma trận** mỗi khi Ares thực hiện di chuyển. Hàm này được truyền vào các giá trị “LEFT”, “RIGHT”, “UP”, “DOWN” , ta xác định tọa độ mới của player_x, player_y dựa trên giá trị này.
  - Nếu là khoảng trống hoặc điểm đích (check_point) thì Ares di chuyển vào, cập nhật giá trị tương ứng cho ma trận.
  - Nếu tọa độ mới của Ares là viên đá, ta kiểm tra xem viên đá có thể đẩy được không bằng cách kiểm tra tọa độ mới của viên đá. Nếu khả thi, ta cập nhật các giá trị trên ma trận, cập nhật giá trị tọa độ mới của viên đá trong stone_weights_dict, cập nhật vị trí Ares và kiểm tra đã hoàn thành trò chơi chưa.
  - Kiểm tra tọa độ các điểm đích, nếu là khoảng trống thì cập nhật giá trị thành điểm đích trong ma trận.
- Trong quá trình chạy, ta sư dụng pygame.event để theo dõi các tương tác của người dùng thông qua chuột và bàn phím để thực hiện các hàm tương ứng.
  - Người dùng sử dụng phím di chuyển để thực hiện hàm move_player

- Người dùng sử dụng các nút bấm chuyển level, BFS, DFS, UCS, A*, chơi lại, dừng di chuyên để thực hiện các hàm hỗ trợ. Đối với các nút bấm thuật toán, chương trình sẽ xin kết quả trả về bởi các thuật toán là các lộ trình để thực hiện hàm start_move_on_instruct. Hàm này tạo một luồng di chuyển và thực hiện vòng lặp move_player theo instruct_steps.



Hình 6: Một số điều hướng của chương trình

III. Hàm bổ trợ

- transferToGameState(layout)
- Hàm này chuyển đổi một bố cục câu đố (dạng chuỗi) thành một trạng thái trò chơi có thể xử lý. Nó thay thế các ký tự đại diện cho các thành phần của trò chơi (tường, Ares, đá, mục tiêu, v.v.) thành các số nguyên để dễ dàng thao tác trong mã. Ngoài ra, nó điều chỉnh kích thước các hàng sao cho chúng có cùng số cột bằng cách thêm các số 0 (khoảng trắng) vào các hàng ngắn hơn. Kết quả cuối cùng là một mảng NumPy chứa trạng thái trò chơi.
- PosOfPlayer(gameState)
- Hàm này tìm vị trí của Ares trên lưới.
- PosOfStones(gameState, weights)
- Hàm này xác định vị trí của các đá trên lưới và liên kết mỗi vị trí đá với trọng số

tương ứng trong danh sách weights. Nếu weights ngắn hơn số đá, các đá còn lại sẽ được gán trọng số mặc định là 1.

- PosOfWalls(gameState)
- Hàm này tìm vị trí của các tường và trả về danh sách các vị trí đó.
- PosOfGoals(gameState)
- Hàm này tìm vị trí của các mục tiêu, bao gồm các ô chứa đá hoặc Ares đặt trên mục tiêu. Nó trả về các vị trí này dưới dạng một danh sách.
- isEndState(posStone)
- Hàm này kiểm tra xem tất cả các đá đã nằm trên các vị trí mục tiêu chưa. Nếu đúng, trò chơi đã thắng.
- isLegalAction(action, posPlayer, posStone)
- Hàm này kiểm tra xem một hành động di chuyển có hợp lệ hay không. Nếu hành động là đẩy, nó kiểm tra xem vị trí mới của đá có trống không. Nếu chỉ là di chuyển, nó kiểm tra xem Ares có va chạm vào tường hoặc đá không.
- legalActions(posPlayer, posStone)
- Hàm này trả về tất cả các hành động di chuyển hợp lệ của Ares, bao gồm di chuyển bình thường hoặc đẩy đá nếu hợp lệ. Mỗi hành động được kiểm tra tính hợp lệ trước khi đưa vào danh sách kết quả.
- updateState(posPlayer, posStone, action)
- Hàm này cập nhật vị trí của Ares và các đá khi thực hiện một hành động. Nếu là hành động đẩy, nó sẽ cập nhật vị trí mới của đá bị đẩy. Kết quả trả về vị trí mới của Ares và các vị trí đá sau khi thực hiện hành động.
- isFailed(posStone)
- Hàm này kiểm tra xem trạng thái hiện tại có thể dẫn đến thất bại không. Nó dùng các mẫu để kiểm tra các vị trí "chết" (các vị trí mà nếu đá nằm ở đó, trò chơi sẽ không thể hoàn thành). Nếu phát hiện vị trí này, hàm trả về True, biểu thị rằng trạng thái này nên được loại bỏ khỏi tìm kiếm.

