

BỘ GIÁO DỤC VÀ ĐÀO TẠO

**TRƯỜNG ĐẠI HỌC SÀI GÒN**

****

**BÁO CÁO LAB02**

Bộ môn: Trí tuệ nhân tạo nâng cao.

Giảng viên hướng dẫn: Thầy Đỗ Như Tài

Thành viên nhóm 22:

Lý Chí Nguyên Mssv: 3122410269

Lầu Vỉ Tường Mssv: 3122410459

Contents

[**I. Lab02\_01\_agents.** 2](#_Toc210593659)

[**a) Lunar\_lander.** 2](#_Toc210593660)

[**b) Robot\_vacuum** 8](#_Toc210593661)

[**II. Lab02\_02\_search.** 19](#_Toc210593662)

[**1. Mazes: Compare search behavior (no animation) – So sánh hành vi tìm kiếm (không có animation).** 19](#_Toc210593663)

[**a) Khái niệm Heuristic h(n)** 20](#_Toc210593664)

[**b) Experiments – Thí nghiệm** 21](#_Toc210593665)

[**b) DFS - Depth First Search** 24](#_Toc210593666)

[**c) DFS with no reached data structure** 29](#_Toc210593667)

[**d) GBFS -  Greedy Best-First Search** 31](#_Toc210593668)

[**e) A\* Search** 34](#_Toc210593669)

[**f) So sánh thời gian giữa các thuật toán (BFS, DFS, GBFS, A\*)** 35](#_Toc210593670)

[**2. Mazes: Compare BFS with A-Star Search using the L Maze. (So sánh BFS với A\* Search bằng L Maze)** 37](#_Toc210593671)

[**3. Maze\_example-Animation\_full** 42](#_Toc210593672)

[**4. Explore heuristic.** 56](#_Toc210593673)

[**5. File: Maze.ipynb.** 69](#_Toc210593674)

[**a) BÁO CÁO NHIỆM VỤ 1: ĐỊNH NGHĨA BÀI TOÁN TÌM KIẾM VÀ XÁC ĐỊNH KÍCH THƯỚC BÀI TOÁN.** 69](#_Toc210593675)

[**b) BÁO CÁO NHIỆM VỤ 2: TÌM KIẾM KHÔNG THÔNG TIN THEO CHIỀU RỘNG VÀ CHIỀU SÂU** 70](#_Toc210593676)

[**c) BÁO CÁO NHIỆM VỤ 3:  TÌM KIẾM CÓ THÔNG TIN Triển khai tìm kiếm tham lam theo chiều rộng tốt nhất và tìm kiếm A\*.** 73](#_Toc210593677)

[**d) BÁO CÁO NHIỆM VỤ 4:  SO SÁNH VÀ THẢO LUẬN** 76](#_Toc210593678)

**Nhóm 22**

|  |  |  |
| --- | --- | --- |
| Họ và tên | MSSV | Phân công |
| Lầu Vỉ Tường | 3122410459 | [Solving a maze using a goal-based Agent](https://colab.research.google.com/github/mhahsler/CS7320-AI/blob/master/Search/Maze.ipynb)  Lab02\_02\_agents  Tổng hợp làm ppt. |
| Lý Chí Nguyên | 3122410269 | [Mazes: Show all mazes for the assignment](https://colab.research.google.com/github/mhahsler/CS7320-AI/blob/master/Search/Show_all_mazes.ipynb)  [Mazes: Compare search behavior (no animation)](https://mhahsler.github.io/CS7320-AI/Search/Maze_Example.html).  [Mazes: Compare search behavior with animations](https://mhahsler.github.io/CS7320-AI/Search/Maze_Example-Animation_full.html).  [Maze: The effect of different heuristics](https://colab.research.google.com/github/mhahsler/CS7320-AI/blob/master/Search/Explore_heuristics.ipynb)  [Mazes: Compare BFS with A-Star Search using the L Maze](https://mhahsler.github.io/CS7320-AI/Search/Maze_BFS_vs_A_Star.html). |

# **I. Lab02\_01\_agents.**

# **a) Lunar\_lander.**

\* Notebook làm gì?

- File notebook này giải quyết bài toán Lunar Lander trong môi trường Gymnasium.

- Đây là một bài toán Reinforcement Learning ( LR – học tăng cường) cổ điển.

- Mục tiêu: điều khiển con tàu vũ trụ (lander) hạ cánh an toàn vào vùng được đánh dấu ở giữa màn hình.

- Đặc điểm:

+ Tàu có thể đẩy lực (thrust) về trái, phải hoặc bật động cơ chính.

+ Trọng lực và động lực học làm tàu rơi xuống.

+ Nếu hạ cánh an toàn -> nhận điểm thưởng.

+ Nếu đâm hoặc ra ngoài màn -> bị phạt.

\* Giới thiệu về bài toán Lunar Lander:

- Môi trường: không gian trạng thái liêu lục với tọa độ x và y nằm trong khoảng [-2.5, 2.5] và bãi đáp nằm ở tọa độ (0,0).

- Trạng thái (state): một vector gồm 8 giá trị:

+ Vị trí ngang (x).

+ Vị trí dọc (y).

+ Vận tốc theo trục x.

+ Vận tốc theo trục y.

+ Góc xoay của tàu.

+ Tốc độ góc (angular velocity).

+ Chân trái cso chạm đất không (boolean).

+ Chân phải có chạm đất không (boolean).

- Hành động (action space): gồm 4 hạnh động.

+ 0: không làm gì cả.

+ 1: Bật động cơ phụ bên trái.

+ 2: Bật động cơ chính (main engine).

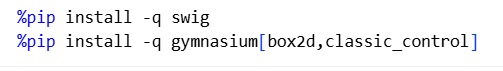
+ 3: Bật động cơ phụ bên phải.

- Phần thưởng (reward): không sử dụng phần thưởng trung gian.

+ Điểm cộng: nếu hạ cánh an toàn +100.

+ Điểm trừ: -100 điểm nếu tàu rơi vỡ.

\* Giải thích code.



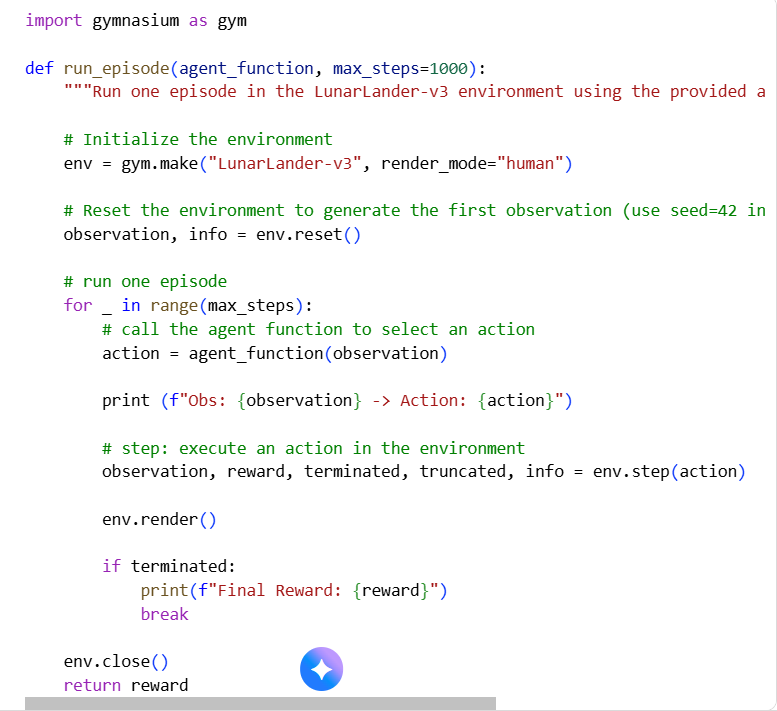
- Cell 1 này là cài đặt các thư viện cần thiết: swing hỗ trợ build code C++ trong Python và gymnasium : cài Gymnasium kèm môi trường Box2D dùng để mô phỏng Lunar Lander.

- Gymnasium environment dùng các hàm chính sau:

+ gym.make(): tạo môi trường.

+ reset(): khởi tạo lại môi trường.

+ step(action): thực hiện hành động và trả về trạng thái mới.



- Cell code này dùng để định nghĩa hàm run\_episode đầu vào là 1 agent\_function và max\_steps được mặc định =1000, thực hiện như sau:

+ Khởi tạo môi trường “LunarLander-v3” với mode là “human” - env.

+ Sau đó lưu trữ trạng thái quan sát – observation - và thông tin -info- bằng hàm env.reset().

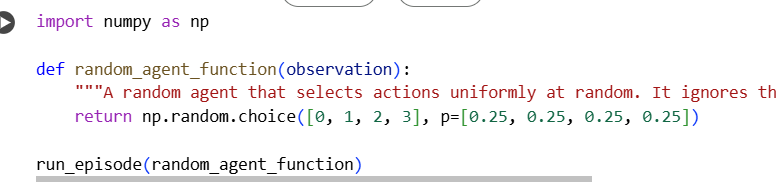
+ Chạy vòng lặp từ 1 đến max\_step thực hiện: gọi agent\_function( observation) để cho agent hành động. Sau đó in ra màn hình với observation này thì sẽ hành động gì. Thực hiện env.step(action) để trả về observation, reward, terminated, truncated,infor mới sau khi thực hiện hành động.

env.render(): hiện thị khung hình (frame) của môi trường để ta quan sát.

terminated là một cờ (flag) để giúp cho ta biết agent đã đạt được thành công hay thất bại. Và trong code với việc kiểm tra teminated= true thì sẽ kết thúc tập và in ra reward cuối cùng.

env.close() đóng môi trường.

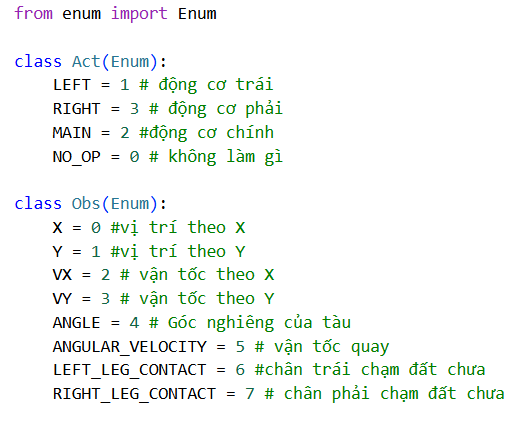
+ Cuối cùng là trả về reward.



- Cell code thứ 3 này định nghĩa một hàm random\_agent\_function đầu vào là observation nhưng không quan tâm đến observation. Hàm này tạo ra một agent ngẫu nhiên bằng cách gọi np.random.choice:

+ Chọn ngẫu nhiên 1 action trong tập [0,1,2,3] với xác suất của mỗi action là 25%.

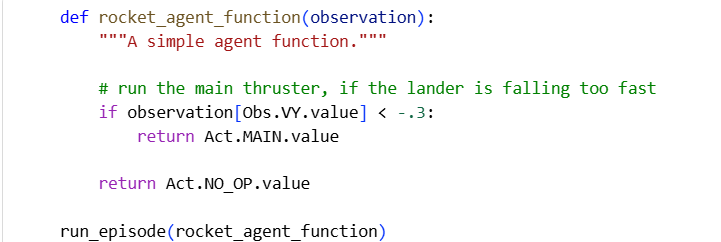
Sau đó là chạy thử agent ngẫu nhiên run\_episode.



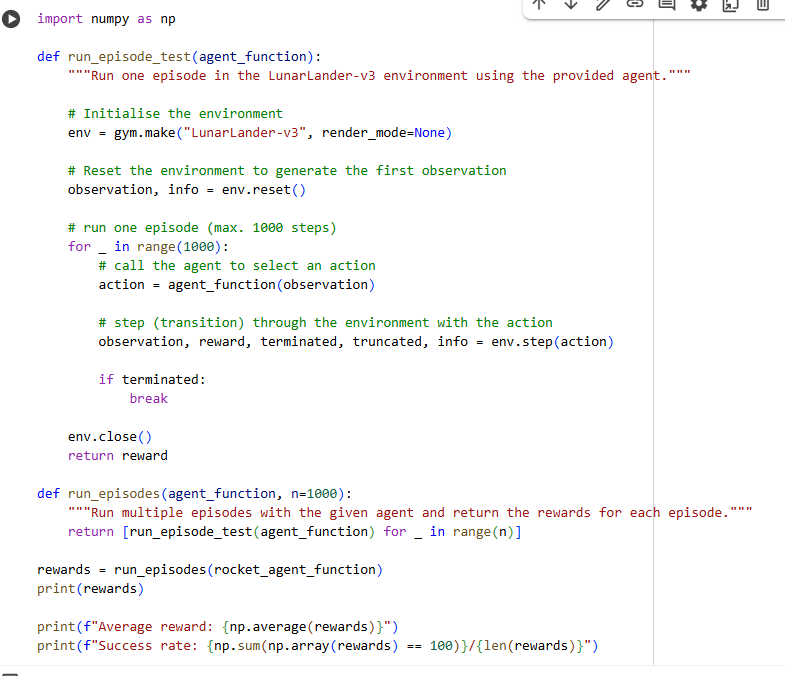
- Cell 4 này khởi tạo 2 class (được comment trên hình) là:

+ Class Act : liệt kê các action trong game (4 hành động).

