ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

LỚP TRÍ TUỆ NHÂN TẠO KHÓA 2023 - 23TNT1

Báo cáo Đồ án

Đề tài: Trò chơi Mê cung

Môn học: Thực hành - Lập trình cho Trí tuệ Nhân tạo

Sinh viên thực hiện:
Đinh Đức Anh Khoa (23122001)
Nguyễn Đình Hà Dương (23122002)
Nguyễn Lê Hoàng Trung (23122004)
Đinh Đức Tài (23122013)

Giáo viên hướng dẫn:
Th.S Nguyễn Trần Duy Minh

Ngày 27 tháng 5 năm 2024

Mục lục

1 Giới thiệu

Đây là bài báo cáo cho Đồ án Trò chơi Mê cung, môn Thực hành - Lập trình cho Trí tuệ Nhân tạo, lớp 23TNT1, Khoa Công nghệ thông tin, Trường Đại học Khoa học tự nhiên - Đại học Quốc gia TP.HCM.

Đồ án được thực hiện bởi nhóm các thành viên:

- Đinh Đức Anh Khoa (23122001)
- Nguyễn Đình Hà Dương (23122002)
- Nguyễn Lê Hoàng Trung (23122004)
- Đinh Đức Tài (23122013)

2 Phân công nhiệm vụ, đánh giá mức độ hoàn thành

Bảng phân công nhiệm vụ cho từng thành viên:

Họ và tên	MSSV	Nhiệm vụ	Mức độ hoàn thành
Đinh Đức Anh Khoa	23122001	- Viết báo cáo thuật toán: A*, BFS, so sánh A* và BFS, phát sinh mê cung, hệ thống gợi ý đường đi - Viết code thuật toán A* - Game tester, quay demo	Tốt (100%)
Nguyễn Đình Hà Dương	23122002	- Design hình ảnh menu và game - Viết code giao diện menu, gameplay; xử lý đồ hoạ và xử lý trong gameplay	Tốt (100%)
Nguyễn Lê Hoàng Trung	23122004	- Viết code và báo cáo thuật toán BFS - Viết code giao diện gameplay; xử lí đồ hoạ, dữ liệu của ma trận và xử lí thao tác, chuyển động của nhân vật	Tốt (100%)
Đinh Đức Tài	23122013	- Viết báo cáo. Viết thuật toán mê cung sinh năng lượng. - Viết code giao diện, chức năng menu; code database - Thiết kế cấu trúc phần mềm	Tốt (100%)

Tự đánh giá mức độ hoàn thành của từng yêu cầu:

Các yêu cầu chính		**Nội dung**	**Tự đánh giá mức độ hoàn thành**
Xử lí tài khoản	Đăng nhập	Người dùng nhập tên đăng nhập và mật khẩu để đăng nhập vào game	Tốt (100%)
	Đăng kí	Nếu người dùng chưa có tài khoản, phải điền tên và mật khẩu để sử dụng	Tốt (100%)
Menu	Chế độ chơi	Lựa chọn chế độ chơi (tự chơi, tự động), độ khó (dễ, trung bình, khó), phát sinh bản đồ (ngẫu nhiên, tự chọn)	Tốt (100%)
	Bảng xếp hạng	Sau khi qua mỗi mê cung, người dùng sẽ được lưu thời gian qua màn, số bước đã sử dụng từ đó lưu vào bảng xếp hạng. Bảng xếp hạng được chia theo độ khó.	Tốt (100%)
	Thoát game	Kết thúc trò chơi	Tốt (100%)
Lưu trạng thái người chơi		Ở chế độ tự chơi, khi người chơi chưa hoàn thành nhưng đã nghỉ giữa chừng thì phải lưu lại được trạng thái và có thể load lại map được khi người dùng quay lại chơi.	Tốt (100%)
Các yêu cầu khác		**Nội dung**	**Tự đánh giá mức độ hoàn thành**
Gợi ý đường đi		Ở chế độ tự chơi, khi người dùng nhấn vào nút gợi ý, hệ thống sẽ hiển thị đường đi từ vị trí hiện tại của nhân vật đến đích.	Tốt (100%)
Âm nhạc, hình nền		Tạo âm nhạc và hình nền cho trò chơi	Tốt (100%)
Lý thuyết trò chơi		Cho trước số bước đi tối đa của Tâm là H, và phát sinh ngẫu nhiên K viên năng lượng trong bản đồ. Tìm đường đi ngắn nhất đến nhà Gia Huy, biết rằng cứ mỗi bước đi sẽ mất 1 năng lượng và ăn được 1 viên năng lượng sẽ có thêm V_k năng lượng với V_k là giá trị phát sinh ngẫu nhiên thuộc {1, 2, 3, 4, 5}. Giá trị của H và K cũng sẽ tương ứng với các mức độ trò chơi.	Khá (65%)

3 Các thư viện và công nghệ

3.1 Python

Ngôn ngữ lập trình Python [1]

Python là một ngôn ngữ lập trình thông dịch, hướng đối tượng, cấp cao với ngữ nghĩa động. Các cấu trúc dữ liệu tích hợp cấp cao của nó, kết hợp với kiểu động và liên kết động, làm cho Python rất hấp dẫn để phát triển ứng dụng nhanh, cũng như sử dụng như một ngôn ngữ kịch bản hoặc ngôn ngữ kết nối để kết nối các thành phần hiện có với nhau. Cú pháp đơn giản, dễ học của Python nhấn mạnh tính dễ đọc và do đó giảm chi phí bảo trì chương trình. Python hỗ trợ các mô-đun và gói, khuyến khích tính mô-đun của chương trình và tái sử dụng mã nguồn. Trình thông dịch Python và thư viện chuẩn phong phú có sẵn dưới dạng mã nguồn hoặc dạng nhị phân mà không mất phí cho tất cả các nền tảng chính, và có thể được phân phối tự do.

Phiên bản

Đồ án sử dụng Python phiên bản **3.11.9**.

3.2 Pygame

Thư viện Pygame [2]

Pygame là một bộ mô-đun Python đa nền tảng được thiết kế để viết trò chơi điện tử. Pygame bao gồm đồ họa máy tính và thư viện âm thanh được thiết kế để sử dụng với ngôn ngữ lập trình Python.

Phiên bản

Đồ án sử dụng Pygame phiên bản **2.5.2**.

3.3 Pygame-menu

Thư viện Pygame-menu [3]

Pygame-menu là một thư viện python-pygame để tạo menu và giao diện người dùng đồ họa (GUI). Pygame-menu hỗ trợ nhiều widget khác nhau, chẳng hạn như nút bấm, bộ chọn màu, đồng hồ, bộ chọn thả xuống, khung, hình ảnh, nhãn, bộ chọn, bảng, đầu vào văn bản, công tắc màu và nhiều hơn nữa, với nhiều tùy chọn để tùy chỉnh.