IV. Thuật toán BFS

a. Nội dung thuật toán

- Thuật toán tìm kiếm theo chiều rộng (BFS) là một phương pháp tìm kiếm trong đồ thị hoặc cây, trong đó nó khám phá tất cả các node ở cùng một cấp độ trước khi di chuyển xuống cấp độ tiếp theo. BFS thường được sử dụng trong các bài toán như tìm kiếm đường đi, kiểm tra tính liên thông, và nhiều ứng dụng khác.
- Các bước cơ bản của thuật toán BFS:
  - Khởi tạo:

- Bắt đầu từ node gốc: Xác định node khởi đầu của đồ thị mà bạn muốn khám phá.
- Thêm node gốc vào hàng đợi (queue): Đưa node gốc vào hàng đợi để chuẩn bị cho quá trình khám phá.
- Đánh dấu node gốc là đã khám phá: Sử dụng một tập hợp (set) để lưu trữ các node đã được khám phá nhằm tránh việc thăm lại chúng.

o Lặp lại quá trình tìm kiếm, trong khi hàng đợi không rỗng:
- Lấy node ở đầu hàng đợi: Lấy node đầu tiên ra khỏi hàng đợi để mở rộng.
- Kiểm tra xem node đó có phải là node mục tiêu hay không:
- Nếu node hiện tại là node mục tiêu, thuật toán kết thúc và trả về đường đi.
- Nếu không, tiến hành thêm các node con của node hiện tại vào hàng đợi.
- Thêm các node con vào hàng đợi: Duyệt qua tất cả các node láng giềng (neighbors) của node hiện tại.
- Đánh dấu các node con là đã khám phá: Đảm bảo rằng không thăm lại các node đã được khám phá trước đó.

o Quay lại (Backtracking),nếu node hiện tại không có node con nào chưa được khám phá:
- Trong thuật toán BFS, không có thao tác quay lại như trong DFS, vì BFS sử dụng hàng đợi để khám phá từng cấp độ một.
- Nếu tất cả các node đã được thăm và không tìm thấy node mục tiêu, thuật toán sẽ kết thúc mà không cần phải quay lại node cha.

b. Ý tưởng vận hành trong sokoban

- **Khởi tạo:**
  - Bắt đầu với trạng thái gốc (bao gồm vị trí của Ares và vị trí của các đá) và đưa vào hàng đợi (queue) để xử lý.
  - Đánh dấu trạng thái gốc là đã khám phá để tránh việc xử lý lại.
- **Lặp lại quá trình tìm kiếm:**
  - Trong mỗi bước, lấy trạng thái ở đỉnh hàng đợi để kiểm tra.
  - Nếu trạng thái đó là trạng thái mục tiêu (tất cả các đá đã được đẩy vào vị trí mục tiêu), thuật toán sẽ trả về đường đi từ trạng thái gốc đến trạng thái mục tiêu.
  - Nếu không, tiếp tục mở rộng các trạng thái con từ trạng thái hiện tại bằng cách kiểm tra tất cả các hành động hợp lệ mà Ares có thể thực hiện.
- **Tránh trạng thái lặp:** Sử dụng một tập hợp (set) để lưu trữ các trạng thái đã khám phá nhằm tránh việc mở rộng lại những trạng thái này, từ đó giảm thiểu số nút cần xử lý.
- **Duy trì đường đi:** Lưu lại các hành động hoặc trạng thái trên đường đi từ trạng thái gốc đến trạng thái mục tiêu để có thể trả về kết quả khi cần, giúp Ares biết được các bước cần thực hiện để giải quyết bài toán.

c. Giải thích chi tiết thuật toán

- **Khởi tạo**

- Bắt đầu từ Node gốc:
  - BFS bắt đầu từ một node gốc trong đồ thị hoặc cây.
  - Node gốc sẽ được đưa vào một hàng đợi (queue) để theo dõi các node cần được khám phá.
- Đánh dấu là đã khám phá: Node gốc cũng được đánh dấu là đã được khám phá để tránh việc mở rộng lại trong tương lai. Điều này giúp đảm bảo rằng thuật toán không quay lại những node đã kiểm tra trước đó.