+ Class Obs: Liệt kê các chỉ số trong observation vector (8 giá trị).



- Code ở cell 5 này thực hiện định nghĩa một agent đơn giản (roket\_agent\_function với đầu vào cũng là observation). Vào trong, sẽ kiểm trả nếu tốc độ theo trục Y mà < -.3 nghĩa là tàu rơi quá nhanh thì sẽ trả về Act.MAIN.value – bật động cơ chính. Còn ngược lại thì trả về Act.NO\_OP.value – không làm gì cả. Sau đó chạy thử agent này run\_episode.

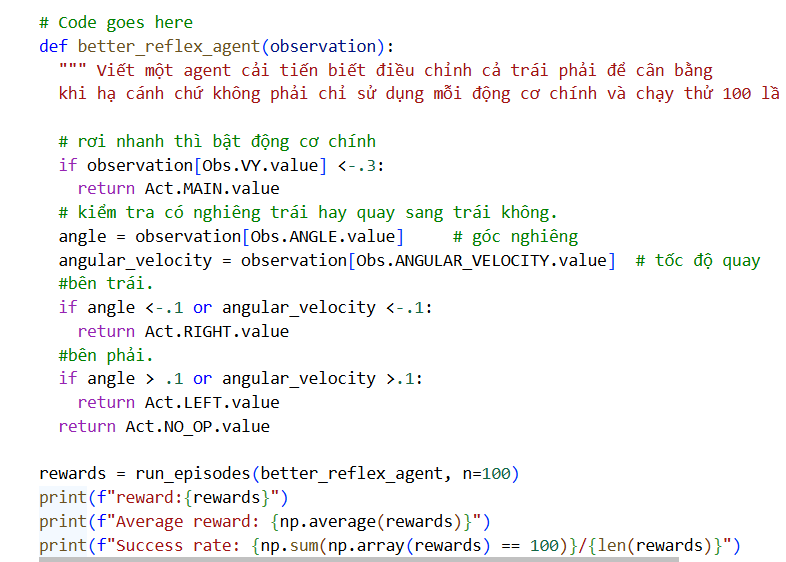


- Cell thứ 6 thực hiện tạo 2 hàm:

+ run\_episode\_test: tương tự với hàm run\_episode ở cell 2 đã nói ở trên thì hàm này cũng thực hiện chạy LunarLander với một agent nhưng khác một chỗ là sẽ không hiện thị giao diện để test nhanh – reder\_mode=None.

+ Hàm run\_episodes: đầu vào là agent\_function và n=1000 số lần chạy. Hàm này sẽ trả về danh sách rewards của agent\_function với n lần chạy thông qua hàm run\_episode\_test.

Sau đó sẽ chạy run\_episodes và in ra list reward, cùng với đó là reward trung bình và tỉ lệ thành công trên list reward.



- Cell thứ 7 này là cell bài tập với yêu cầu xây dựng một agent suy luận tốt hơn thay vì chỉ biết sử dụng mỗi động cơ chính thì agent này phải biết sử dụng cả động cơ bên trái và bên phải ở đây em chọn góc nghiêng và tốc độ xoay để kiểm tra vì:

+ Với góc nghiêng (angle) sẽ cho ta biết con tàu đang nghiêng về phía nào nhiều để mình điều chỉnh lại cho cân bằng.

+ Còn với tốc độ xoay (angular\_velocity) để cho ta biết tàu đang xoay nhanh về hướng nào để mình điểu chỉnh.

Tưởng tự với hàm rocket\_agent\_function ở trên thì em cũng kiểm trả nếu tàu đang rơi quá nhanh thì em sẽ bật động cơ chính. Nếu angle< -0.1 và angular\_velocity < -0.1 thì cho thấy tàu đang nghiêng về bên trái => bật động cơ phải và ngược lại angle > 0.1 và angular\_velocity > 0.1 thì cho thấy tàu đang nghiêng về bên phải => bật động cơ trái. Nếu không trường hợp nào xảy ra thì mình không làm gì.

# **b) Robot\_vacuum**

\* Giới thiệu bài toán:

- Robot vacuum là một bài toán yêu cầu xây dựng một mô hình mô phỏng hoạt động của robot hút bụi trong một môi trường lưới – grid- (size n\*n).

- Mục tiêu: dọn sạch toàn bộ ô có bụi trong môi trường.

- Hoạt động -action- của robot gồm:



+ Các hướng di chuyển north, south, west, east tương đương là lên, xuống, trái và phải.

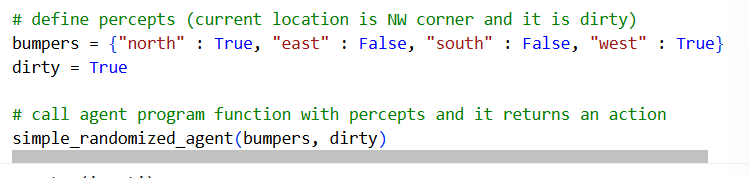
+ Hành động suck là thực hiện hành động hút bụi tại vị trí hiện tại.

- Môi trường có thể tạo ngẫu nhiên vị trí các ô bẩn ( với xác xuất mỗi ô là 0.2) và vị trí bắt đầu của robot hay ở code là agent.

\* Code:



- Code này khởi tạo các hoạt động -actions- của agent có thể có và một hàm đơn giản để agent chọn hành động bằng cách sử dụng chọn ngẫu nhiêu 1 hành động trong actions.

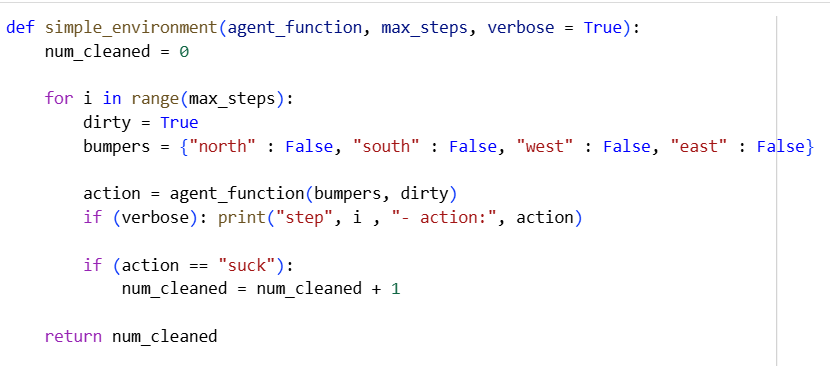


- Như trên code ta thấy:

+ Khởi tạo bumpers, đây là biến lưu trữ thông tin từ cảm biến va chạm với tường của agent. Là một dictionary với 4 khóa “north”, “south”, “west”, “east” và mỗi khóa này sẽ có một giá trị True hoặc False để cho biết agent có va chạm với tường theo hướng đó không.

+ dirty là biến đại diện cho ô đó bẩn hay không với True là có và False là không.

+ Và cuối cùng là chạy thử hàm simple\_randomized\_agent.

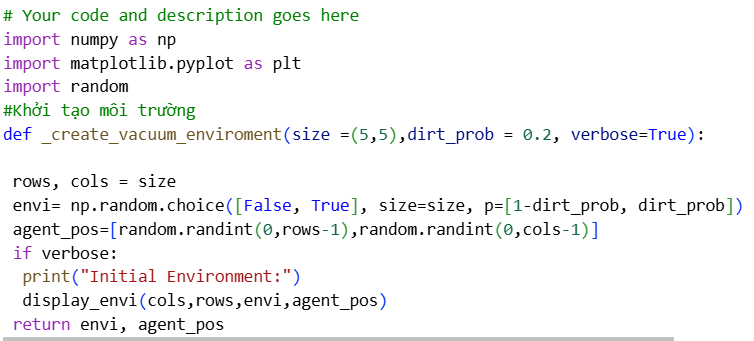


- Code này khởi tạo một môi trường đơn giản để đếm số lần agent có hành động “suck” với đầu vào là một hàm agent -agent\_function , số bước lớn nhất có thể thực hiện – max\_step, và biến verbose là một cờ để in ra màn hình nếu có giá trị là True và ngược lại.

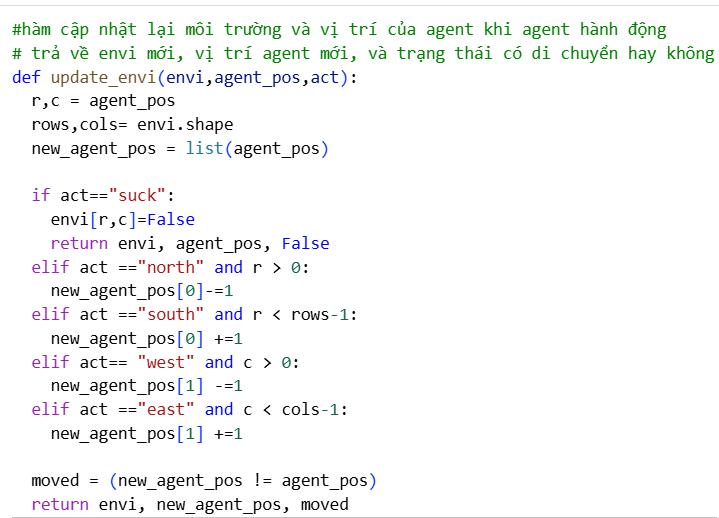
- Trong vòng lặp từ 1 đến hết max\_steps, sẽ khởi tạo dirty và bumpers. Sau đó lấy action từ agent\_function. In ra nếu verbose là True và nếu action là “suck” thì sẽ được cộng dồn – num\_cleaned. Cuối cùng là trả vè num\_cleaned.

- Nhiệm vụ 1: yêu cầu:

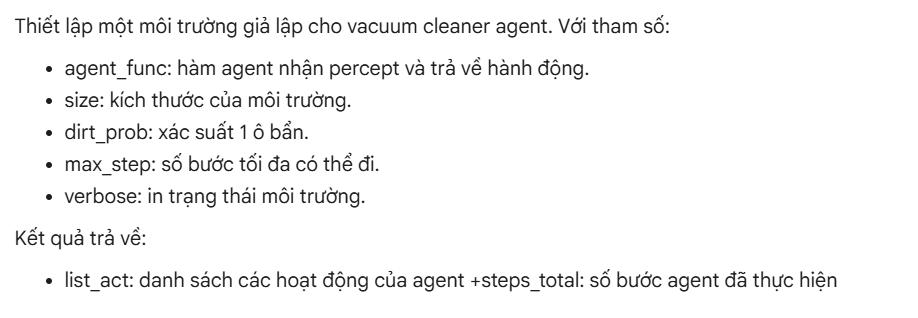
+ Khởi tạo môi trường -hàm \_create\_vacuum\_enviroment- với kích thước n\*n, bụi được đặt ngẫu nhiên trên các ô với xác xuất là 0.2 và agent cũng được đặt một vị trí ngẫu nhiên, code như sau:

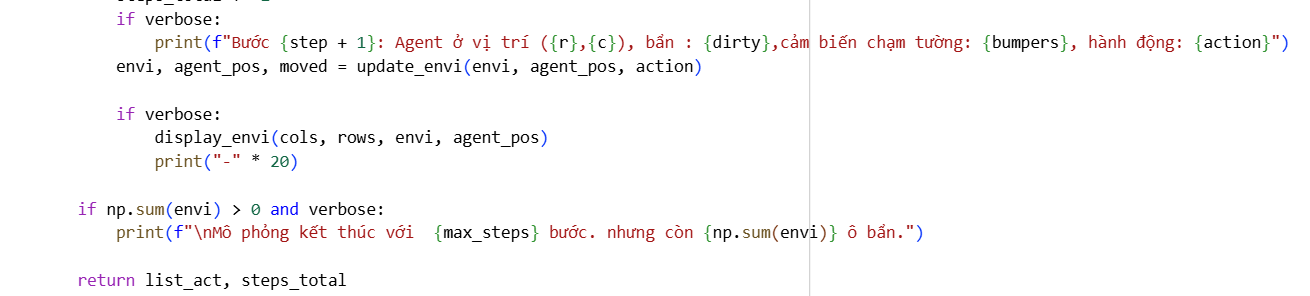


+ Theo dõi vị trí agent -update\_envi- : Hàm này sẽ cập nhật lại vị trí của agent dựa trên hành động của agent.

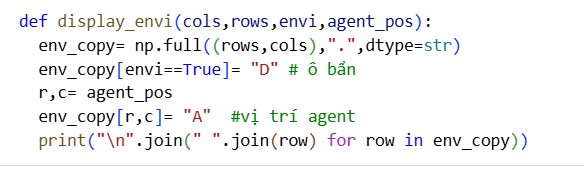


+ Tiếp theo sẽ là một hàm mô phỏng agent trong môi trường -simulation\_envi- : hàm này sẽ trả về cho ta một danh sách hành động của agent -list\_action- và tổng số bước agent đã thực hiện -step\_total-. Chạy từng bước lấy hành động của agent dựa trên hàm agent\_function sau đó hành động này sẽ được thêm vào list\_action và step\_total sẽ được cộng thêm một. Sau mỗi hành động của agent thì môi trường -envi- và vị trí mới của agent -agent\_pos- sẽ được cập nhật lại qua hàm agent -update\_envi .