Phiên bản

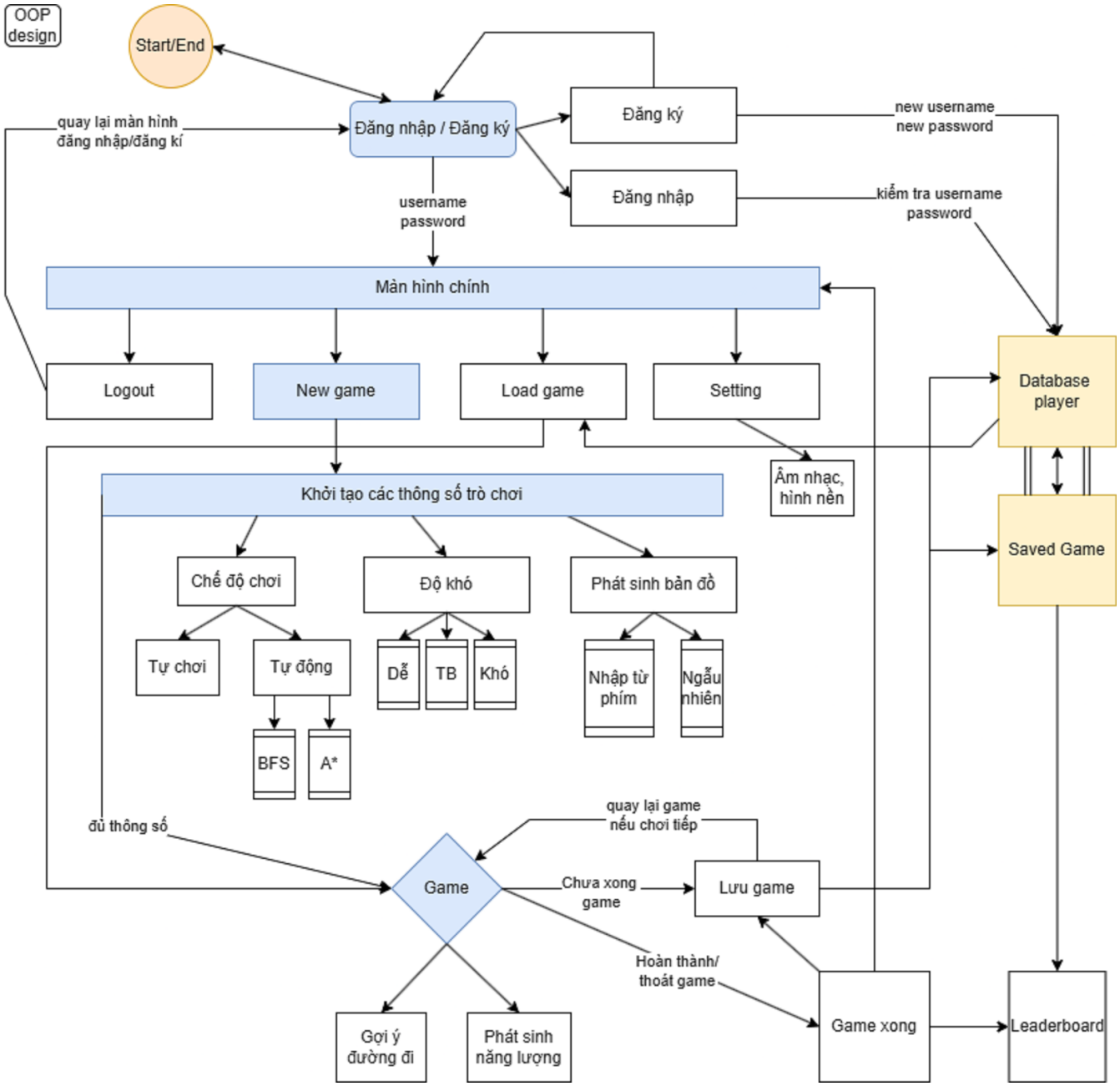
Đồ án sử dụng Pygame-menu phiên bản **4.4.3**.

4 Các thành phần trong code

4.1 Tổng quan về code

Cấu trúc

Các thành phần trong game được tổ chức như hình bên dưới:



Hình 1: Sơ đồ tổ chức code

Phương pháp lập trình

Code trong đồ án được tổ chức theo phương pháp lập trình hướng đối tượng. Điều này giúp code rõ ràng, rành mạch, không bị chồng chéo các biến và chức năng với nhau.

Tổ chức code

Code được tổ chức thành các file với đuôi *.py*. Mỗi file chứa các class, chịu trách nhiệm cho các chức năng riêng, trong đó:

- **main.py** : Gọi các tệp tin và class khác, khởi tạo trò chơi.
- **database.py** : Lưu trữ thông tin người dùng, các ván chơi và xử lí xuất nhập thông tin, gồm 1 class: `UserDatabase`
- **login_startgame.py** : Xử lí giao diện đăng kí, đăng nhập, menu, gồm 2 class: `LoginMenu`, `MenuGame`
- **maze_generator.py** : Khởi tạo các thuộc tính cho mê cung, gồm 2 class: `Cell`, `Maze`
- **maze_solver.py** : Tìm đường đi cho mê cung, gồm 2 class: `priority_queue`, `mazeSolver`
- **utils.py** : Xử lí thông tin của 1 thư mục, gồm 1 hàm `import_folder`
- **tile.py** : Hiển thị hình ảnh lên một ô / màn hình (đường đi, gợi ý, đường thuật toán tìm kiếm...), gồm 3 class: `Goal`, `Hint`, `Auto`.
- **Display.py** : Tạo các nút bấm, đồng hồ, hộp văn bản, gồm 5 class: `Clock`, `Button`, `Button_Image`, `TextBox`, `Display`,
- **player.py** : Quản lí trạng thái và hành vi của nhân vật người chơi, gồm 1 class: `Player`
- **level.py** : Quản lí các thành phần của trò chơi, gồm 1 class: `Level`
- **game.py** : Khởi tạo 1 game, gồm 1 class: `Game`

Các file phương tiện khác

Các file còn lại có mục đích hiển thị hình ảnh, âm thanh, font chữ,... được chia vào 4 thư mục:

- **assets** : Ảnh nền, hoạt hoạ game mặc định
- **Themebeach** : Ảnh nền với theme biển
- **font** : Các font chữ
- **sound** : Các file âm thanh

4.2 Chi tiết các thành phần code

4.2.1 Vận hành database

File database.py - class `UserDatabase`

- **Mục đích**: Xử lí các yêu cầu liên quan đến lưu trữ và trích xuất dữ liệu người dùng, các ván chơi, cài đặt mặc định...
- **Thuộc tính**:
  - *filename*: tên file json - file lưu trữ thông tin.
  - *users*: chứa thông tin của người dùng.
- **Phương thức**:
  - *load_data*: Tải dữ liệu từ file json vào thuộc tính *users*.
  - *save_data*: Lưu dữ liệu của thuộc tính *users* vào file json.
  - *register_user*: Đăng kí một người dùng mới, lưu vào file json.
  - *login_user*: Kiểm tra tính hợp lệ của username và password với dữ liệu hiện có.
  - *load_users*: Trả về toàn bộ dữ liệu của một người chơi.
  - *load_game*: Trả về dữ liệu của một game của một người chơi nào đó.
  - *save_game*: Lưu dữ liệu một game của một người chơi.
  - *leaderboard*: Sắp xếp thứ hạng game của các người chơi, trả về thứ hạng theo độ khó.

4.2.2 Giao diện các menu

File login_startgame.py - class `LoginMenu`

- **Mục đích**: Tạo menu xử lí đăng nhập, đăng kí tài khoản người chơi.
- **Thuộc tính**:
  - *surface*: Màn hình để hiển thị menu. Đây là một thuộc tính được khởi tạo dựa trên thư viện pygame-menu.
  - *main_menu*: Menu chính của phần đăng nhập, đăng kí. Đây là một thuộc tính được khởi tạo dựa trên thư viện pygame-menu.
  - Ngoài ra còn có các thuộc tính như các menu con, button, widget, âm thanh... là thành phần cấu tạo nên *main_menu*.
- **Phương thức**:
  - *check_login*, *check_register*: Kiểm tra tính hợp lệ của username và password được đăng nhập hoặc đăng kí, đưa ra thông báo.
  - *reset_noti_login*, *reset_noti_regis*: Tạo thông báo lúc đăng kí/đăng nhập cho người chơi biết.