- **Lặp lại quá trình tìm kiếm**
  - Khi nào để lặp lại: Thuật toán sẽ tiếp tục thực hiện quá trình tìm kiếm cho đến khi hàng đợi rỗng (không còn node nào để khám phá) hoặc khi node mục tiêu được tìm thấy.
  - Lấy Node ở Đầu Hàng Đợi: Tại mỗi bước, thuật toán lấy node ở đầu hàng đợi để kiểm tra. Điều này có nghĩa là BFS sẽ mở rộng tất cả các node ở cùng một mức trước khi chuyển sang mức tiếp theo.
  - Kiểm Tra Node Mục Tiêu: Sau khi lấy node từ hàng đợi, thuật toán kiểm tra xem node đó có phải là mục tiêu hay không:
    - Nếu là mục tiêu: Thuật toán kết thúc và trả về đường đi từ node gốc đến node mục tiêu.
    - Nếu không phải: Tiếp tục mở rộng node này bằng cách tìm các node con.
- **Mở rộng Node:** Thêm Node Con vào Hàng Đợi:
  - Các node con (những node liền kề) của node hiện tại sẽ được thêm vào hàng đợi.
  - Mỗi node con cũng được đánh dấu là đã khám phá để tránh lặp lại. Điều này đảm bảo rằng tất cả các node ở cùng một mức được kiểm tra trước khi tiến tới mức tiếp theo.
- **Tránh Trạng Thái Lặp:** Sử dụng Tập Hợp để Lưu Trữ Node Đã Khám Phá:
  - Để tránh việc lặp lại các node đã được khám phá, BFS sử dụng một tập hợp (set) để lưu trữ các node này.
  - Nếu một node con đã được khám phá, nó sẽ không được thêm vào hàng đợi một lần nữa, giúp giảm thiểu số lượng node cần kiểm tra.
- **Duy trì Đường Đi:** Lưu Lại Đường Đi Từ Node Gốc Đến Node Mục Tiêu:
  - BFS có thể lưu lại các node trên đường đi từ node gốc đến node mục tiêu.
  - Điều này cho phép thuật toán trả về kết quả cuối cùng dễ dàng, cung cấp cho người dùng thông tin về cách mà thuật toán đã tìm thấy đường đi.
- **Kết thúc thuật toán:** Khi tìm thấy node mục tiêu, thuật toán sẽ trả về chuỗi các node đã qua để đến trạng thái này. Nếu hàng đợi trở nên rỗng và không tìm thấy node mục tiêu, điều này có nghĩa là không có giải pháp cho bài toán và thuật toán sẽ kết thúc mà không trả về kết quả.

d. Đánh giá thuật toán

- **Ưu điểm của BFS:**
  - **Tìm Đường Đi Ngắn Nhất:**

    - BFS đảm bảo tìm được đường đi ngắn nhất từ node gốc đến node mục tiêu trong đồ thị không có trọng số. Điều này rất quan trọng trong các bài toán như tìm đường trong mạng lưới.
  - **Khám Phá Đầy Đủ:**
    - BFS khám phá tất cả các node ở mức hiện tại trước khi chuyển sang mức tiếp theo, giúp phát hiện tất cả các giải pháp có thể có ở mỗi mức.
  - **Khả Năng Thích Ứng Cao:**
    - BFS có thể áp dụng cho nhiều loại bài toán khác nhau, bao gồm tìm kiếm trong đồ thị, lập kế hoạch, và các trò chơi chiến lược.
- **Nhược điểm của BFS:**
  - **Tốn Bộ Nhớ**: BFS có thể tiêu tốn một lượng lớn bộ nhớ, đặc biệt khi tìm kiếm trong các đồ thị lớn hoặc không có giới hạn. Điều này do việc lưu trữ tất cả các node con trong hàng đợi và đánh dấu các node đã khám phá.
  - **Tốc Độ Chậm trong Các Đồ Thị Rộng:** Trong những đồ thị rất rộng (nhiều nhánh), BFS có thể chậm hơn so với các thuật toán tìm kiếm khác như DFS, vì BFS cần khám phá tất cả các node ở mức trước khi tiến lên.
  - **Không Thích Hợp cho Đồ Thị Có Trọng Số:** Nếu đồ thị có trọng số, BFS không đảm bảo tìm ra đường đi ngắn nhất, vì nó không xem xét trọng số của các cạnh. Thay vào đó, các thuật toán như Dijkstra hoặc A* sẽ được ưu tiên sử dụng.

V. Thuật toán DFS

a. Nội dung thuật toán

- Thuật toán tìm kiếm theo chiều sâu (DFS) là một phương pháp tìm kiếm trong đồ thị hoặc cây, nơi nó đi sâu vào các nhánh của đồ thị trước khi quay lại để tìm kiếm các nhánh khác. DFS thường được sử dụng để giải quyết các bài toán như tìm kiếm đường đi, tìm kiếm chu trình, và nhiều bài toán khác.
- Các bước cơ bản của thuật toán DFS:
  - Khởi tạo:
    - Bắt đầu từ node gốc và đưa nó vào một ngăn xếp (stack).
    - Đánh dấu node gốc là đã khám phá.
  - Lặp lại quá trình tìm kiếm: Trong khi ngăn xếp không rỗng, thực hiện các bước sau:
    - Lấy node ở đỉnh ngăn xếp ra để mở rộng.
    - Kiểm tra xem node đó có phải là node mục tiêu hay không:
      - Nếu có, thuật toán kết thúc và trả về đường đi.
      - Nếu không, thêm tất cả các node con của node hiện tại vào ngăn xếp và đánh dấu chúng là đã khám phá.
  - Quay lại (Backtracking): Nếu node hiện tại không có node con nào chưa được khám phá, thuật toán quay lại node cha để tiếp tục khám phá.