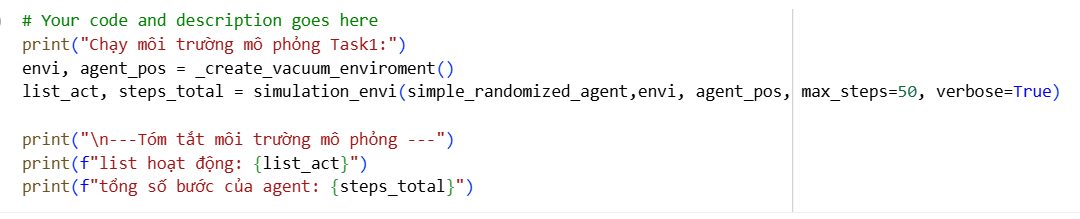




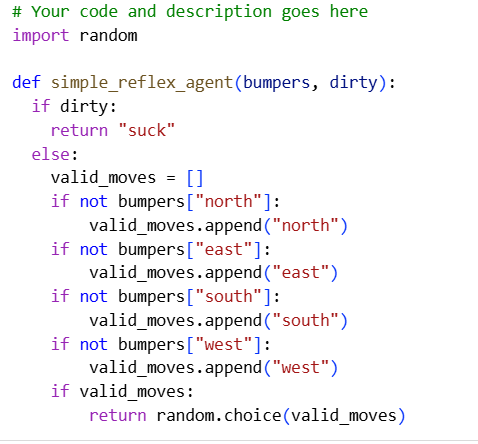
+ Em có thêm một hàm để hiện thi môi trường: với ô bẩn thì em kí hiệu là D tại vị trí đó còn vị trí agent em kí hiệu là A, hàm này giúp em trực quan hóa để quan sát được agent sẽ đi trong môi trường như thế nào.



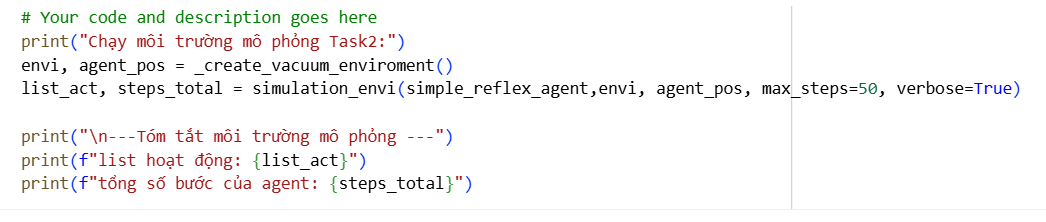
+ Sau khi có các hàm cơ bản rồi thì em thực hiện chạy thử nghiệm mô phỏng, ở đây hàm agent\_function em sử dụng là hàm simple\_randimize\_agent định nghĩa ở trên:



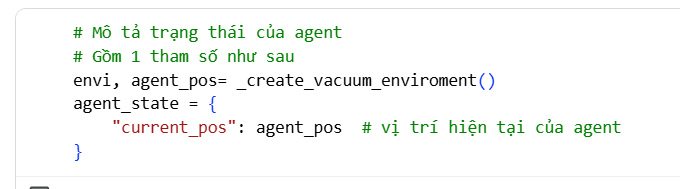
- Nhiệm vụ 2: nhiệm vụ yêu cầu xây dựng lại hàm agent\_function là reflex agent với nếu mà ô hiện tại bẩn thì sẽ trả về hành động "suck" còn ngược lại sẽ kiểm tra các vị trí có chạm tường không nếu không chạm tường sẽ thêm vào trong mảng moves sau đó chọn random các ô có thể đi được:



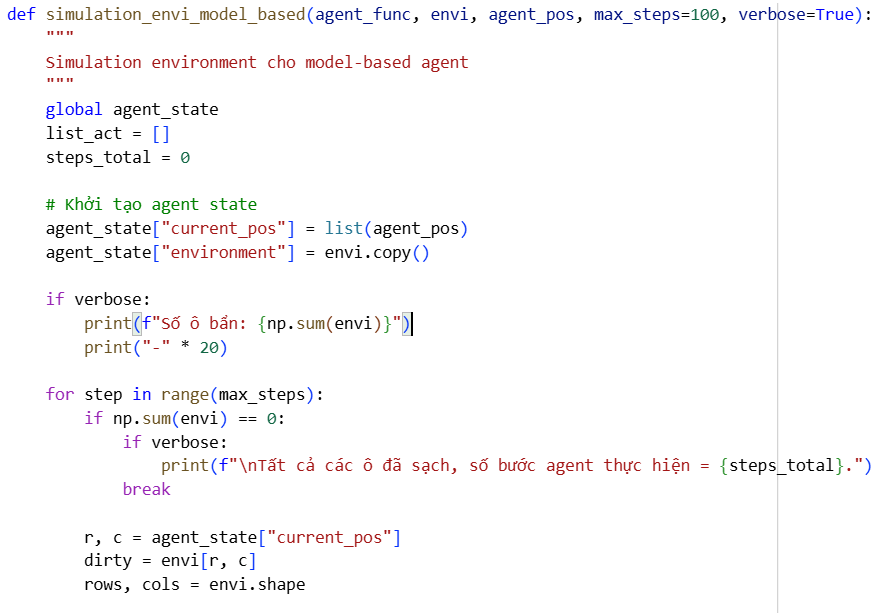
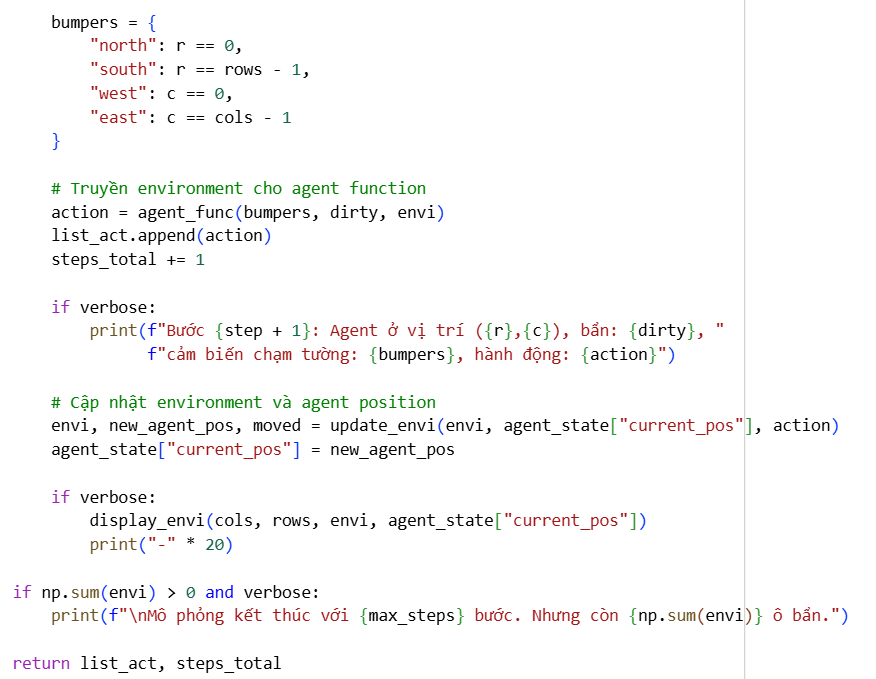
+ Sau đó là em thử nghiệm mô phỏng agent với hàm giả lập – simulation\_envi - ở trên và hàm agent\_function là simple\_reflex\_agent:



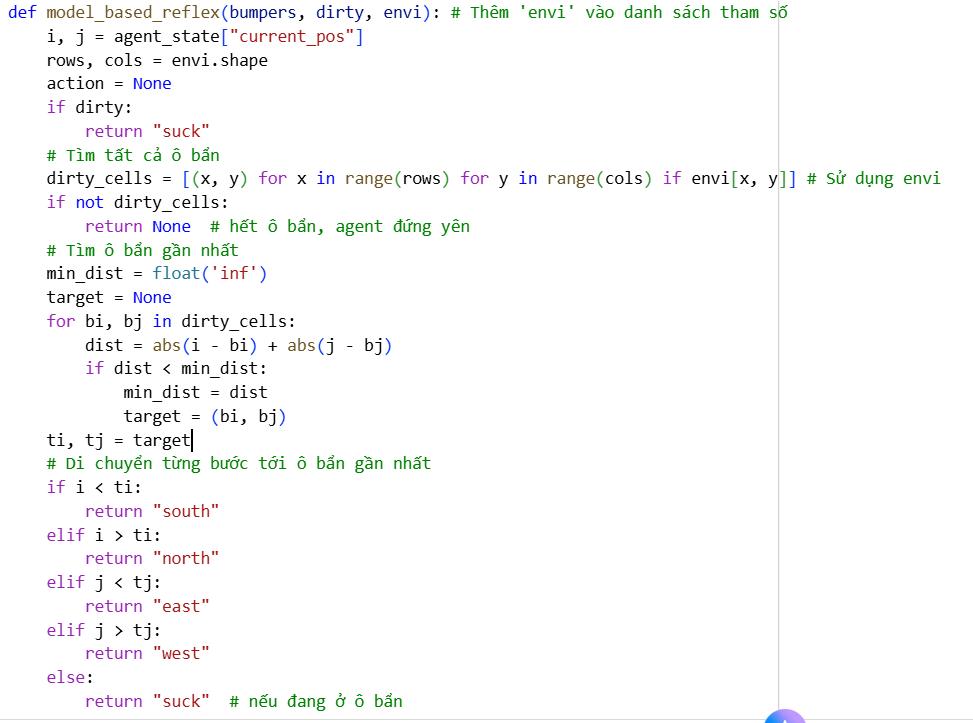
- Nhiệm vụ thứ 3: yêu cầu triển khai một model\_based reflex agent cho robot, nhiệm vụ này yêu cầu sử dụng trạng thái -state- để ghi nhớ những gì agent làm và cảm nhận được cho đến nay để thông qua đó agent sẽ hoạt động tốt hơn. Để agent tốt hơn trong việc dọn dẹp thì em sử dụng chiến lược tim ô bẩn gần nhất với agent và di chuyển tới đo để dọn dẹp “suck” do đó ở trạng thái em chỉ sử dụng 1 tham số là current\_pos lưu trữ vị trí hiện tại của agent.



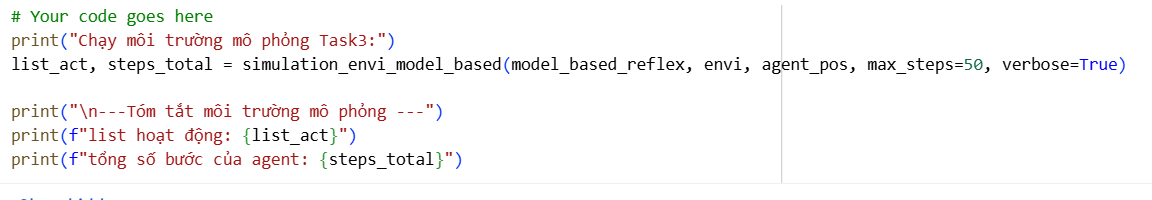
+ Tiếp theo ở hàm simulation\_envi do ở trên không có state nên em cần tinh chỉnh lại là em cập nhật vị trí mới của agent cho state lưu trữ để agent cảm nhận và hành động, hàm có tên simulation\_envi\_model\_based:

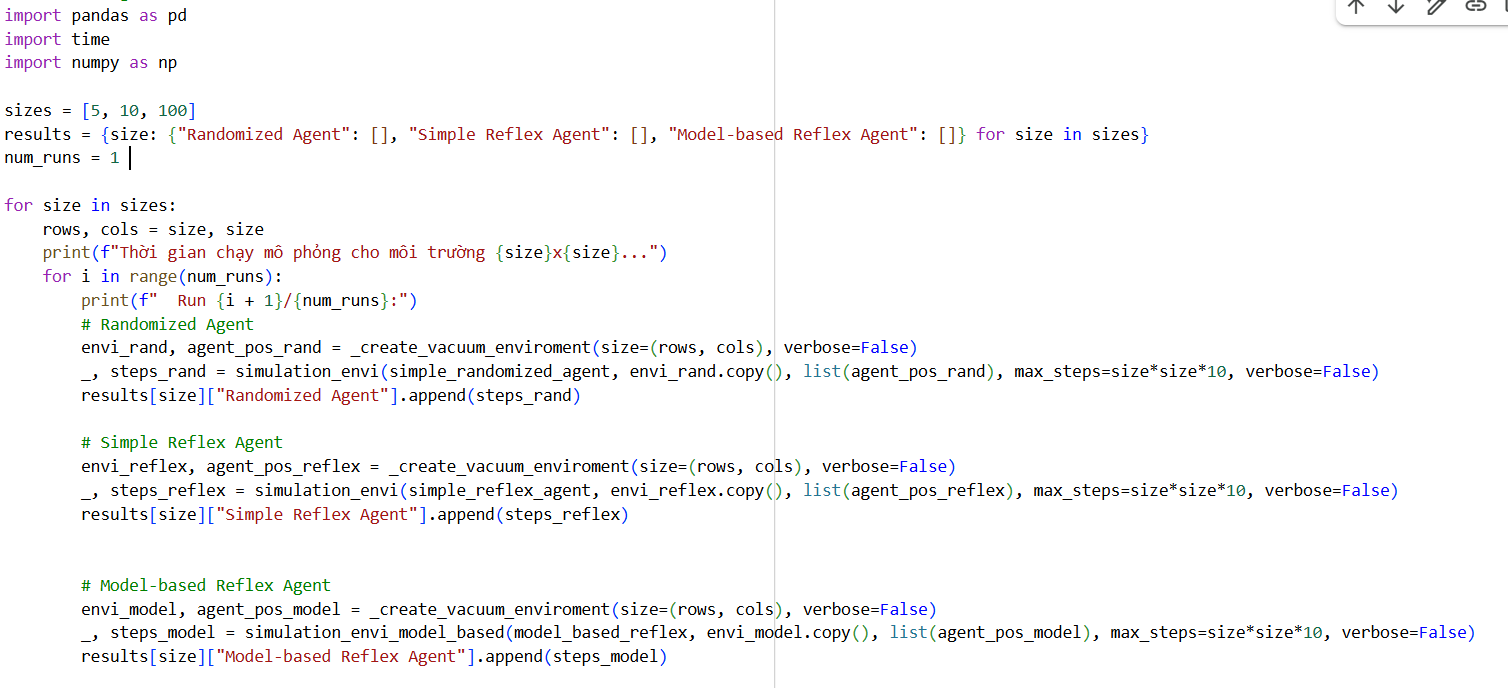
+Tiếp theo em xây dựng hàm model\_based\_reflex: hàm này sẽ tìm vị trí của ô bẩn gần nhất rồi từ đó di chuyển từng bước tới ô bẩn đó và dọn dẹp:

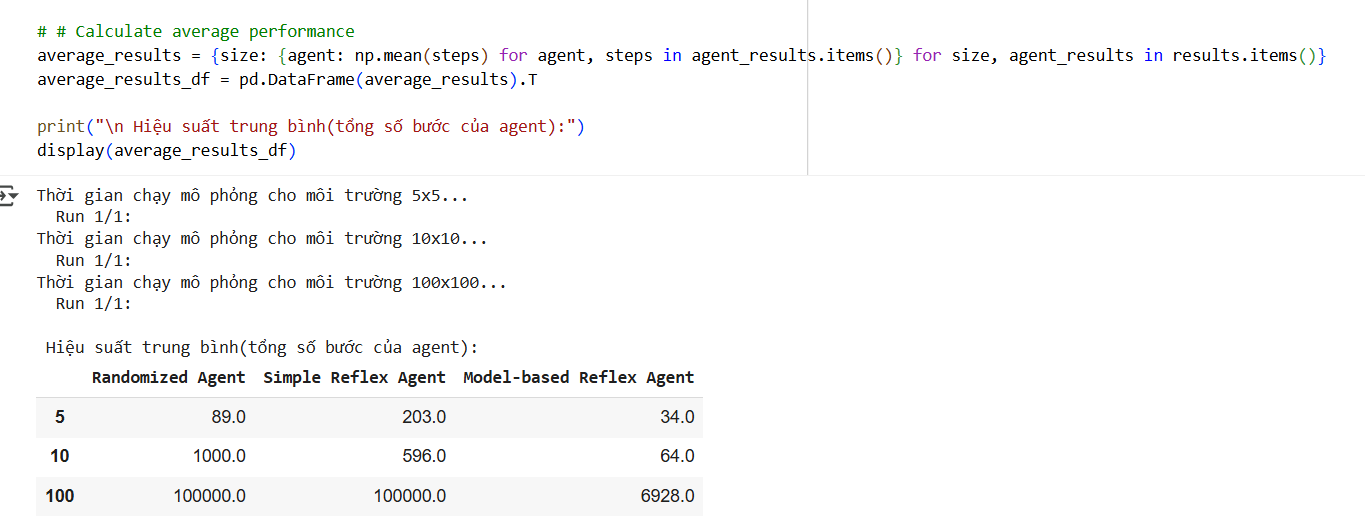


+ Cuối cùng em triển khai mô phỏng với model\_based\_reflex này:



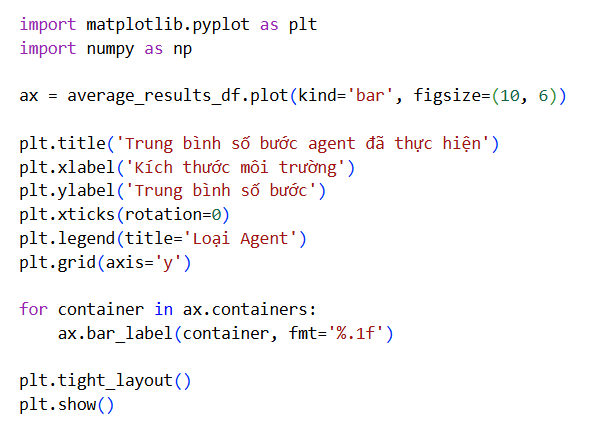
- Nhiệm vụ 4: nhiệm vụ này yêu cầu đánh giá hiện quả của 3 hàm agent\_function theo từng nhiệm vụ 1,2,3 với size của môi trường lần lượt là 5×5, 10×10 và  100×100 với số lần chạy là 100 lần. Do 100 lần chạy rất lâu để có được kêt quả nên code em chỉ chạy đại diện 1 lần:

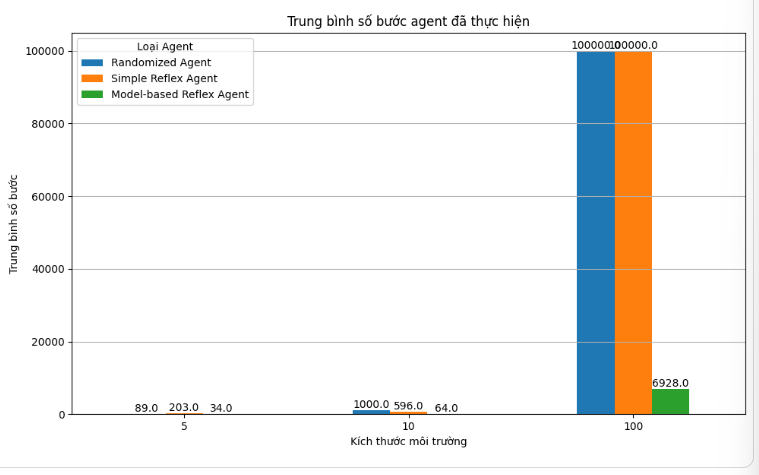




+ Qua bảng kết quả ta thấy model\_based\_reflex hoạt động tốt nhất với số bước thực hiện ít nhất còn simple\_reflex do cũng chọn ngẫu nhiên các hướng đi có thể đi được nên cũng chỉ tốt hơn một xíu so với simple\_randomized\_agent.

+ Tiếp theo là sử dụng thư viện matplotlib để trực quan hóa kết quả sang dạng biểu đồ.



  
 - Nhiệm vụ 5: mô tả hiệu suất của các agent\_funtion trong 5 trường hợp:

+ Nếu được đặt vào phòng hình chữ nhật có kích thước không xác định.

+ Nếu khu vực dọn dẹp có hình dạng bất thường.

+ Nếu phòng chứa chướng ngại vật.

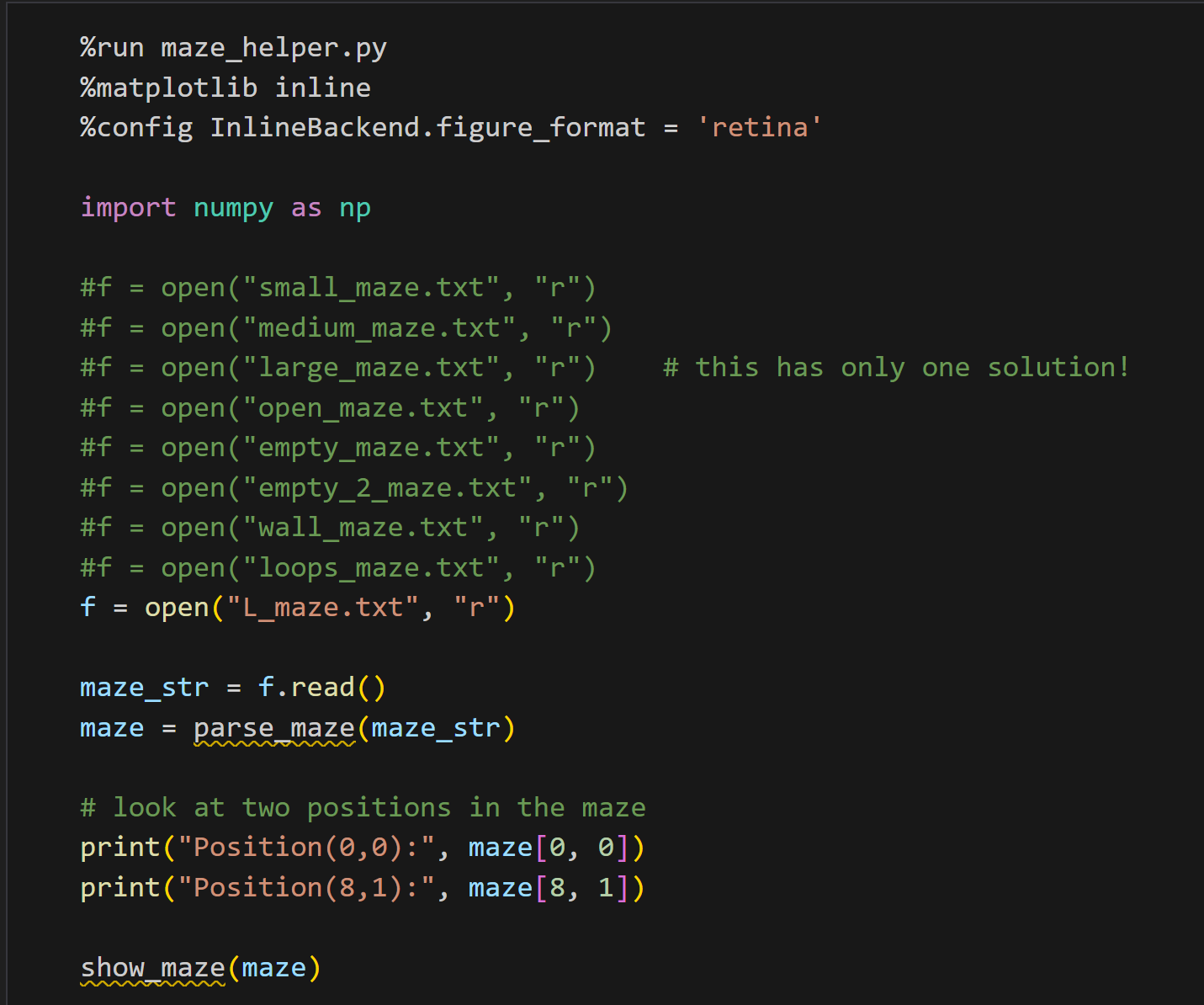
+ Nếu cảm biến bụi không hoàn hảo (10% sai).

+ Nếu cảm biến va chạm không hoàn hảo (10% không báo tường).

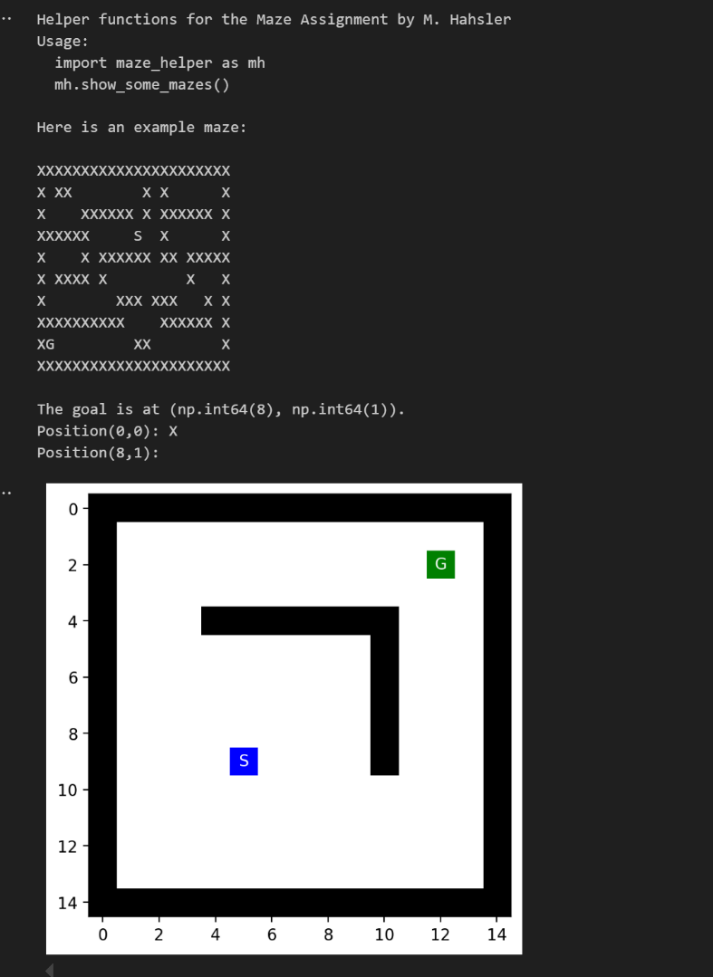
Em đã trả lời chi tiết trong notebook.