- *init_theme*: Khởi tạo các cài đặt, hình ảnh, định dạng menu.
- *init_login_menu*, *init_register_menu*, *init_main_menu*, *init_menu*: Khởi tạo menu và các thành phần của menu (dựa trên thư viện pygame-menu).
- *start*: Hàm dùng để gọi việc hiển thị menu đăng nhập, đăng kí.

File login_startgame.py - class `MenuGame`

- **Mục đích**: Tạo menu, xử lí các hoạt động tương tác với hệ thống của người chơi đã đăng nhập (bắt đầu game mới, tải một game đã chơi, cài đặt game, đăng xuất,...).
- **Thuộc tính**:
  - *username*, *password*: Username và password của người chơi.
  - *surface*: Màn hình để hiển thị menu. Đây là một thuộc tính được khởi tạo dựa trên thư viện pygame-menu.
  - *main_menu*: Menu chính khi người chơi đã đăng nhập. Đây là một thuộc tính được khởi tạo dựa trên thư viện pygame-menu.
  - Ngoài ra còn có các thuộc tính như các menu con, button, widget, âm thanh... là thành phần cấu tạo nên *main_menu*.
- **Phương thức**:
  - *update_menu_sound_switch*, *change_bgm*, *change_sfx*: Cài đặt âm thanh.
  - *change_theme*: Thay đổi theme.
  - *reset_all_setting*: Thiết lập các cài đặt trở về mặc định.
  - *return_to_login*: Đăng xuất tài khoản và quay trở về menu đăng nhập, đăng xuất.
  - *get_data_leaderboard*: Gọi dữ liệu từ database, sắp xếp tạo leaderboard.
  - *start_a_saved_game*: Bắt đầu một game đã được lưu.
  - *init_theme*: Khởi tạo các cài đặt, hình ảnh định dạng menu.
  - *init_start_game*, *init_load_game*, *init_leaderboard*, *init_setting*, *init_main_menu*, *init_menu*: Khởi tạo menu và các thành phần của menu (dựa trên thư viện pygame-menu).
  - *start*: Hàm dùng để gọi việc hiển thị menu chính khi người chơi đã đăng nhập.

4.2.3 Khởi tạo mê cung và tìm đường đi

File maze_generator.py - class `Cell`

- **Mục đích**: Định nghĩa các thuộc tính và phương thức cho một ô trong mê cung.
- **Thuộc tính**:
  - x, y: Toạ độ của ô trong ma trận.
  - *pos*: Toạ độ hiển thị của ô trong screen.

- *width*, *wall_width* : Chiều rộng của ô, độ rộng của tường

- **Phương thức**:
  - *draw*: Vẽ hình ảnh trong 1 ô.
  - *render*: Vẽ tường bao quanh.
  - *neighbor*: Trả về tọa độ của các ô kề cạnh không bị ngăn cách bởi tường.

File maze_generator.py - class `Maze`

- **Mục đích**: Định nghĩa các thuộc tính và phương thức cho mê cung.
- **Thuộc tính**:
  - *size*: Kích thước của mê cung
  - *startX*, *startY*, *endX*, *endY*: Tọa độ ô bắt đầu và tọa độ ô kết thúc
  - *width*, *wall_width* : Chiều rộng của ô, độ rộng của tường
  - *grid*: Ma trận của mê cung
  - *trace*: Tọa độ của ô trước đó đã đi vào
  - *hint*: Gợi ý ô tiếp theo hướng đến điểm kết thúc
- **Phương thức**:
  - *breakWall*: Phá tường theo hướng.
  - *mazeGenerate*: Sinh một mê cung.
  - *render*: Hiển thị mê cung.
  - *makeHint*: Tạo gợi ý đường đi bằng BFS cho toàn bộ ô trong mê cung.
  - *hint*: Trả về đường đi gợi ý cho người chơi hướng đến điểm kết thúc đến khi gặp ngã ba.

File maze_solver.py - class `priority_queue`

- **Mục đích**: Khởi tạo, định nghĩa hàng đợi ưu tiên (priority queue)
- **Thuộc tính**:
  - *heap*: Một danh sách chứa các phần tử trong heap.
- **Phương thức**:
  - *push*: Sử dụng phương thức *heappush* từ mô-đun `heapq` để thêm một phần tử vào trong *heap* (đảm bảo *heap* có phần tử nhỏ nhất luôn ở đầu danh sách).
  - *pop*: Sử dụng phương thức *heappop* từ mô-đun `heapq`, loại bỏ và trả về phần tử nhỏ nhất từ *heap*.
  - *peek*: Trả về phần tử nhỏ nhất trong *heap* mà không loại bỏ nó.
  - *__len__*: Trả về số lượng phần tử hiện tại trong *heap*.

File maze_solver.py - class `MazeSolver`

- **Mục đích**: Sử dụng các thuật toán tìm đường (A*, BFS) để trả về đường đi.
- **Thuộc tính**:
  - *maze*: Một mê cung, được khởi tạo dựa trên class *Maze* (file maze_generator.py)
- **Phương thức**:
  - *tracePath*: Trả về danh sách gồm thứ tự di chuyển (đường đi) tìm được, bao gồm cả điểm bắt đầu và kết thúc.
  - *AStarSearch*: Chạy thuật toán A* và trả về danh sách thứ tự thăm các ô.
  - *BFS*: Chạy thuật toán BFS và trả về danh sách thứ tự thăm các ô.

4.2.4 Xử lí game

File utils.py - function `import_folder`

- **Mục đích**: Nhập và xử lí hình ảnh từ một thư mục, trả về surface (pygame) đã được chỉnh sửa kích thước.

File tile.py - class `Goal`, `Hint`, `Auto`

- **Mục đích**: Hiển thị ảnh đường đi thuật toán tìm kiếm, đường đã đi, chiến thắng game.
- **Các class**:
  - `Goal`: Vẽ thông báo chiến thắng.
  - `Hint`: Vẽ đường đã đi qua.
  - `Auto`: Vẽ thuật toán tìm kiếm.

File Display.py - class `Clock`, `Button`, `Button_Image`, `TextBox`, `Display`

- **Mục đích**: Tạo giao diện người dùng với các nút bấm, đồng hồ, hộp văn bản.
- **Các class**:
  - `Clock`: Quản lý thời gian, gồm phút, giây và có các phương thức để cập nhật và hiển thị thời gian.
  - `Button`: Quản lý nút bấm đơn giản với các khả năng tương tác với nút bấm đó.
  - `Button_Image`: Quản lý một nút bấm với hình ảnh, bao gồm cả hình ảnh khi nhấn và âm thanh khi nhấn.
  - `TextBox`: Quản lý một hộp văn bản.
  - `Display`: Hiển thị các đối tượng `Clock`, `Button`, `Button_Image`, `TextBox` lên màn hình.