b. Ý tưởng vận hành trong sokoban

- Khởi tạo: Bắt đầu với node gốc, đưa vào ngăn xếp và đánh dấu là đã khám phá.
- Lặp lại quá trình tìm kiếm:
  - Trong mỗi bước, lấy node ở đỉnh ngăn xếp để kiểm tra.
  - Nếu node đó là mục tiêu, thuật toán sẽ trả về kết quả.
  - Nếu không, tiếp tục mở rộng các node con và thêm vào ngăn xếp.
- Tránh trạng thái lặp: Sử dụng một tập hợp để lưu trữ các node đã khám phá nhằm tránh việc mở rộng lại những node này.
- Duy trì đường đi: Lưu lại các node trên đường đi từ node gốc đến node mục tiêu để có thể trả về kết quả khi cần.

c. Giải thích chi tiết thuật toán

- Khởi tạo: Bắt đầu từ Node Gốc:
  - DFS bắt đầu từ một node gốc trong đồ thị hoặc cây.
  - Node gốc sẽ được đưa vào một ngăn xếp (stack) để theo dõi các node cần được khám phá.
  - Node gốc cũng được đánh dấu là đã được khám phá để tránh việc mở rộng lại trong tương lai.
- Lặp lại quá trình tìm kiếm
  - Khi nào để lặp lại: Thuật toán sẽ tiếp tục thực hiện quá trình tìm kiếm cho đến khi ngăn xếp rỗng (không còn node nào để khám phá) hoặc khi node mục tiêu được tìm thấy.
  - Lấy Node ở Đỉnh Ngăn Xếp: Tại mỗi bước, thuật toán lấy node ở đỉnh ngăn xếp để kiểm tra. Điều này có nghĩa là DFS sẽ luôn mở rộng node mà nó vừa thêm vào gần nhất.
  - Kiểm Tra Node Mục Tiêu: Sau khi lấy node từ ngăn xếp, thuật toán kiểm tra xem node đó có phải là mục tiêu hay không:
    - Nếu là mục tiêu: Thuật toán kết thúc và trả về đường đi từ node gốc đến node mục tiêu.
    - Nếu không phải: Tiếp tục mở rộng node này bằng cách tìm các node con.
- Mở rộng Node: Thêm Node Con vào Ngăn Xếp:
  - Các node con (những node liền kề) của node hiện tại sẽ được thêm vào ngăn xếp.
  - Mỗi node con cũng được đánh dấu là đã khám phá để tránh lặp lại.
- Quay lại (Backtracking): Quay lại khi không còn Node Con:
  - Nếu node hiện tại không còn node con nào để khám phá (tất cả đã được khám phá hoặc không hợp lệ), thuật toán sẽ quay lại node cha.
  - Việc quay lại cho phép thuật toán tiếp tục tìm kiếm ở các nhánh khác chưa được khám phá.
- Tránh Trạng Thái Lặp: Sử dụng Tập Hợp để Lưu Trữ Node Đã Khám Phá:

- Để tránh việc lặp lại các node đã được khám phá, DFS sử dụng một tập hợp (set) để lưu trữ các node này.
- Nếu một node con đã được khám phá, nó sẽ không được thêm vào ngăn xếp một lần nữa.

- Duy trì Đường Đi: Lưu Lại Đường Đi Từ Node Gốc Đến Node Mục Tiêu:
  - DFS có thể lưu lại các node trên đường đi từ node gốc đến node mục tiêu.
  - Điều này cho phép thuật toán trả về kết quả cuối cùng dễ dàng.

d. Đánh giá thuật toán

- Ưu điểm:
  - Khám phá các nhánh sâu: DFS hiệu quả trong các bài toán mà người dùng muốn khám phá một nhánh sâu trước khi quay lại.
  - Ít bộ nhớ: Trong một số trường hợp, DFS có thể sử dụng ít bộ nhớ hơn so với các thuật toán khác (như BFS), đặc biệt là khi đồ thị có nhiều nhánh nhưng ít chiều sâu.
- Nhược điểm:
  - Không đảm bảo tìm ra giải pháp tối ưu: DFS không tìm kiếm theo cách tối ưu và có thể không tìm thấy đường đi ngắn nhất.
  - Có thể gặp tình trạng bế tắc: Nếu không có cơ chế kiểm tra, DFS có thể rơi vào chu trình vô hạn, ví dụ khi có các vòng lặp trong đồ thị.