# **II. Lab02\_02\_search.**

# **1. Mazes: Compare search behavior (no animation) – So sánh hành vi tìm kiếm (không có animation).**



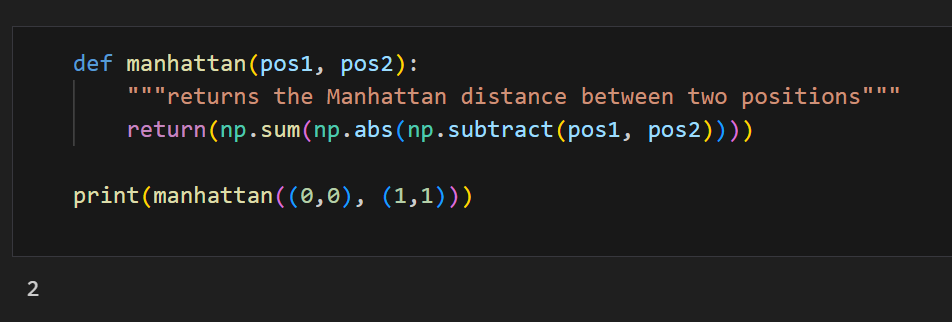
* Khởi chạy file maze\_helper.py dùng matplotlib trực tiếp trong notebook để vẽ và figure\_format = 'retina' tăng độ nét của hình vẽ hiển thị.
* Tiến hành mở và đọc file văn bản “L\_maze.txt”. Sau đó lấy toàn bộ nội dung file dưới dạng chuỗi ký tự và dùng hàm parse\_maze trong maze\_helper.py để chuyển chuỗi đó thành mảng 2D numpy. Ví dụ: ‘X’ = tường; ‘ ‘ (space) = đường trống; ‘S’ = start; ‘G’ = goal.
* Kiểm tra ngẫu nhiên 2 vị trí bất kì và in ra. Tiến hành vẽ mê cung thông qua hàm show\_maze(maze), hàm này dùng matplotlib hiển thị mê cung như một hình ảnh trực quan: tường có màu in đậm, đường đi có màu sáng hoặc trắng.



# **a) Khái niệm Heuristic h(n)**

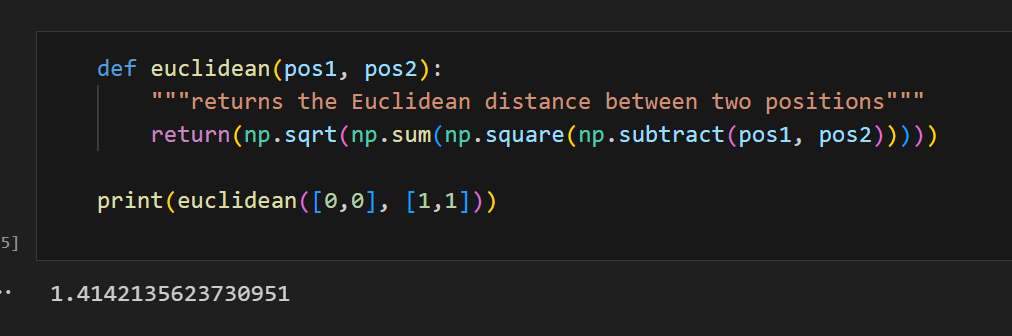
Trong tìm kiếm đường đi (A\*, Greedy Best-First,…), h(n) là hàm ước lượng chi phí từ nút hiện tại n đến mục tiêu.  
Hai loại heuristic thường gặp:

\* Manhattan distance: thích hợp khi chỉ đi lên-xuống-trái-phải.



* Hàm Manhattan dùng pos1, pos2 để lấy toạ độ dạng tuple/list. Sau đó, tính vector chênh lệch thông qua np.subtract(pos1, pos2), dùng np.abs lấy trị tuyệt đối trước khi tính tổng các bước. Ví dụ: di chuyển từ vị trí (0,0) đến vị trí (1,1) cần 1 bước ngang + 1 bước dọc kết quả = 2.

**\* Euclidean distance**: khoảng cách đường thẳng.

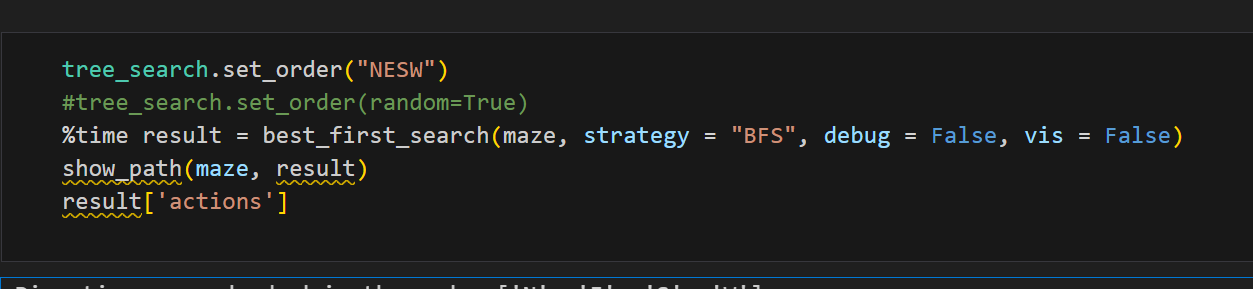


* Hàm Eucliden tương tự hàm Manhattan cách toạ độ và tính vector chênh lệch nhưng dùng np.square lấy bình phương từng thành phần, sau đó tính tổng trước khi lấy căn bậc hai 2 vị trí.

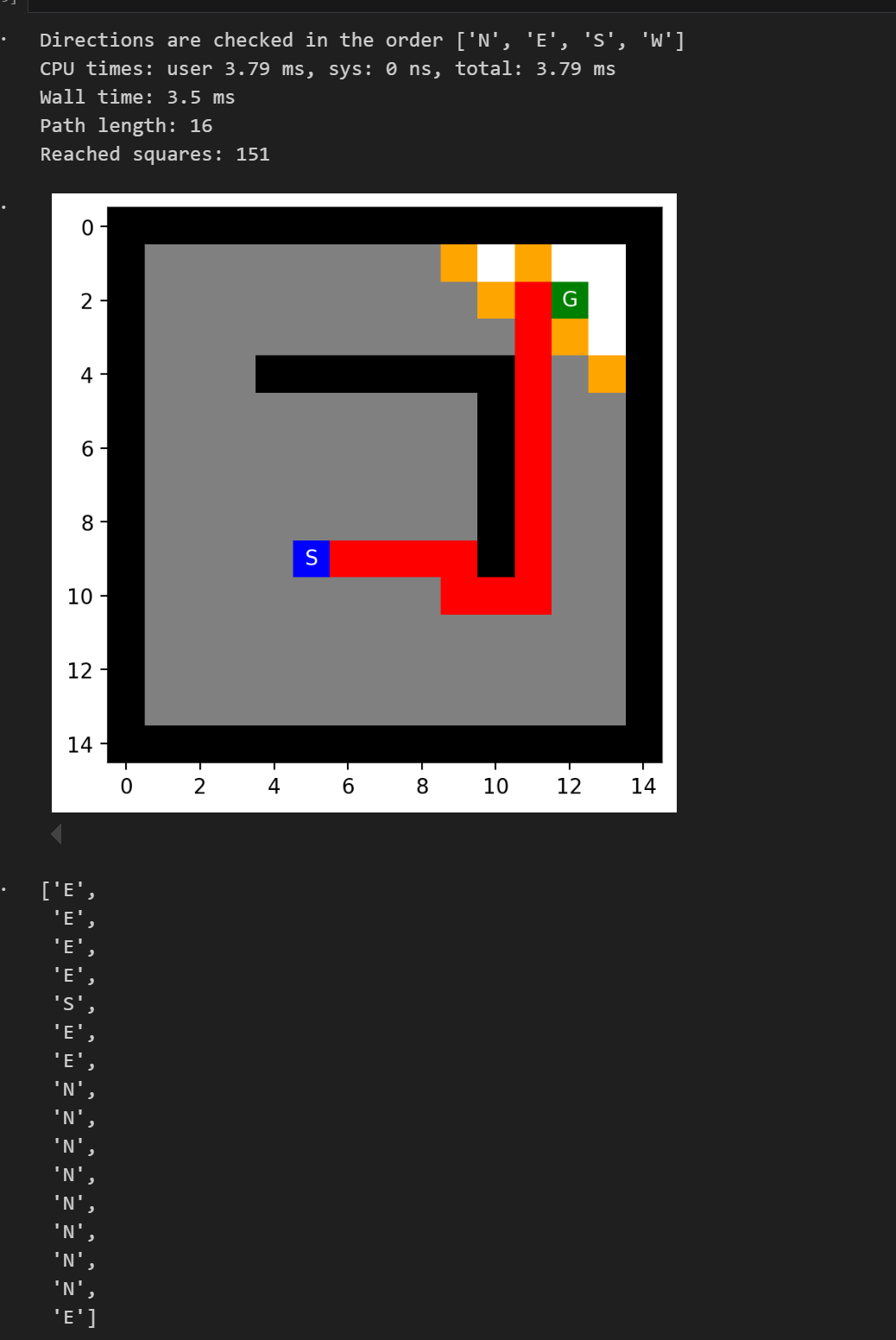
# **b) Experiments – Thí nghiệm**

**\*BFS - Breadth-First Search**

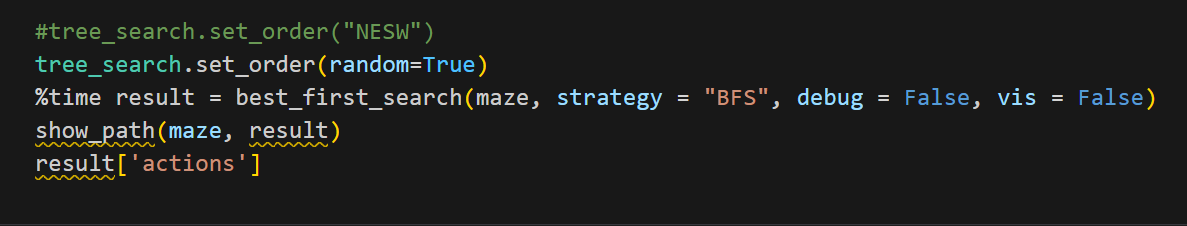
* Cách 1: Duyệt theo hướng “NESW”



* Sử dụng thuật toán BFS duyệt có hướng **“**NESW**”** các ô xung quanh từ Bắc 🡪 Đông 🡪 Nam 🡪 Tây.
* Chạy thuật toán BFS để tìm đường đi trong mê cung. Kết quả “result” là một dict chứa thứ tự thực hiện các bước, thống kê thời gian CPU mà Jupyter %time đo được (CPU time), thời gian thực tế (Wall time), độ dài đường đi tìm được (Path length), tổng số ô (nút) BFS đã duyệt qua trong mê cung trước khi tìm thấy mục tiêu (Reached squares) – con số này thể hiện độ rộng của việc tìm kiếm: càng nhiều ô được mở rộng, càng tốn bộ nhớ. Hiển thị mê cung và tô màu đường đi từ start ‘S’ đến goal ‘G’. Trả về danh sách hành động đi từ điểm bắt đầu đến kết thúc thông qua result[‘action’].