File level.py - class `Level`

- **Mục đích**: Quản lí trò chơi, bao gồm người chơi, các thành phần của mê cung, tìm kiếm đường đi, hiển thị gợi ý...
- **Thuộc tính**:
  - *game*: Đối tượng Game, gồm các thông tin ban đầu của trò chơi.
  - *player*: Đối tượng Player, quản lí hoạt ảnh nhân vật.
  - *solver*: Đối tượng MazeSolver, tìm đường đi trong mê cung bằng thuật toán.
  - Ngoài ra còn có các thuộc tính khác để xử lí trò chơi.
- **Phương thức**:
  - *pack_data*: Trả về dữ liệu trạng thái của người chơi và các gợi ý để lưu hoặc truyền dữ liệu.
  - *getAuto*: Tìm đường đi dựa trên thuật toán cho trước.
  - *run*: Vẽ mê cung, hiệu ứng, gợi ý,..., xử lí logic game, điều kiện thắng.

File player.py - class `Player`

- **Mục đích**: Quản lý trạng thái và hành vi của nhân vật người chơi trong trò chơi, bao gồm việc xử lý đồ họa, di chuyển, và tương tác với môi trường mê cung.
- **Thuộc tính**:
  - *maze*, *level*: Mê cung, lớp level
  - *tilesize*: Kích thước của mỗi ô trong lưới.
- **Phương thức**:
  - *import_player_assets*: Gọi dữ liệu hoạt ảnh nhân vật.
  - *input*: Xử lí đầu vào từ người chơi.
  - *move*: Xử lí nhân vật di chuyển.
  - *getPosition*: Lấy vị trí hiện tại của người chơi.
  - *animate*: Biểu diễn hình ảnh nhân vật.
  - *getHint*: Lấy gợi ý đường đi.
  - *update*: Cập nhật hình ảnh nhân vật.

File game.py - class `Game`

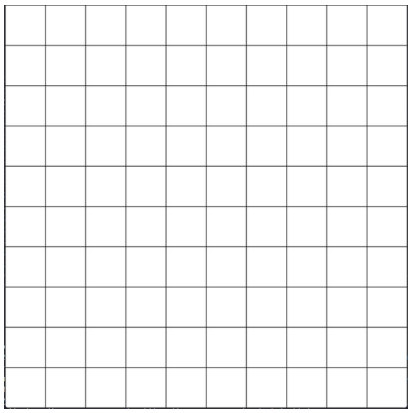
- **Mục đích**: Khởi tạo một game.
- **Thuộc tính**:
  - *mode_play*: Chế độ chơi (Player, Auto (A*), Auto (BFS))
  - $difficult$, *start*, *end*: Kích thước bản đồ, toạ độ bắt đầu, đích đến.
  - *username*, *game_name*: Tên người chơi, tên của game đang chơi.
  - *step*: Số bước người chơi đã đi.
  - Ngoài ra còn có các thuộc tính khác được khởi tạo từ các file xử lí game.
- **Phương thức**:
  - *pack_data*: Gom dữ liệu của game để đưa tới database.
  - *run*: Bắt đầu chạy game.

5 Thuật toán phát sinh mê cung, các thuật toán tìm đường và gợi ý đường đi

5.1 Thuật toán phát sinh mê cung: Randomized DFS

Trong chương trình, chúng tôi sử dụng thuật toán **duyệt theo chiều sâu ngẫu nhiên** (randomized DFS) [4] để sinh ra một mê cung. Thuật toán này sẽ sinh ra một mê cung có dạng cây. Tức là nếu coi mỗi ô trong ma trận là một đỉnh của đồ thị thì đồ thị này sẽ liên thông và không có chu trình. Điều này giúp đảm bảo từ một ô, ta có thể đi đến bất kỳ ô nào khác trong mê cung bằng một con đường duy nhất.

Ban đầu, mê cung là một ma trận gồm có các ô trống, mỗi ô trống sẽ có 4 bức tường bao xung quanh.



Hình 2: Ma trận các ô trống

Thuật toán

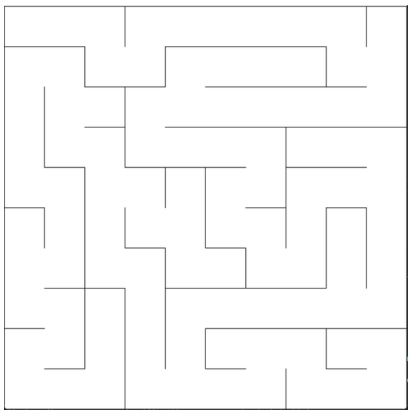
Thuật toán bắt đầu từ một ô bất kỳ trong mê cung, sau đó được thực hiện theo các bước:

Bước 1: Gọi ô hiện tại đang được duyệt là ô (x, y), ta thực hiện đánh dấu ô này là đã được duyệt qua.

Bước 2: Lần lượt chọn các ô xung quanh chung cạnh với ô (x, y) theo một thứ tự ngẫu nhiên. Tạm gọi ô đang được chọn là ô (nx, ny).

Bước 3: Nếu ô (nx, ny) chưa được duyệt thì ta gọi đệ quy duyệt tới ô đó và xóa đi bức tường giữa ô (x, y) và ô (nx, ny). Ngược lại không làm gì cả.

Sau khi thực hiện xong thuật toán ta được một mê cung thỏa mãn tính chất ở trên.



Hình 3: Mê cung được phát sinh sau khi thực hiện thuật toán

Mã giả Thuật toán sinh mê cung randomized DFS

Algorithm 1 Randomized DFS

function RANDOMIZEDDFS(x, y)
    // Đánh dấu đã duyệt ô (x, y)
    $visited(x, y) \leftarrow True$

    // Lấy danh sách các ô xung quanh (x, y) theo thứ tự ngẫu nhiên
    $to\_visit \leftarrow random\_shuffle(neighbor(x, y))$

    for $(nx, ny) \in to\_visit$ **do**
        if $visited(x, y) == False$ **then**
            loại bỏ bức tường nằm giữa hai ô (x, y) và (nx, ny)
            randomizedDFS(nx, ny)

5.2 Thuật toán tìm đường: BFS

Thuật toán **duyệt đồ thị ưu tiên chiều rộng** *(Breadth-first search - BFS)* là một trong những thuật toán tìm kiếm cơ bản và thiết yếu trên đồ thị. Ứng dụng của BFS có thể giúp ta giải quyết tốt một số bài toán trong thời gian và không gian **tối thiểu**. Đặc biệt là bài toán tìm kiếm đường đi ngắn nhất từ một đỉnh gốc tới tất cả các đỉnh khác. Trong bài toán tìm đường đi trong mê cung này, ta sẽ coi mỗi ô của mê cung là **1 đỉnh**, nếu từ 1 ô này có thể sang 1 ô khác thì sẽ có **1 cạnh** nối giữa 2 đỉnh này.

Cơ chế hoạt động

Từ một điểm xuất phát sẽ duyệt các điểm xung quanh có thể tới được, từ các điểm đã được duyệt đó sẽ thực hiện lại cơ chế trên để duyệt sang các điểm xung quanh khác, cứ như vậy cho đến khi duyệt tất cả các nhánh để tìm được nhánh có chứa đích đến thì dừng lại. Thứ tự ưu tiên của một đường đi thuật toán BFS là những đỉnh nào gần đỉnh xuất phát hơn sẽ được duyệt trước.

Thuật toán

Thuật toán sử dụng một hàng đợi (queue) để chứa các đỉnh. Các đỉnh này sau đó sẽ được duyệt theo quy tắc FIFO (first-in, first-out) của *queue*. Tức là đỉnh nào vào *queue* trước sẽ được duyệt trước.

Bước 1: Khởi tạo

- Các đỉnh đều ở trạng thái chưa được đánh dấu, ngoại trừ đỉnh xuất phát **s** đã được đánh dấu.
- Một hàng đợi ban đầu chỉ chứa 1 phần tử là **s**.