VI. Thuật toán UCS

a. Nội dung thuật toán

- Thuật toán Tìm kiếm Chi phí Thống nhất (UCS) là một thuật toán tìm kiếm đường đi trên đồ thị, trong đó mỗi cạnh có một chi phí khác nhau. UCS là một biến thể của thuật toán Tìm kiếm theo Chiều rộng (BFS) nhưng thay vì tìm kiếm dựa trên độ sâu của các node, UCS tìm kiếm dựa trên chi phí đường đi từ node gốc đến mỗi node đích.
- Các bước cơ bản của thuật toán UCS:
  - Khởi tạo một hàng đợi ưu tiên (priority queue) chứa node gốc với chi phí bằng 0.
  - Tại mỗi bước, lấy ra node có chi phí thấp nhất từ hàng đợi để mở rộng.
  - Kiểm tra nếu node đó là đích, thuật toán dừng và trả về đường đi.
  - Nếu không phải đích, thêm các node con của node hiện tại vào hàng đợi với chi phí là tổng chi phí từ gốc đến node con.
  - Các node đã mở rộng (explored nodes) được đánh dấu để tránh lặp lại.
  - UCS đảm bảo tìm thấy đường đi tối ưu với chi phí thấp nhất từ node gốc đến node đích, nếu có tồn tại.

b. Ý tưởng vận hành trong sokoban

- Trong trò chơi Sokoban, Ares cần di chuyển các khối đá (Stones) vào các vị trí đích

(targets). UCS có thể được áp dụng để tìm đường đi ngắn nhất hoặc ít chi phí nhất, từ vị trí ban đầu đến khi tất cả các khối đá đã được đặt đúng chỗ.

- Ứng dụng của UCS trong Sokoban:
  - Mỗi trạng thái của trò chơi (cấu hình của vị trí Ares và các khối đá) là một node trong đồ thị.
  - Các phép di chuyển khả thi (lên, xuống, trái, phải) từ một trạng thái sẽ tạo ra các node con.
  - Chi phí di chuyển thường tính bằng số bước đi, nhưng cũng có thể áp dụng các quy tắc chi phí khác.
  - Mục tiêu là tìm đường đi với chi phí thấp nhất mà dẫn đến trạng thái chiến thắng (tất cả khối đá ở vị trí đích).
  - UCS sẽ duy trì hàng đợi ưu tiên của các trạng thái với chi phí thấp nhất và liên tục mở rộng các trạng thái này cho đến khi đạt được trạng thái mục tiêu.

c. Giải thích chi tiết thuật toán

- Khởi tạo hàng đợi ưu tiên (priority queue):
  - Bắt đầu với trạng thái ban đầu của Sokoban (vị trí của Ares và các khối đá) với chi phí là 0.
  - Đưa trạng thái này vào hàng đợi ưu tiên.
- Lặp lại quá trình tìm kiếm:
  - Tại mỗi bước, lấy trạng thái có chi phí thấp nhất từ hàng đợi.
  - Kiểm tra xem trạng thái này có phải là trạng thái chiến thắng (tất cả khối đá đều ở vị trí đích) hay không:
    - Nếu có, thuật toán kết thúc và trả về đường đi với chi phí thấp nhất.
    - Nếu không, thuật toán tiếp tục
  - Nếu chưa đạt đích, tiếp tục mở rộng trạng thái hiện tại bằng cách thử các phép di chuyển (lên, xuống, trái, phải).
- Tính chi phí di chuyển:
  - Mỗi lần di chuyển, tính chi phí của trạng thái mới (ví dụ, tăng chi phí thêm 1 cho mỗi bước hoặc tuỳ quy định chi phí cụ thể).
  - Đưa trạng thái mới vào hàng đợi với chi phí tương ứng nếu trạng thái đó chưa được mở rộng hoặc có chi phí thấp hơn chi phí trước đó.
- Tránh trạng thái lặp:
  - Sử dụng một tập hợp các trạng thái đã mở rộng để không mở lại các trạng thái đã đi qua.
  - Nếu trạng thái mới đã từng được mở rộng, chỉ chấp nhận lại nếu có chi phí thấp

hơn chi phí đã lưu trước đó.

- Duy trì đường đi tối ưu: Đảm bảo rằng thuật toán sẽ chọn các trạng thái với chi phí thấp nhất để mở rộng, từ đó tìm ra đường đi tối ưu từ trạng thái ban đầu đến trạng thái chiến thắng.