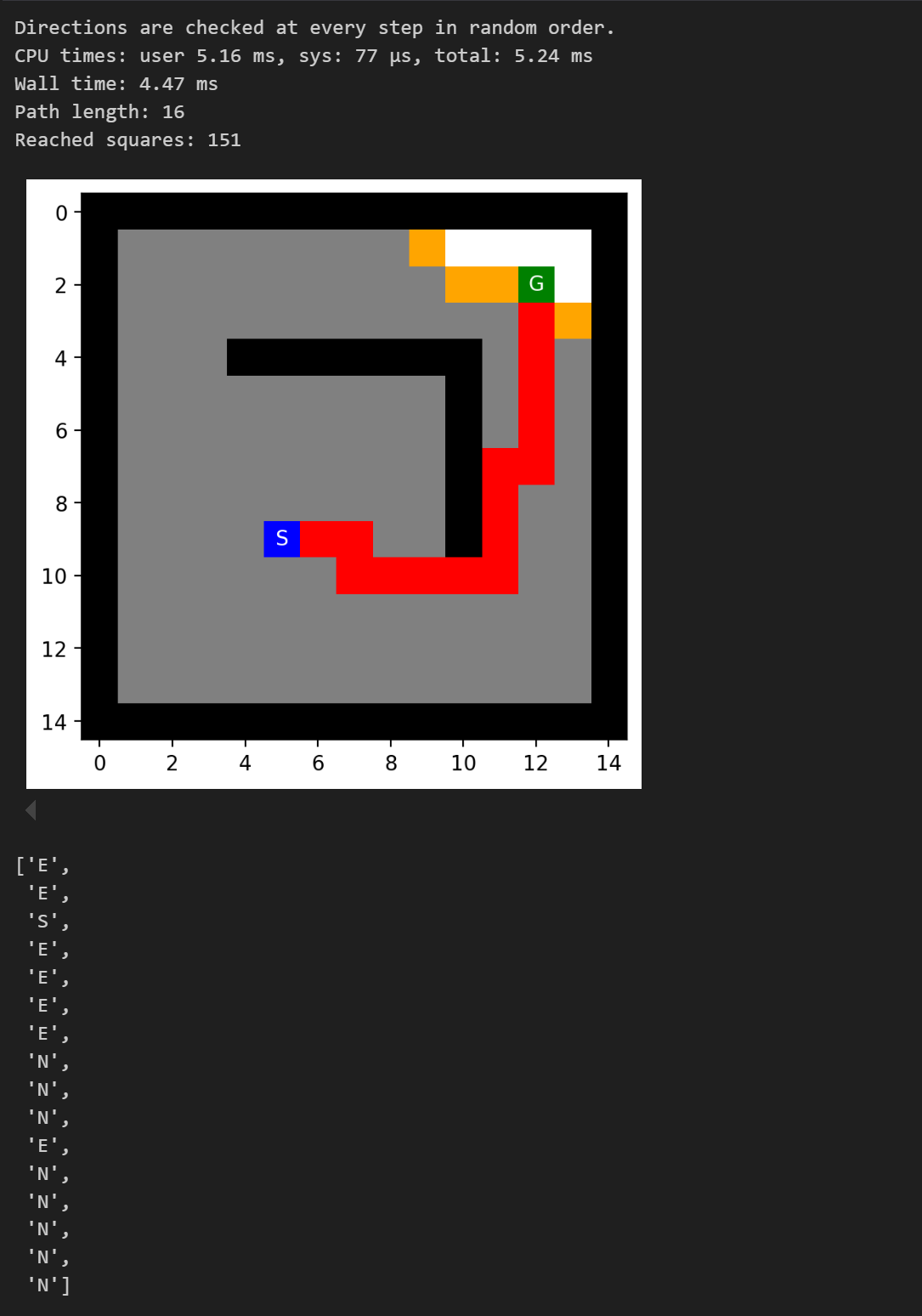


* Kết quả cho thấy cách duyệt có hướng bằng thuật toán BFS tìm được 1 đường dài 16 bước sau khi kiểm tra 151 ô. Toàn bộ quá trình mất khoảng 3-4 mili giây.
* Cách 2: duyệt theo hướng ngẫu nhiên

****

* Sử dụng thuật toán BFS duyệt theo các hướng ngẫu nhiên random=True. Mỗi lần mở rộng một ô, moudle tree\_search sẽ xáo trộn thứ tự các hướng (N, E, S, W) ngẫu nhiên để quan sát xem BFS vẫn tìm được đường đi dù không có thứ tự cố định.
* Tiến hành tìm kiếm và hiển thị thời gian cũng như số bước đã thực hiện được như khi duyệt có hướng cố định. Vẽ mê cung và hiển thị kết quả các hành động **result[‘action’]** khi di chuyển tìm kiếm từ điểm bắt đầu đến mục tiêu.

\*\* Lưu ý: do hướng được duyệt ngẫu nhiên nên kết quả có thể khác nhau sau mỗi lần chạy, nhưng số bước vẫn tối thiểu vì BFS luôn tìm đường đi ngắn nhất.

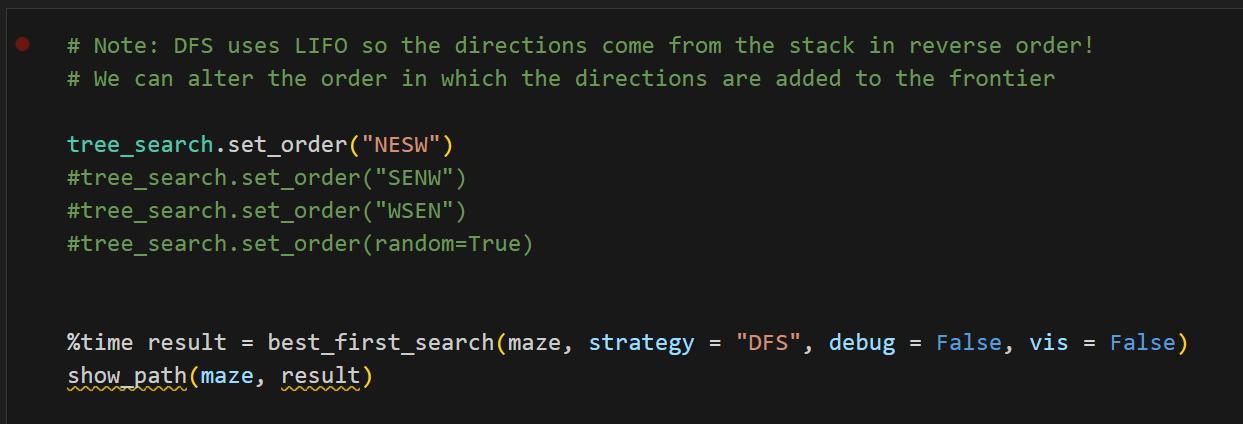
****

* Kết quả cho thấy thuật toán BFS duyệt theo hướng ngẫu nhiên cũng tìm được 1 đường ngắn nhất dài 16 bước sau khi duyệt qua 151 ô. Quá trình mất khoảng 4 đến hơn 5 mili giây.

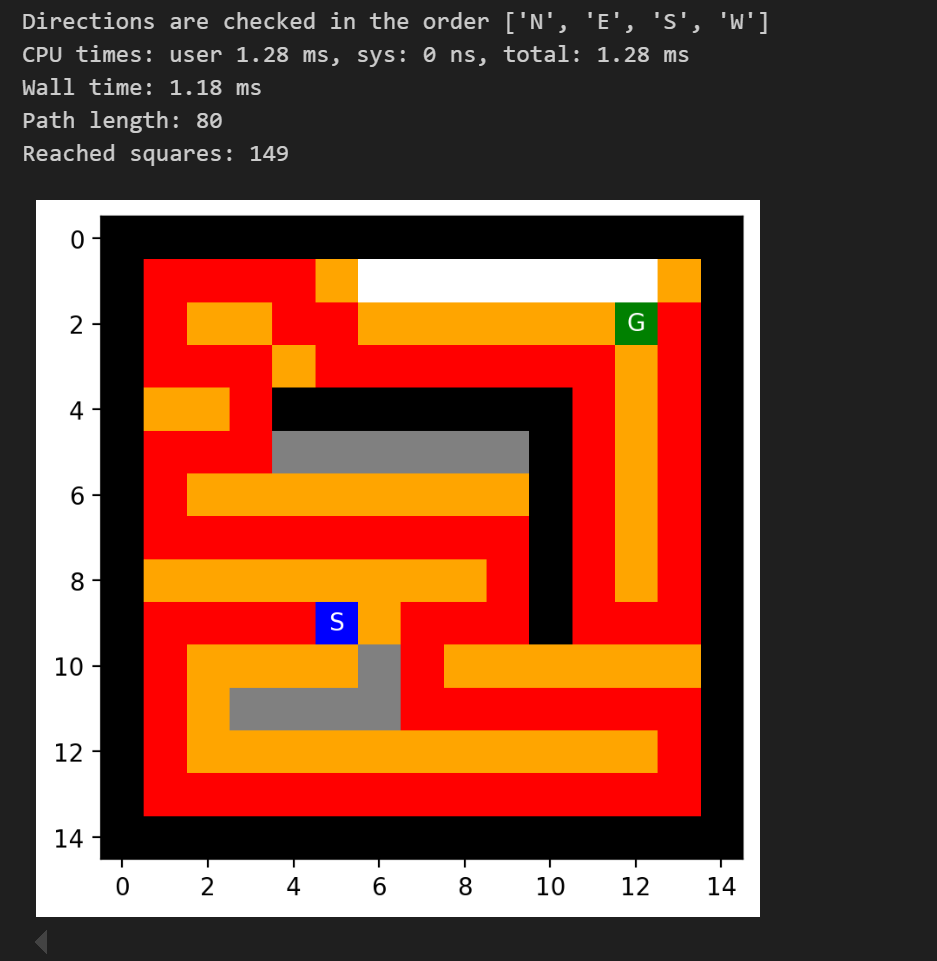
# **b) DFS - Depth First Search**

Thuật toán có độ phức tạp là O(bm) với m là độ dài đường đi dài nhất trong cây tìm kiếm, nó tệ hơn thuật toán BFS có độ phức tạp là O(bd) với d là độ sâu gần nhất của mục tiêu.

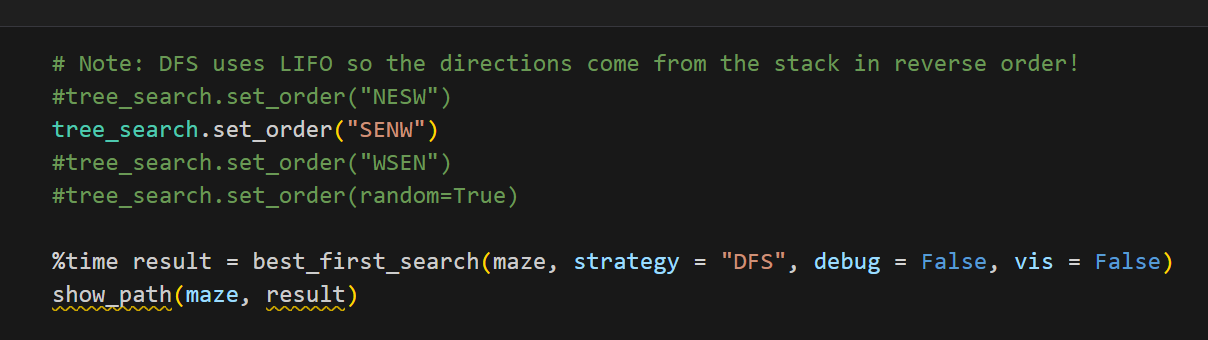
* **Cách 1: duyệt theo hướng “NESW”**

****

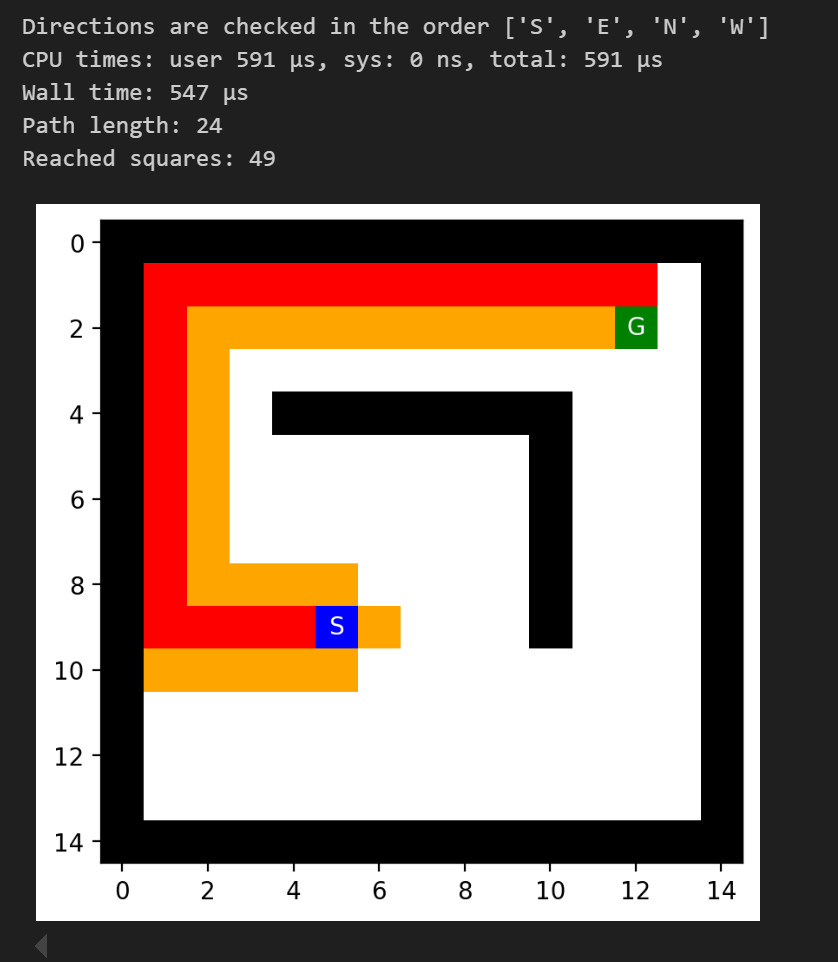
* Vì DFS sử dụng LIFO - (Last-In-First-Out) nên các hướng từ ngăn danh sách sẽ theo thứ tự ngược lại. Ở cách 1 duyệt theo hướng Bắc 🡪 Đông 🡪 Nam 🡪 Tây nhưng thực tế sẽ ngược lại là Tây 🡪 Nam 🡪 Đông 🡪 Bắc.
* Sử dụng %time để đo thời gian thực thi, best\_first\_search được tái sử dụng nhưng **strategy= “DFS”** khiến nó hoạt động theo DFS. Sau đó tiến hành vẽ mê cung và đánh dấu đường đi tìm được.