Bước 2: Lặp lại các bước sau cho đến khi hàng đợi rỗng:

- Lấy đỉnh **u** ra khỏi hàng đợi.
- Xét tất cả những đỉnh **v** kề với **u** mà chưa được đánh dấu, với mỗi đỉnh **v** đó:
  - Đánh dấu **v** đã thăm.
  - Lưu lại vết đường đi từ **u** đến **v**.
  - Đẩy **v** vào trong hàng đợi.

Mã giả Thuật toán tìm đường BFS

Algorithm 2 BFS

```
function BFS(startNode, endNode) // Đưa vào đỉnh bắt đầu và kết thúc
    // Danh sách đỉnh cần duyệt sẽ được đưa vào một hàng đợi
    queue ← hàng_đợi_rỗng
    đưa startNode vào queue

    // Khởi tạo mảng 1 chiều visited đánh dấu những đỉnh đã thăm, giá trị khởi tạo là chưa thăm
    visited[startNode] ← đã thăm

    while queue không rỗng do
        currentNode ← đỉnh tiếp theo trong queue
        if currentNode == endNode then
            return tồn tại đường đi từ đỉnh bắt đầu đến đỉnh kết thúc
        Loại bỏ currentNode khỏi queue
        for eachneighbor ∈ các_đỉnh_kề(currentNode) do // duyệt các đỉnh kề đỉnh hiện tại
            if currentNode không thể tới được neighbor then
                continue
            visited[neighbor] ← đã thăm
            đưa neighbor vào queue
     return không tồn tại đường đi từ đỉnh bắt đầu đến đỉnh kết thúc
```

Độ phức tạp thời gian: Gọi $|V|$ là số lượng đỉnh và $|E|$ là số lượng cạnh của mê cung. Độ phức tạp thời gian của thuật toán này là $O(|V| + |E|)$.

5.3 Thuật toán tìm đường: A*

A* [5] là giải thuật tìm kiếm trong đồ thị, tìm đường đi từ nút hiện tại đến đích sử dụng một hàm để ước lượng chi phí hay còn gọi là hàm heuristic. Từ trạng thái đường đi hiện tại, A* xây dựng tất cả các đường đi có thể, sử dụng hàm heuristic để đánh giá, tìm kiếm đường đi tối ưu và lưu giữ tập các lời giải trong một hàng đợi ưu tiên (priority queue). Thứ tự ưu tiên của một đường đi x được quyết định bởi hàm $f(x) = g(x) + h(x)$.

Trong đó:

- $g(x)$ là chi phí của đường đi từ nút xuất phát đến nút hiện tại.
- $h(x)$ là hàm đánh giá heuristic ước lượng chi phí từ nút hiện tại đến đích.

Hàm $f(x)$ có giá trị càng thấp thì độ ưu tiên càng cao. Có nghĩa là ta luôn ưu tiên duyệt các nút ở gần mục tiêu hơn. Nếu hàm heuristic h có tính chất đơn điệu (hay nhất quán) tức thỏa mãn điều kiện $h(x) \leq d(x, y) + h(y)$ với mọi cạnh $x - y$ (ở đây $d(x, y)$ là độ dài cạnh $x - y$) thì thuật toán A* sẽ đảm bảo tìm ra một đường đi tối ưu.

Vì bài toán tìm đường đi trong mê cung của ta là bài toán tìm đường đi trên cây có nghĩa là tồn tại một đường đi duy nhất từ đỉnh bắt đầu đến đỉnh kết thúc mà không đi qua một đỉnh quá hai lần nên ta không cần quan tâm đến tính đơn điệu của hàm heuristic. Điều này sẽ khiến thuật toán chạy nhanh hơn trong đa phần trường hợp. Trong chương trình, chúng tôi sử dụng hàm heuristic là bình phương khoảng cách euclid giữa điểm hiện tại và điểm kết thúc trong mê cung. Nói cách khác, nếu đỉnh hiện tại là ô (x, y) và đỉnh kết thúc là ô $(endX, endY)$, ta có hàm heuristic:

$$h(x, y) = (endX - x)^2 + (endY - y)^2$$

Khi di chuyển qua một ô thì độ dài đường đi tăng thêm 1 nên nếu đỉnh hiện tại là ô (x, y) và đỉnh tiếp theo là ô (u, v), ta có:

$$g(u, v) = g(x, y) + 1$$

Vậy ta có **độ ưu tiên** cho ô (u, v) nếu ô trước đó là ô (x, y):

$$\begin{aligned} f(u, v) &= g(u, v) + h(u, v) \\ &= [g(x, y) + 1] + [(endX - u)^2 + (endY - v)^2] \end{aligned}$$

Mã giả cho thuật toán A*:

Algorithm 3 A*

function AStar(*startNode*, *endNode*) // Đưa vào đỉnh bắt đầu và kết thúc
    // Danh sách đỉnh cần duyệt sẽ được đưa vào một hàng đợi ưu tiên
    openSet ← *priority_queue*()
    // Chi phí của đường từ điểm xuất phát đến điểm hiện tại
    $g \leftarrow$ {giá trị các đỉnh: $\infty \forall$ đỉnh và 0 với đỉnh bắt đầu}
    // Tổng chi phí
    $f \leftarrow$ {giá trị các đỉnh: $\infty \forall$ đỉnh và $heuristic(startNode)$ với đỉnh bắt đầu}

    Đưa cặp *startNode* cùng chi phí $f(startNode)$ vào *openSet*
    while *openSet* không rỗng **do**
        currentNode ← đỉnh trong openSet có tổng chi phí thấp nhất
        if *currentNode* == *endNode* **then**
            return tìm thấy đường đi
        Loại bỏ *currentNode* khỏi *openSet*

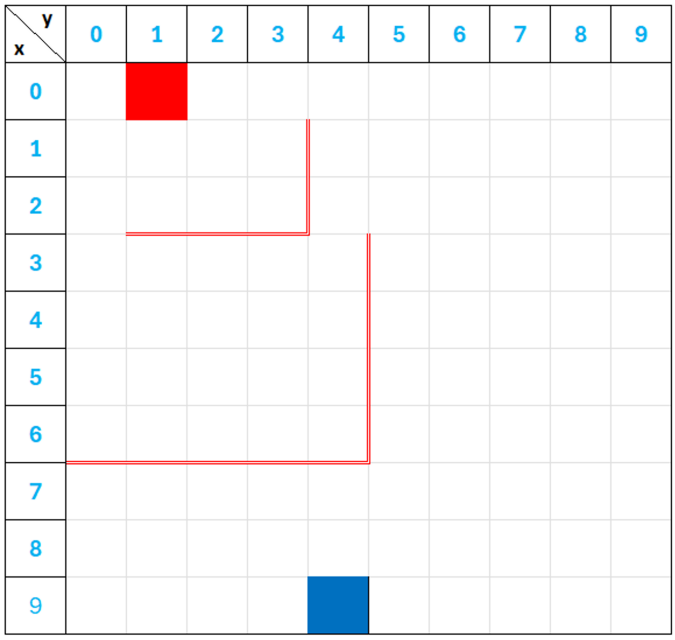
        // duyệt qua các đỉnh kề hợp lệ với đỉnh hiện tại
        for *each neighbor* ∈ các_đỉnh_kề(*currentNode*) **do**
            $newG \leftarrow g(currentNode) + 1$
            $newF \leftarrow newG + heuristic(currentNode)$
            if $newG < g(neighbor)$ **then**
                $g(neighbor) \leftarrow newG$
                $f(neighbor) \leftarrow newF$
                đưa cặp *neighbor* và $f(neighbor)$ vào *openSet*
    return thất bại, không tìm thấy đường đi

Độ phức tạp thời gian

Độ phức tạp thời gian của thuật toán A* phụ thuộc vào đánh giá heuristic. Trong trường hợp tệ nhất, A* sẽ có độ phức tạp là $O(b^d)$ với b là số cạnh trung bình của mỗi đỉnh và d là số lượng nút trên đường đi tối ưu từ đỉnh bắt đầu đến đỉnh kết thúc. Mặc dù vậy, A* vẫn là một trong những thuật toán tìm kiếm đường đi tốt nhất trong hầu hết các trường hợp.