d. Đánh giá thuật toán

- Ưu điểm:
  - Tìm kiếm giải pháp tối ưu: UCS giúp đảm bảo tìm được đường đi ngắn nhất để đưa tất cả các khối đá về đích, miễn là chi phí mỗi bước đi được định nghĩa rõ ràng.
  - Phù hợp với bài toán không có thông tin heuristic: Ares's adventure (hay với tên gọi là trò chơi Sokoban) là bài toán mà trạng thái đích không dễ dự đoán trước hoặc ước lượng khoảng cách bằng heuristic, nên UCS là một lựa chọn khả thi để đảm bảo tìm giải pháp tối ưu.
- Trường hợp tối ưu cho UCS trong bài toán Sokoban:
  - Số lượng khối đá ít và không gian thoáng:
    - Khi số lượng khối đá ít và có nhiều khoảng trống trong bản đồ, UCS dễ dàng tìm được đường đi tối ưu mà không gặp nhiều trạng thái chồng chéo.
    - Khoảng trống rộng giúp giảm thiểu khả năng rơi vào các tình huống bế tắc, và thuật toán không cần tốn nhiều thời gian để duyệt qua các trạng thái không cần thiết.
  - Ít rào chắn và không gian hẹp:
    - Nếu bản đồ có ít rào chắn hoặc chỉ có các đường đi đơn giản, UCS có thể duyệt nhanh qua các trạng thái khả dĩ và nhanh chóng tìm ra giải pháp ngắn nhất.
    - Các bản đồ dạng hành lang đơn, ít ngã rẽ phức tạp sẽ tối ưu hơn vì UCS có thể đi thẳng đến đích mà không cần xét nhiều khả năng phân nhánh.
  - Vị trí đích và khối đá dễ tiếp cận:
    - Nếu các vị trí đích được đặt gần nhau và dễ tiếp cận, UCS sẽ nhanh chóng tìm thấy các trạng thái đạt yêu cầu mà không cần duyệt sâu vào các phần khác của không gian trạng thái.
    - Trường hợp này giảm thiểu số lần đẩy đá vào các vị trí xa, giúp tối ưu thời gian tìm kiếm.
- Nhược điểm:
  - Tốn nhiều thời gian và bộ nhớ: Sokoban có không gian trạng thái lớn, nên UCS dễ gặp vấn đề về hiệu suất. Độ phức tạp không gian của UCS có thể nhanh chóng vượt quá khả năng xử lý của bộ nhớ, đặc biệt là với các bản đồ lớn và nhiều khối đá.

- Không hiệu quả trong xử lý các trạng thái bế tắc: UCS không có khả năng phát hiện và tránh các trạng thái bế tắc (các tình huống khối đá bị đẩy vào góc và không thể di chuyển). Điều này làm gia tăng độ phức tạp và có thể khiến thuật toán lãng phí thời gian vào các hướng đi không có giải pháp.

- Trường hợp tệ nhất cho UCS trong bài toán Sokoban
  - Số lượng khối đá nhiều và không gian chật hẹp:
    - Khi số lượng khối đá nhiều và bản đồ có không gian chật hẹp, UCS sẽ phải xét nhiều trạng thái đẩy đá khác nhau, tăng độ phức tạp của không gian tìm kiếm.
    - Các khối đá có thể chặn lối đi và dẫn đến nhiều trường hợp bế tắc, buộc UCS phải duyệt qua nhiều trạng thái mà không có tiến triển trong việc đạt đến trạng thái đích.
  - Nhiều ngã rẽ và rào chắn phức tạp:
    - Các bản đồ có nhiều ngã rẽ, rào chắn hoặc đường cụt sẽ làm tăng độ phức tạp, khiến UCS phải duyệt qua nhiều khả năng di chuyển không hiệu quả.
    - Khi Ares bị buộc phải chọn giữa nhiều hướng khác nhau, UCS sẽ phải xử lý tất cả các khả năng, dẫn đến lãng phí tài nguyên khi duyệt qua các ngã rẽ không cần thiết.
  - Vị trí đích khó tiếp cận và dễ bị chặn:
    - Nếu các vị trí đích nằm sâu trong các khu vực khó tiếp cận, đòi hỏi di chuyển các khối đá qua nhiều bước đẩy phức tạp, UCS sẽ tốn nhiều thời gian xử lý.
    - Trong các trường hợp này, UCS có thể phải duyệt qua nhiều tổ hợp vị trí khối đá trước khi tìm ra giải pháp tối ưu.
  - Cấu trúc bản đồ dễ gây bế tắc:
    - Trong các bản đồ có nhiều vị trí dễ gây bế tắc (ví dụ, các góc hoặc hẻm cụt), UCS không tự động tránh được các tình huống này vì không có heuristic.
    - Khi khối đá bị đẩy vào góc không thể thoát ra, UCS sẽ mất nhiều thời gian để khám phá hết các khả năng dẫn đến trạng thái đó trước khi quay lại thử hướng đi khác, gây ra độ phức tạp thời gian cao.

VII. Thuật toán A*

a. Nội dung thuật toán

- Định nghĩa các hàm trong chi phí trong A*:
  - g(n): chi phí từ điểm bắt đầu đến một điểm n. Đây là chi phí thực tế đã đi từ điểm xuất phát.
  - h(n): Chi phí ước lượng từ điểm n đến đích, còn gọi là hàm heuristic. Heuristic giúp dự đoán xem trạng thái hiện tại còn cách đích bao xa.