****

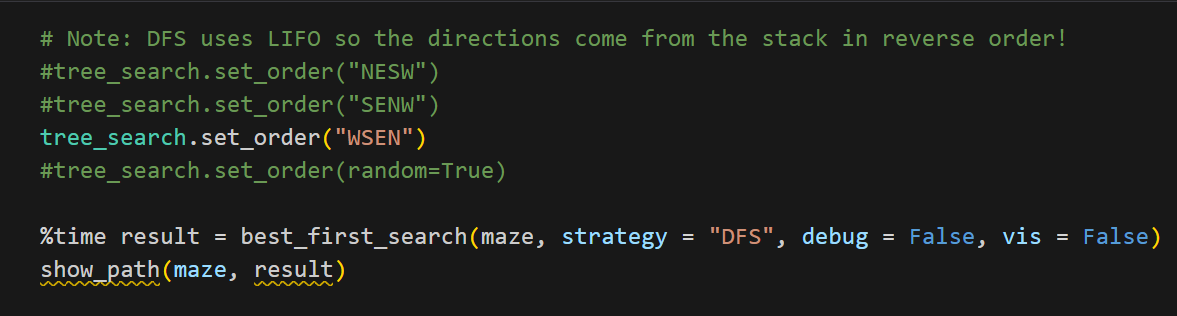
* Kết quả hiển thị thuật toán DFS duyệt theo hướng “NESW” chỉ mất khoảng hơn 1 mili giây để duyệt qua 149 ô và tìm ra đường đi dài tới 80 bước từ điểm bắt đầu đến mục tiêu.
* **Cách 2: duyệt theo hướng “SENW”**

****

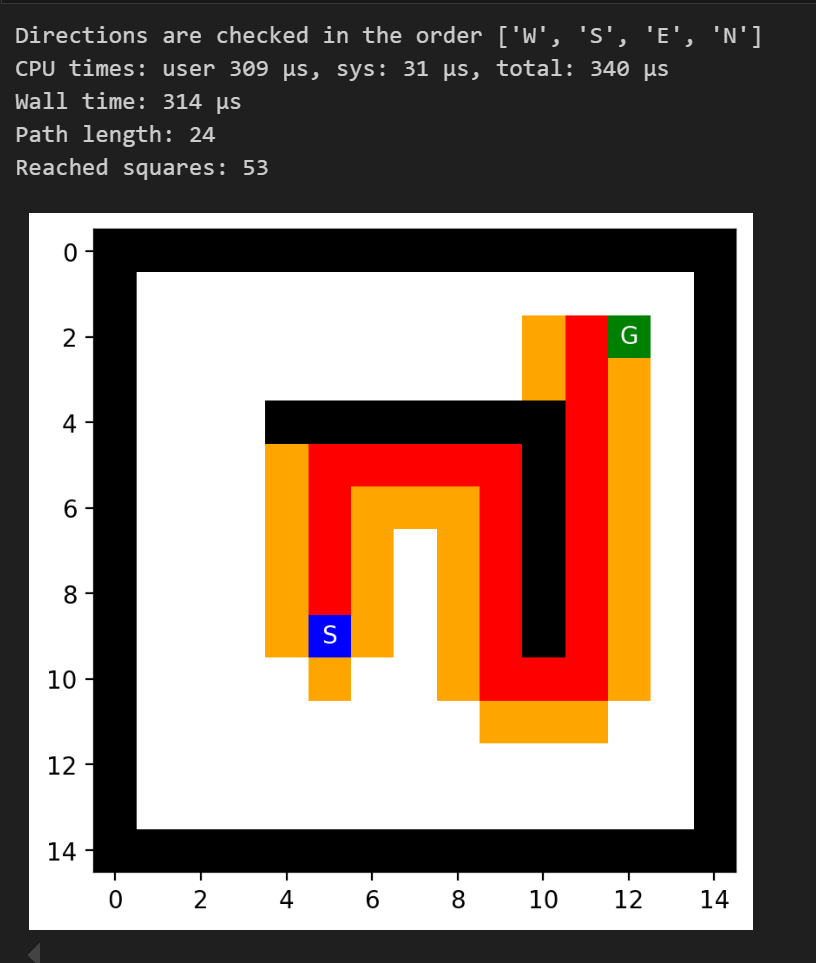
* Ở cách 2 duyệt theo hướng Nam 🡪 Đông 🡪 Bắc 🡪 Tây nhưng thực tế sẽ ngược lại là Tây 🡪 Bắc 🡪 Đông 🡪 Nam.
* Sử dụng %time để đo thời gian thực thi, best\_first\_search được tái sử dụng nhưng **strategy= “DFS”** khiến nó hoạt động theo DFS. Sau đó tiến hành vẽ mê cung và đánh dấu đường đi tìm được.

****

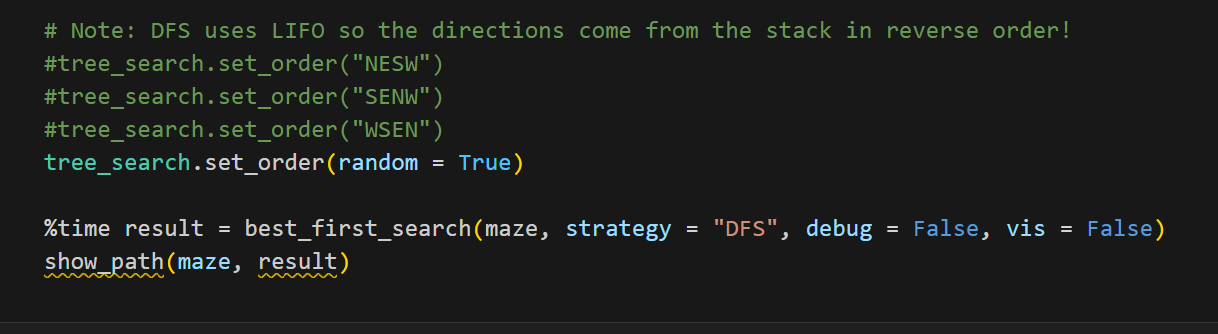
* Kết quả hiển thị thuật toán DFS duyệt theo hướng “SENW” chỉ mất khoảng ~ 0,5 mili giây (~ 500 micro giây) để duyệt qua 49 ô và tìm ra đường đi dài 24 bước từ điểm bắt đầu đến mục tiêu.
* **Cách 3: duyệt theo hướng “WSEN”**

****

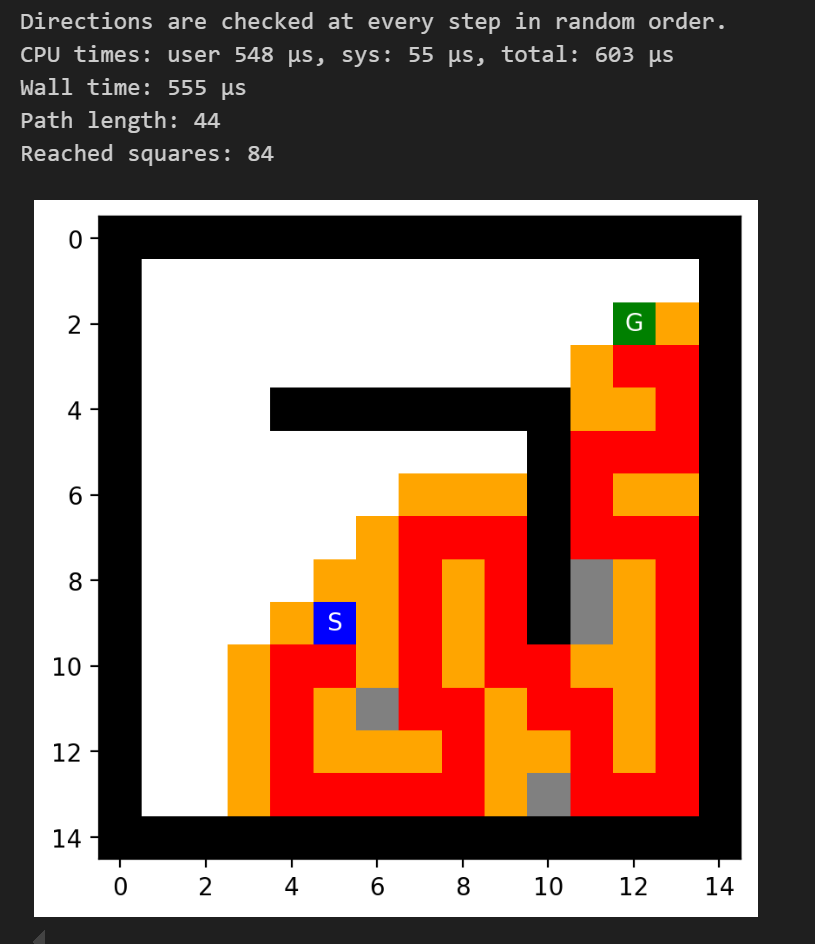
* Ở cách 3 duyệt theo hướng Tây 🡪 Nam 🡪 Đông 🡪 Bắc nhưng thực tế sẽ ngược lại là Bắc 🡪 Đông 🡪 Nam 🡪 Tây.
* Sử dụng %time để đo thời gian thực thi, best\_first\_search được tái sử dụng nhưng **strategy= “DFS”** khiến nó hoạt động theo DFS. Sau đó tiến hành vẽ mê cung và đánh dấu đường đi tìm được.

****

* Kết quả hiển thị thuật toán DFS duyệt theo hướng “SENW” chỉ mất khoảng ~ 0,3 mili giây (~ 300 micro giây) để duyệt qua 53 ô và tìm ra đường đi dài 24 bước từ điểm bắt đầu đến mục tiêu. Điều này tiếp tục cho thấy **thứ tự ưu tiên hướng đi ảnh hưởng lớn** đến hiệu quả của DFS: cùng một mê cung nhưng số ô duyệt và đường đi có thể khác đáng kể.
* **Cách 4: duyệt theo hướng ngẫu nhiên**

****

* Ở cách 4 mỗi khi mở rộng một ô, các hướng **N, E, S, W sẽ được xáo trộn ngẫu nhiên**. Vì DFS dùng **LIFO (stack)**, thứ tự bạn thêm vào vốn đã ngẫu nhiên, và khi lấy ra còn phụ thuộc ngược lại => đường đi sẽ **khó đoán** mỗi lần chạy.
* Sử dụng %time để đo thời gian thực thi, best\_first\_search được tái sử dụng nhưng **strategy= “DFS”** khiến nó hoạt động theo DFS. Sau đó tiến hành vẽ mê cung và đánh dấu đường đi tìm được.

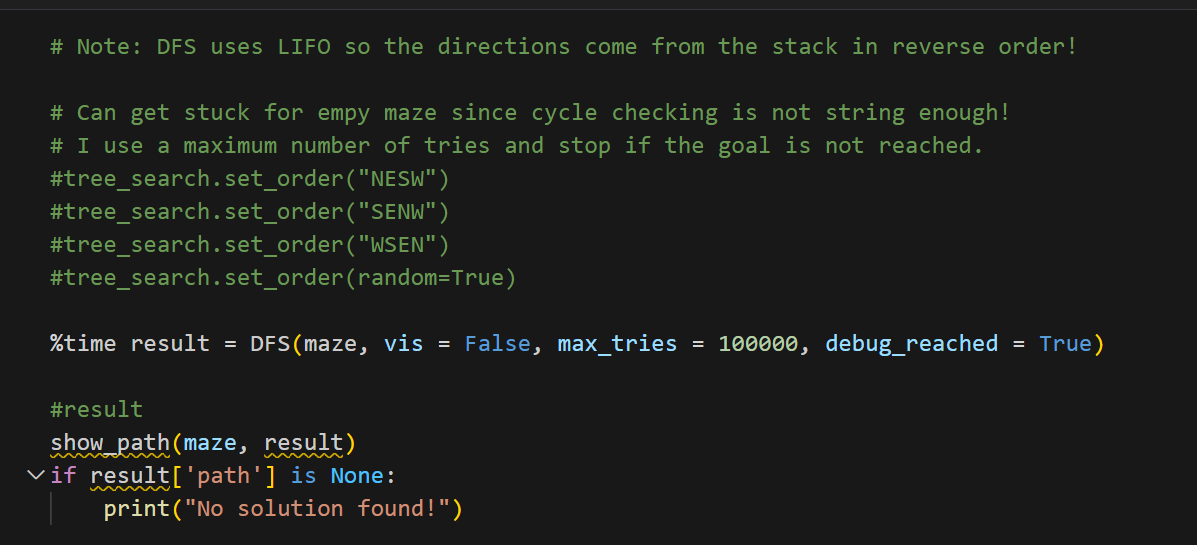
****

* Kết quả hiển thị thuật toán DFS duyệt theo hướng ngẫu nhiên mất khoảng 0,5 – 0,6 mili giây (khoảng 500 – 600 micro giây) để duyệt qua 84 ô và tìm dược đường dài 44 bước. Điều này minh họa rõ ràng rằng với DFS, thứ tự mở rộng hướng đi ảnh hưởng mạnh đến cả số ô duyệt và độ dài đường đi. (kết quả có thể khác nahu sau những lần chạy thử)