Ví dụ minh hoạ

Trong ví dụ này ta chỉ quan tâm đến cách thuật toán A* hoạt động trong một ma trận bất kỳ, không phải cách thuật toán hoạt động ở trong một mê cung. Giả sử ta có một ma trận kích thước $10 * 10$ như hình 4 với ô màu đỏ là ô xuất phát và ô màu xanh là ô kết thúc. Các viền màu đen đậm hơn bên ngoài và viền màu đỏ ở trong là các bức tường. Ở đây, ta sẽ lập sẵn các bảng giá trị cho từng hàm để mô tả thuật toán tốt hơn.



Hình 4: Ma trận giả định với điểm xuất phát, kết thúc và các bức tường

Sử dụng công thức cho hàm $g(u, v)$ và $h(u, v)$ như đã nói ở trên, ta sẽ có được hai bảng giá trị cho hàm g và h như hình 5.

x \ y	0	1	2	3	4	5	6	7	8	9
0	1		1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7	8	9
2	3	2	3	4	5	6	7	8	9	10
3	4	5	6	7	6	7	8	9	10	11
4	5	6	7	8	7	8	9	10	11	12
5	6	7	8	9	8	9	10	11	12	13
6	7	8	9	10	9	10	11	12	13	14
7	16	15	14	13	12	11	12	13	14	15
8	17	16	15	14	13	12	13	14	15	16
9	18	17	16	15		13	14	15	16	17

g(u, v)

x \ y	0	1	2	3	4	5	6	7	8	9
0	97		85	82	81	82	85	90	97	106
1	80	73	68	65	64	65	68	73	80	89
2	65	58	53	50	49	50	53	58	65	74
3	52	45	40	37	36	37	40	45	52	61
4	41	34	29	26	25	26	29	34	41	50
5	32	25	20	17	16	17	20	25	32	41
6	25	18	13	10	9	10	13	18	25	34
7	20	13	8	5	4	5	8	13	20	29
8	17	10	5	2	1	2	5	10	17	26
9	16	9	4	1		1	4	9	16	25

h(u, v)

Hình 5: Bảng giá trị g và h trong ma trận

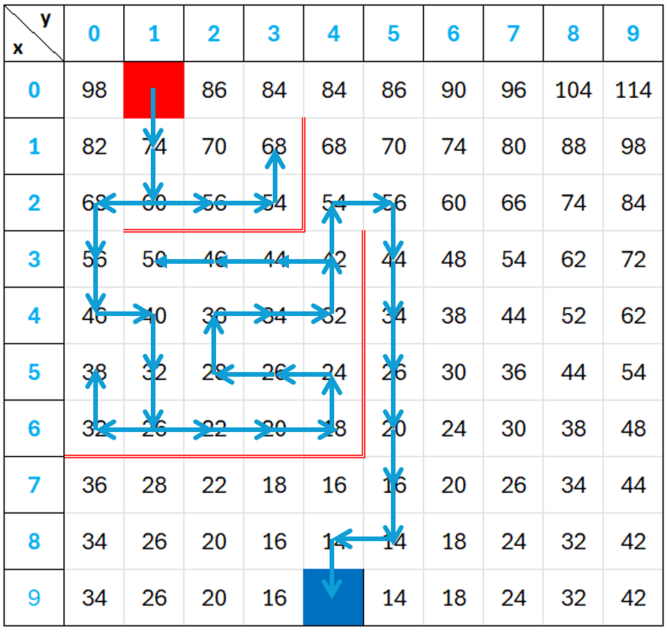
Vì $f(u, v) = g(u, v) + h(u, v)$ nên ta sẽ có bảng cuối cùng như hình 6:

x \ y	0	1	2	3	4	5	6	7	8	9
0	98		86	84	84	86	90	96	104	114
1	82	74	70	68	68	70	74	80	88	98
2	68	60	56	54	54	56	60	66	74	84
3	56	50	46	44	42	44	48	54	62	72
4	46	40	36	34	32	34	38	44	52	62
5	38	32	28	26	24	26	30	36	44	54
6	32	26	22	20	18	20	24	30	38	48
7	36	28	22	18	16	16	20	26	34	44
8	34	26	20	16	14	14	18	24	32	42
9	34	26	20	16		14	18	24	32	42

f(u, v)

Hình 6: Bảng giá trị hàm f trong ma trận

Thuật toán khi bắt đầu chạy sẽ xuất phát ở ô màu đỏ là ô $(0, 1)$, duyệt qua các ô chung cạnh xung quanh để tìm một đường đi tới ô kết thúc. Thứ tự duyệt các ô sẽ dựa vào hàm $f(u, v)$ ta vừa tính được, ô có giá trị của hàm f nhỏ hơn sẽ được ưu tiên duyệt trước. Ta có được sơ đồ đường đi của thuật toán như hình 7. Có thể thấy rằng, đường đi được tìm thấy không phải là đường đi ngắn nhất. Đó là bởi vì ta đã bỏ qua tính chất đơn điệu của hàm heuristic như đã nói ở trên.