- f(n) = g(n) + h(n): Tổng chi phí dự đoán để đi qua điểm n. A* sẽ chọn mở rộng các trạng thái có f(n) nhỏ nhất.

- Giả sử điểm bắt đầu là start và điểm đích là goal. Tạo một hàng đợi ưu tiên. Các bước thực hiện thuật toán A*:
  - Bước 1: Khởi tạo:
    - Đặt g(start) = 0 vì chi phí đến với chính nó là 0.
    - Tạo một tập hợp closed_set để lưu trữ các trạng thái đã mở rộng.
    - Đưa trạng thái bắt đầu start vào hàng đợi ưu tiên với chi phí f(start) = g(start) + h(start).
  - Bước 2: Lặp qua các trạng thái trong hàng đợi:
    - Trong mỗi vòng lặp, lấy trạng thái n có chi phí f(n) thấp nhất từ hàng đợi ưu tiên.
    - Nếu n là điểm đích goal, dừng thuật toán và trả về đường đi từ start đến goal
    - Nếu không thì thực hiện bước 3.
  - Bước 3: Mở rộng trạng thái:
    - Duyệt qua các trạng thái con của trạng thái n.
    - Với mỗi trạng thái con m, tính toán chi phí thực tế g(m) = g(n) + cost(n,m) và chi phí dự đoán f(m) = g(m) + h(m).
    - Nếu trạng thái m chưa có trong closed_set hoặc có giá trị g(m) nhỏ hơn giá trị cũ thì thêm m vào closed_set với chi phí f(m).
  - Bước 4: Kết thúc:
    - Tiếp tục lặp cho đến khi tìm thấy đích hoặc hàng đợi rỗng, nếu rỗng trả về None.
    - Tính chất của A*:
  - A* có thể xử lý hiệu quả các bài toán tìm đường trong không gian tìm kiếm lớn, đặc biệt khi hàm heuristic h(n) tốt, giúp thuật toán giảm số lượng trạng thái cần mở rộng và tối ưu hóa chi phí tính toán.

b. Ý tưởng vận hành trong sokoban

- Đại diện trạng thái:
  - Mỗi trạng thái n của trò chơi gồm vị trí của nhân vật và tất cả các khối đá. Mỗi trạng thái chứa thông tin cần thiết để xác định trạng thái trò chơi hiện tại.
  - Mỗi cạnh biểu thị một nước đi hợp lệ như nhân vật di chuyển hoặc đẩy khối đá.
- Trạng thái ban đầu: vị trí khởi tạo của nhân vật và các khối đá.
- Trạng thái mục tiêu: các khối đá đặt đúng vào đích sao cho chi phí và tổng khối lượng đẩy là thấp nhất.

- Hàm heuristic h(n): Nhóm em sử dụng cách tính tổng khoảng cách Manhattan từ mỗi khối đá đến đích gần nhất làm hàm h(n). Khoảng cách Manhattan là tổng chênh lệch tọa độ x và tọa độ y. Heuristic này là chấp nhận được vì nó không bao giờ ước lượng quá cao chi phí để đạt được mục tiêu. Nghĩa là, nó tính toán khoảng cách thực tế mà khối đá cần di chuyển đến mục tiêu mà không tính đến các chướng ngại vật.
- Hàm chi phí hiện tại g(n): tính số nước đi đã thực hiện. Mỗi nước đi có chi phí là 1.
- Frontier: hàng đợi ưu tiên.
- Actions: danh sách các hành động.
- Cost_so_far: đóng vai trò như là closed_set.
- Explored_set: tập hợp các trạng thái đã được khám phá và đánh giá.

c. Giải thích chi tiết thuật toán

- Trạng thái bắt đầu:
  - beginAres và beginStone xác định vị trí lần lượt của Ares và các khối đá.
  - startStone: lưu trữ vị trí bắt đầu của Ares và các khối đá, được biểu diễn dưới dạng tuple. Mỗi trạng thái lưu trữ .
- Danh sách mở và đóng:
  - Frontier: Hàng đợi ưu tiên lưu trữ các trạng thái cần đánh giá, sắp xếp theo chi phí f = g + h. Chi phí này đảm bảo rằng các trạng thái có chi phí thấp hơn sẽ được mở trước.
  - ExploredSet: exploredSet chứa các trạng thái đã được duyệt qua, tránh lặp lại các trạng thái đã kiểm tra.
- Hàm heuristic h(n): heuristic(posAres, posStone) tính tổng khoảng cách Manhattan từ mỗi khối đá đến đích gần nhất.
- Vòng lặp chính:
  - Lấy trạng thái có chi phí thấp nhất từ frontier. Trạng thái bao gồm vị trí Ares (currentAresPos), vị trí các khối đá (currentStonePos) và chi phí để đạt đến trạng thái này (current_cost).
  - Kiểm tra trạng thái kết thúc: Nếu vị trí các đá đã đạt đích thì kết thúc thuật toán và xuất ra kết quả.
- Duyệt các hành động hợp lệ:
  - Các hành động hợp lệ: Duyệt qua các hành động hợp lệ của Ares từ vị trí hiện tại legalActions(currentAresPos, currentStonePos). Mỗi hành động hợp lệ là di chuyển hoặc đẩy.
  - Với mỗi hành động, cập nhật newPosAres và newPosStone từ hàm updateState.