Hình 7: Sơ đồ đường đi thuật toán A*

5.4 So sánh thuật toán tìm đường BFS và A*

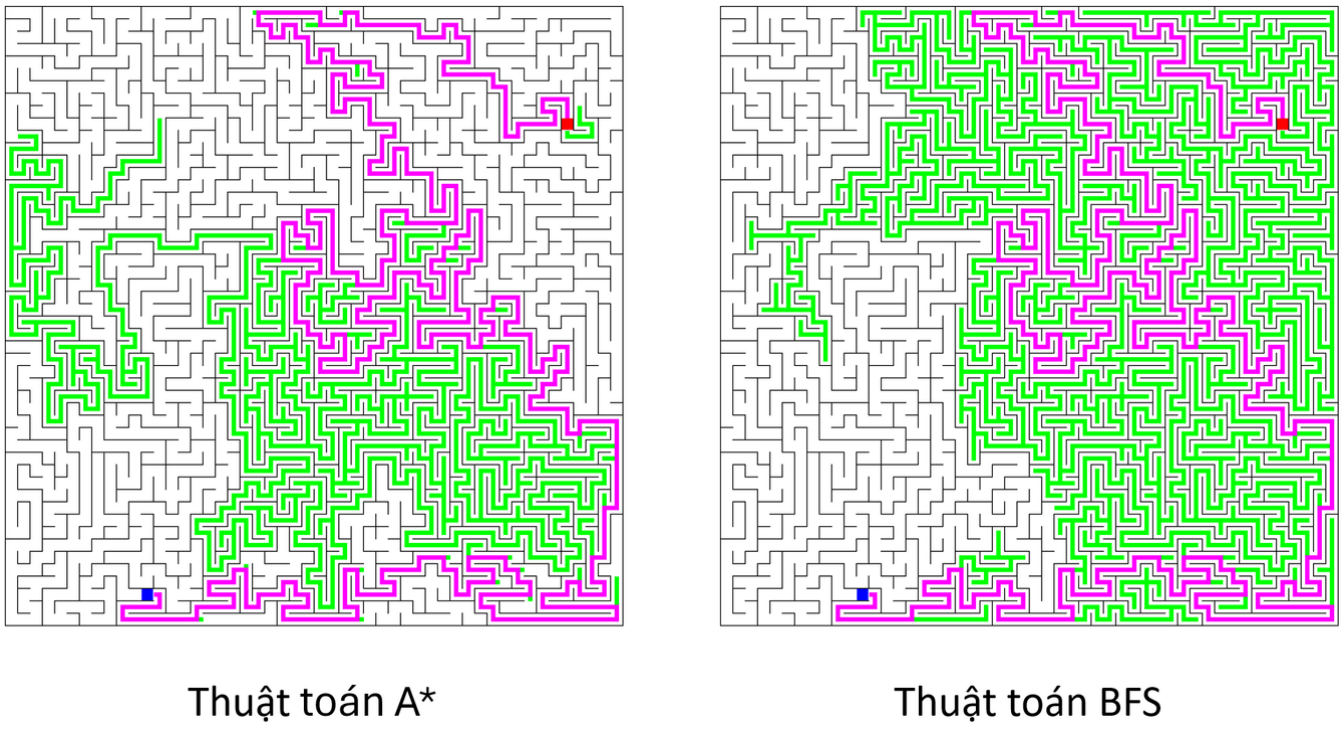
Về hiệu suất

Sau 10 lần thử nghiệm và quan sát hai thuật toán A* và BFS với các mê cung kích thước $100 * 100$ khác nhau, chúng tôi có được bảng thống kê thời gian chạy của hai thuật toán này như hình 8:

Thuật toán \ Số lần	1	2	3	4	5	6	7	8	9	10
A*	5.20 (s)	13.93 (s)	5.69 (s)	10.50 (s)	10.47 (s)	8.01 (s)	11.62 (s)	2.02 (s)	6.34 (s)	8.92 (s)
BFS	7.11 (s)	11.94 (s)	7.38 (s)	10.49 (s)	15.89 (s)	16.92 (s)	12.92 (s)	3.73 (s)	10.61 (s)	12.61 (s)

Hình 8: So sánh thời gian chạy của 2 thuật toán A* và BFS

Có thể thấy rằng trong đa phần trường hợp, thuật toán A* sẽ có tốc độ nhanh hơn thuật toán BFS. Giải thích cho điều này là vì khác với BFS phải tìm kiếm sang toàn bộ các nút lân cận có cùng độ sâu, thuật toán A* sẽ chỉ ưu tiên tìm kiếm các nút được cho là gần hơn với mục tiêu dựa vào hàm heuristic. Vì vậy, thuật toán A* trong đa phần trường hợp sẽ duyệt qua ít đỉnh hơn so với BFS như hình 9 (ô bắt đầu là ô màu đỏ và ô kết thúc là ô màu xanh). Mặc dù phải duyệt qua ít đỉnh hơn song thuật toán A* vẫn sẽ có trường hợp chạy chậm hơn. Đó là bởi A* sử dụng một hàng đợi ưu tiên (priority queue) có độ phức tạp khi thêm và bớt một phần tử là $O(log(n))$ với n là kích thước của hàng đợi. Trong khi đó, BFS sử dụng một hàng đợi (queue) có độ phức tạp khi thêm và bớt một phần tử là $O(1)$.



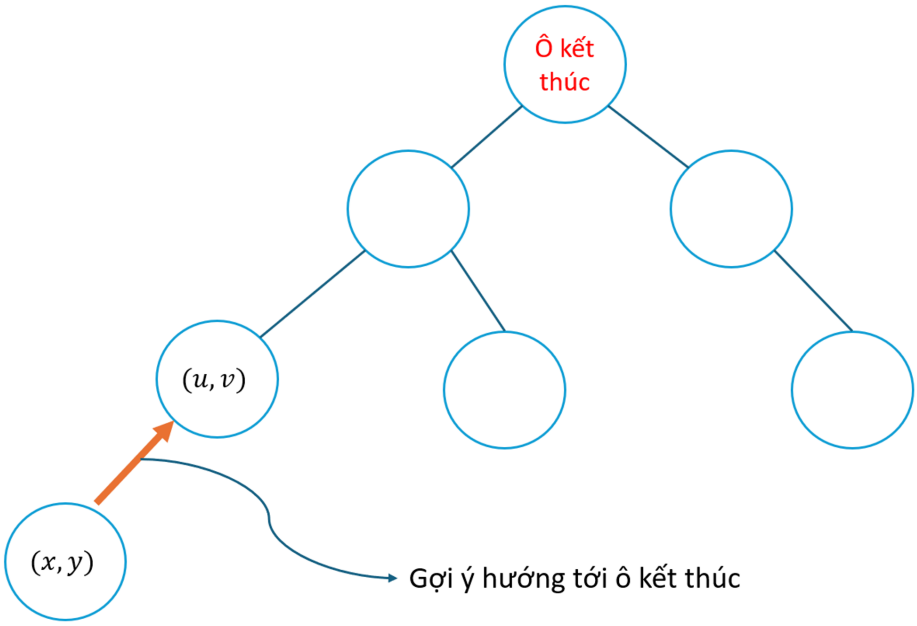
Hình 9: Minh hoạ A* và BFS tìm đường trong mê cung

Về bộ nhớ

Cả BFS và A* đều có độ phức tạp về không gian trong trường hợp tệ nhất là $O(|V|)$ với $|V|$ là số nút trong đồ thị. Tùy vào trường hợp đồ thị phức tạp hay không, bộ nhớ sử dụng của BFS sẽ phụ thuộc vào số nút được lưu giữ vào hàng đợi (queue). Còn đối với thuật toán A* sẽ là phụ thuộc vào hàm heuristic và số nút được đưa vào hàng đợi ưu tiên (priority queue).

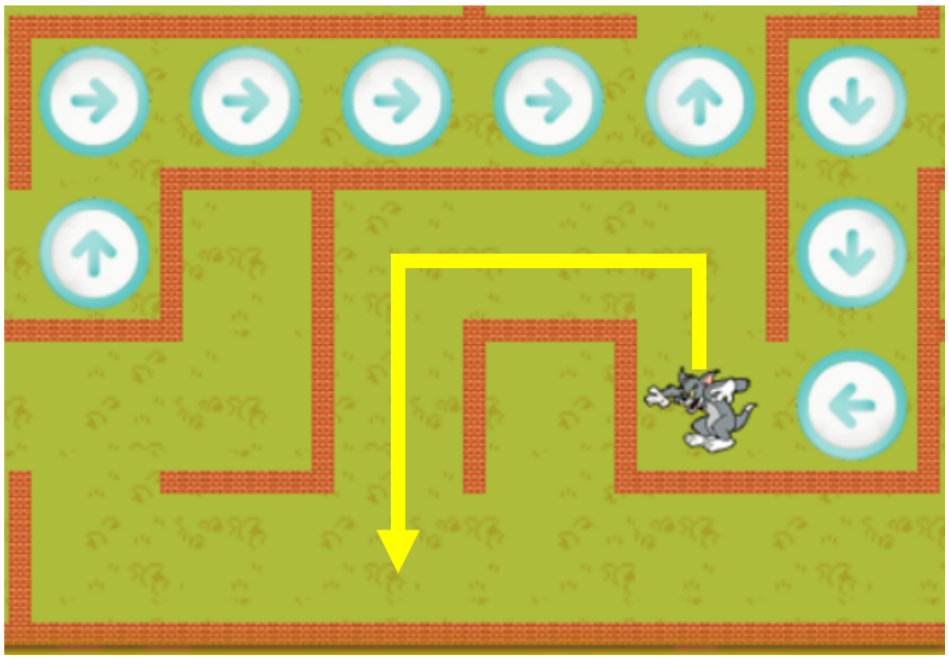
5.5 Hệ thống gợi ý đường đi

Để tạo hệ thống gợi ý, ban đầu sau khi sinh ra một mê cung, chúng tôi đồng thời sử dụng thuật toán BFS để xây dựng một đồ thị mới, bắt đầu duyệt từ ô kết thúc và đi tới tất cả các ô khác trong mê cung. Lúc này, đồ thị nhận được là một cây có nút gốc tương ứng với ô kết thúc của mê cung. Và vì đồ thị là một cây nên tại một nút (x, y) bất kỳ, ta biết rằng nút này có duy nhất một nút cha (u, v). Hướng đi từ ô(x, y) tới ô (u, v) cũng là hướng đi hướng tới ô kết thúc.



Hình 10: Sơ đồ cây gợi ý

Sau khi người chơi ấn nút gợi ý, hệ thống sẽ tự động đưa người chơi đi theo con đường duy nhất hướng tới ô kết thúc cho đến khi gặp một ngã ba hoặc ngã tư. Lúc này, người chơi sẽ có thể tiếp tục tự mình chơi tiếp hoặc có thể ấn nút gợi ý một lần nữa.



Hình 11: Hệ thống sẽ dẫn người chơi đi theo đường màu vàng và dừng ở ngã ba tại đầu mũi tên.

6 Lý thuyết trò chơi: Tối ưu đường đi với năng lượng phát sinh ngẫu nhiên

6.1 Bài toán và các yêu cầu

Bài toán

Bài toán được đặt ra như sau: "Cho trước **số bước đi tối đa** của Tâm là **H**, và **phát sinh ngẫu nhiên K** viên năng lượng trong bản đồ. Tìm đường đi ngắn nhất đến nhà Gia Huy, biết rằng cứ mỗi bước đi sẽ mất 1 năng lượng và ăn được 1 viên năng lượng sẽ có thêm V_k năng lượng với V_k là giá trị phát sinh ngẫu nhiên thuộc {1, 2, 3, 4, 5}. Giá trị của **H** và **K** cũng sẽ tương ứng với các mức độ trò chơi. Hãy trình bày chi tiết đề xuất của bạn về các giá trị này và lý do."

Phân tích yêu cầu

Dễ thấy bài toán có 2 yêu cầu chính:

1. Cho trước giá trị của H và K. **Tìm đường đi ngắn nhất** đến đích.
2. **Tìm giá trị** của **H** và **K** tương ứng với độ lớn của bản đồ (20x20, 50x50, 100x100).

và 2 cách mà viên năng lượng được phát sinh:

1. Tất cả viên năng lượng V_k được phát sinh giá trị **trước khi bắt đầu trò chơi**.
2. Giá trị viên năng lượng V_k được phát sinh **khi chạm đến**.

6.2 Bài toán tìm đường đi ngắn nhất khi V_k được phát sinh khi chạm đến

Đặt tên các giá trị

Với mỗi ô trong mê cung, ta đều có thể tìm được đường đi ngắn nhất đến đích. Gọi số bước của đường đi này là S. Cùng với mỗi ô trong mê cung, ta gọi H_{now} là số bước đi còn lại có thể đi.

Gọi m là số ô năng lượng có thể đi tới. Với $m \geq 1$, gọi P_i $(1 \leq i \leq m)$ là số bước để đi đến ô năng lượng có thể đi tới. P_1 là số bước đi đến ô năng lượng gần nhất

Hướng giải quyết

Trong trường hợp $H_{now} \geq S$ tại ô bất kì, bài toán ngay lập tức được giải.

Ta dễ thấy khi $H_{now} < S$, ta phải tìm cách để đưa $H_{now} \geq S$, điều này chỉ có thể thực hiện khi ta nhận được năng lượng V_k. Do đó, tại các ô năng lượng và ô bắt đầu mới tồn tại khả năng cho $H_{now} \geq S$. Suy ra, ta phải đi đến: hoặc là đích đến (nếu $H_{now} \geq S$), hoặc là một trong các ô năng lượng có thể di chuyển tới được.

Từ vị trí đang đứng, nếu $H_{now} < P_1$ hay $m = 0$ (điều kiện $H_{now} < S$), ta không tìm được đường đi đến đích, bài toán kết thúc. Ngược lại, nếu $H_{now} \geq P_1$ hay $m \geq 1$, ta sẽ **chọn ô năng lượng** `good` ($1 \leq good \leq m$) để di chuyển tới, với $|(H_{now} - P_{good} + 1) - S_{good}|$ **đạt giá trị nhỏ nhất** (tức là càng tiến sát tới mục tiêu tìm được đường đi: $H_{now} \geq S$)

6.3 Bài toán tìm đường đi ngắn nhất khi V_k được phát sinh trước khi bắt đầu trò chơi

6.4 Đề xuất các giá trị H và K

Đề xuất các giá trị H và K

Chúng tôi đề xuất $H = 1,5 \times S$, $K = 0,2 \times level^2$, với S là số bước đi ít nhất để đến đích và $level \in \{20, 50, 100\}$ tương ứng với các mức độ của trò chơi.

Lý do: Người chơi thông thường sẽ không tìm được đường đi tốt nhất để đi trong mê cung. Vì vậy, đường đi thường lớn hơn giá trị S, chúng tôi chọn $1,5 \times S$ và thêm $2 \times level^2$ viên năng lượng - mật độ phân bố phù hợp với mê cung.

7 Hướng dẫn cài đặt, minh hoạ trò chơi

7.1 Hướng dẫn cài đặt

Cài đặt Python và các thư viện

Để trò chơi hoạt động, đầu tiên cần cài đặt Python và các thư viện phụ trợ

- python: phiên bản 3.11.9
- pygame: phiên bản 2.5.2
- pygame-menu: phiên bản 4.4.3

Mở Command Prompt và chạy các lệnh:

```
pip install python==3.11.9
pip install pygame==2.5.2
pip install pygame-menu==4.4.3
```

Cài đặt game

Tải file game từ Github: https://github.com/htrung1105/MazeGame.

Bắt đầu game

Vào thư mục game đã tải, chạy file `main.py`

7.2 Minh hoạ trò chơi

Demo

Video demo trò chơi:

https://www.youtube.com/playlist?list=PLrrNR4O04Hcso0_6GI8eDOB5dsBtBEIS3

Tài liệu

[1] What is Python? Executive Summary.
https://www.python.org/doc/essays/blurb/.

[2] Pygame.
https://www.pygame.org/docs/.

[3] pygame menu.
https://pygame-menu.readthedocs.io/en/latest/index.html.

[4] Randomized Depth-First-Search Algorithm for Maze Generation.
https://medium.com/@nacerkroudir/randomized-depth-first-search-algorithm-for-maze-generation-fb2d83702742.

[5] A* Pathfinding Algorithm.
https://www.baeldung.com/cs/a-star-algorithm.