- Bỏ qua trạng thái không hợp lệ bằng hàm isFailed(newPosStone) (deadlock)

- Tính toán chi phí: Nếu hành động là đẩy một khối đá, thuật toán sẽ tính thêm chi phí theo khối lượng của khối đá weightList[pushed_stone_index], còn nếu di chuyển bình thường không đẩy thì chi phí là 1.
- Kiểm tra và thêm trạng thái mới vào Frontier:
  - So sánh chi phí: Nếu trạng thái mới new_state chưa được duyệt qua, hoặc chi phí mới thấp hơn chi phí trước đó của cùng trạng thái cost_so_far[new_state] thì:
    - Cập nhật chi phí mới cho trạng thái cost_so_far[new_state] = new_cost.
    - Thêm trạng thái mới vào frontier với ưu tiên là tổng của chi phí mới và giá trị của heuristic (new_cost + heuristic (newPosAres, newPosStone))
    - Thêm hành động vào hàng đợi hành động actions, để theo dõi đường đi (các hành động đã thực hiện) từ đầu đến trạng thái đích.
- Khi tìm thấy trạng thái kết thúc, thuật toán sẽ:
  - Ghi lại các bước di chuyển.
  - Tính toán các số liệu như thời gian, bộ nhớ, số node đã mở và chi phí đẩy tổng cộng.
  - Ghi thông tin và kết quả.

d. Đánh giá thuật toán

- Tính phù hợp của A* trong các loại bản đồ Sokoban:
  - Bản đồ nhỏ và trung bình: A* hoạt động hiệu quả trên các bản đồ kích thước nhỏ và trung bình, nơi không gian trạng thái chưa quá lớn. Trong trường hợp này, số lượng trạng thái cần duyệt và lưu trữ vẫn trong tầm kiểm soát, nên thời gian tính toán và bộ nhớ không bị tiêu tốn quá nhiều.
  - Bản đồ có ít chướng ngại vật và đá: Với các bản đồ có số lượng đá và chướng ngại vật (tường) vừa phải, A* có thể tìm được đường đi tối ưu dễ dàng mà không cần tốn quá nhiều tài nguyên.
- Hạn chế: Tiêu tốn nhiều bộ nhớ và thời gian trên bản đồ lớn, Sokoban có không gian trạng thái lớn do nhiều khả năng di chuyển và tương tác giữa Ares và đá.Khi bản đồ phức tạp và có nhiều đá, không gian trạng thái mở rộng rất nhanh, làm cho A* cần nhiều bộ nhớ để lưu trữ các trạng thái đã duyệt và danh sách mở rộng.

VIII. So sánh thuật toán

PHẦN 4: TÀI LIỆU THAM KHẢO

I. Tài liệu nhóm

[1] Đường dẫn github: https://github.com/Dzivilord/Sokoban.git

[2] Đường dẫn video mô tả:

II. Tài liệu tham khảo

[1] Angusfung, 'sokoban-AI' [Trực tuyến]. Địa chỉ: https://github.com/angusfung/sokoban-AI

[2] Stuart Russel & Peter Norvig, 'Artificial Intelligence A Morden Approach' Third Edition.

[3] Hiếu Lê, 'Nhập môn Trí Tuệ Nhân Tạo - Bài tập lớn - Áp dụng giải thuật tìm kiếm vào game SOKOBAN – HCMUT'[Trực tuyến]. Địa chỉ: https://www.youtube.com/watch?v=ev6XPeJ6mnw&t=99s

[4] Phạm Vĩnh Lộc, 'Demo Sokoban with BFS and A*' [Trực tuyến]. Địa chỉ: https://www.youtube.com/watch?v=bKK74HN4T9c&t=88s

[5] Tim Wheeler, 'Basic Search Algorithms on Sokoban' [Trực tuyến]. Địa chỉ: https://timallanwheeler.com/blog/2022/01/19/basic-search-algorithms-on-sokoban/

[6] Xbandrade, 'sokoban-solver-generator' [Trực tuyến]. Địa chỉ: https://github.com/xbandrade/sokoban-solver-generator/tree/main