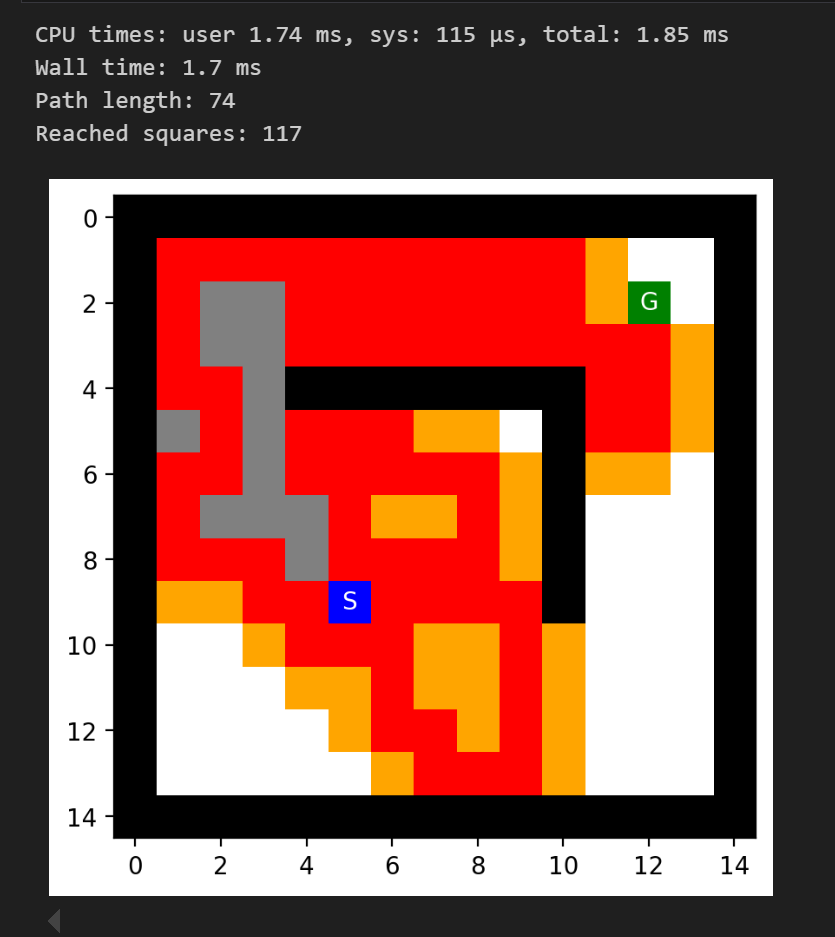
# **c) DFS with no reached data structure**

Giải thuật này có độ phức tạp là O(bm) thay vì O(bm) (như khi ta tái sử dụng thuật toán tree search của BFS). Vì phiên bản “no reached” không lưu lại nên không phát hiện được chu trình, không thể kiểm tra tất cả các chu trình, có thể rơi vào vòng lặp vô hạn. Một số cách xử lý khả thi:

* Dừng sau một số lần thử cố định và trả về “không có lời giải” (*max\_tries*).
* Sử dụng IDS (Iterative Deepening Search) để giải quyết vấn đề này.

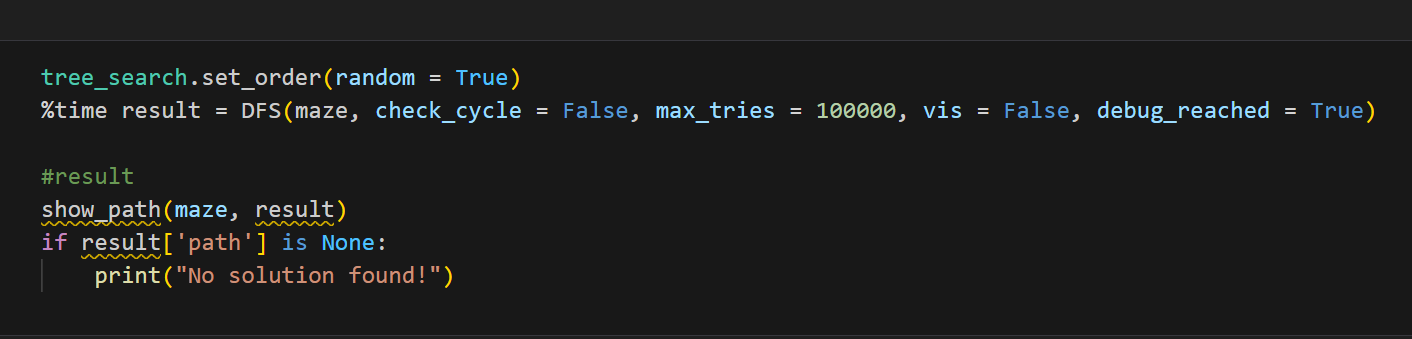


* Nếu mê cung không có chướng ngại (gần như “trống rỗng”), DFS có thể lặp vô hạn vì kiểm tra chu trình (cycle checking) không đủ chặt. Do đó, đặt giới hạn max\_tries: số bước tối đa trước khi dừng.
* Tiến hành chạy và đo thời gian với giới hạn số lần thử.
* Vẽ lại mê cung và đánh dấu đường đi. Kiểm tra nếu không tìm thấy đường đi đến đích (không có ‘path’), in ra thông báo.

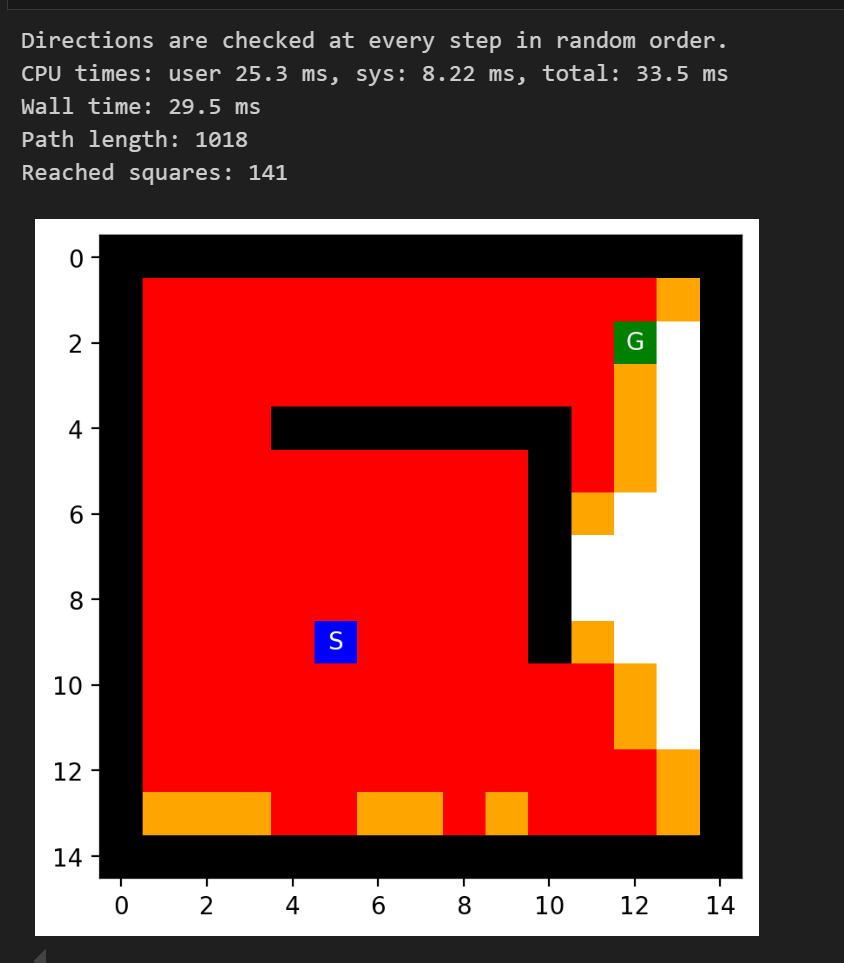


* Kết quả hiển thị thuật toán DFS no reached mất khoảng gần 1.5 – 2 mili giây để duyệt qua 117 ô và tìm được đường dài 74 bước.

\*\* Chúng ta có thể thực hiện một bước đi ngẫu nhiên và không kiểm tra các chu trình.  
Cách này đảm bảo cuối cùng sẽ đến được bất kỳ ô nào, kể cả ô đích, nhưng sẽ tạo ra một đường đi rất dài. Đường đi này có thể được đơn giản hóa, và ý tưởng đó dẫn đến thuật toán **Trémaux**.



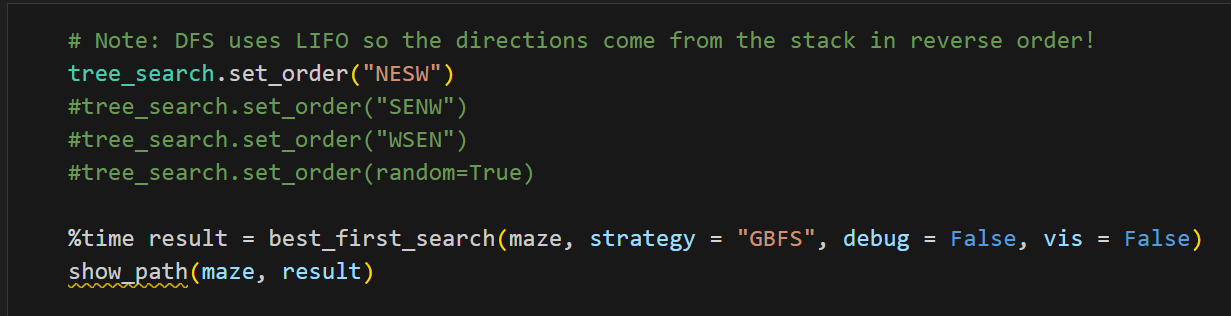
* Dùng DFS với cách đặt thứ tự duyệt các hướng đi (Đông, Tây, Nam, Bắc) theo cách ngẫu nhiên cho mỗi bước.
* Tiến hành đo thời gian thực thi của hàm, không kiểm tra vòng lặp **check\_cycle = False** với cách này DFS sẽ không lưu tập “đã đến” nên có thể đi lại nhiều lần trên cùng ô đó, có thể lặp vô hạn.
* Để tránh vòng lặp vô hạn, đặt **max\_tries = 100000** để thuật toán dừng sau 100000 bước nếu chưa tìm được đích đến.
* **debug\_reached = True** cho phép ghi nhận/thống kê các ô đã ghé qua.
* Cuối cùng kiểm tra xem **result** có đường đi “path” hay không, nếu không in ra “No solution found!”.

****

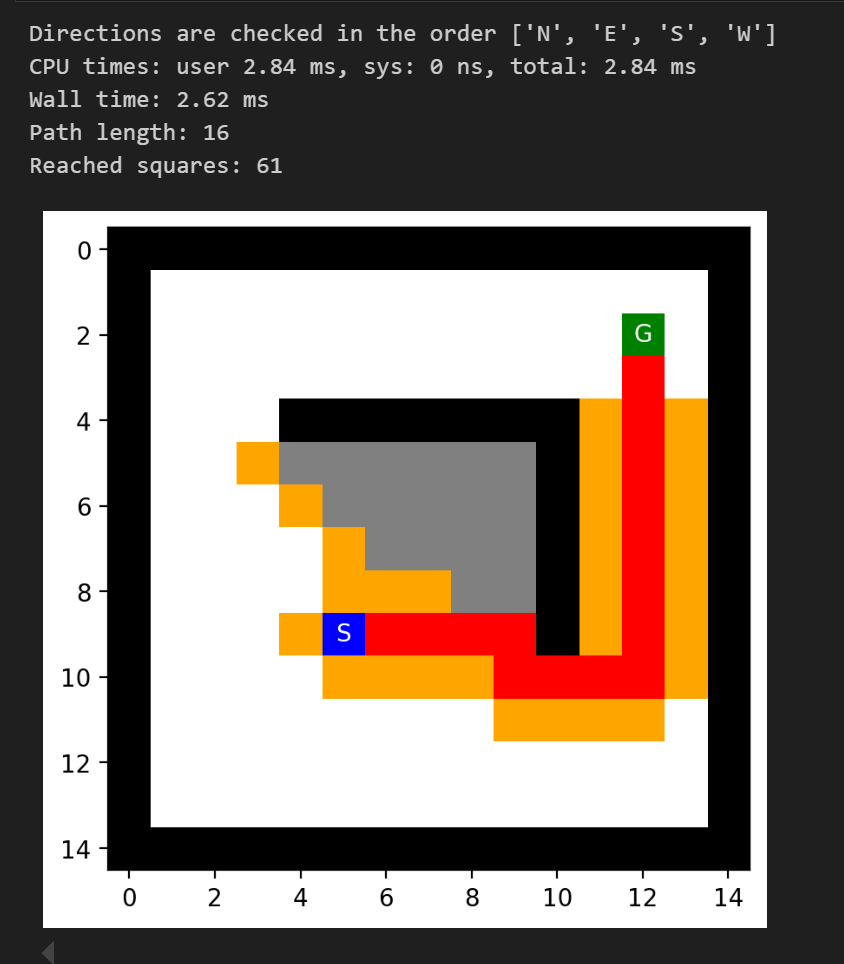
* Kết quả trả về thuật toán mất khoảng 30 mili giây để duyệt qua 141 ô khác nhau và tìm ra đường đi dài là 1018 bước.

# **d) GBFS -  Greedy Best-First Search**

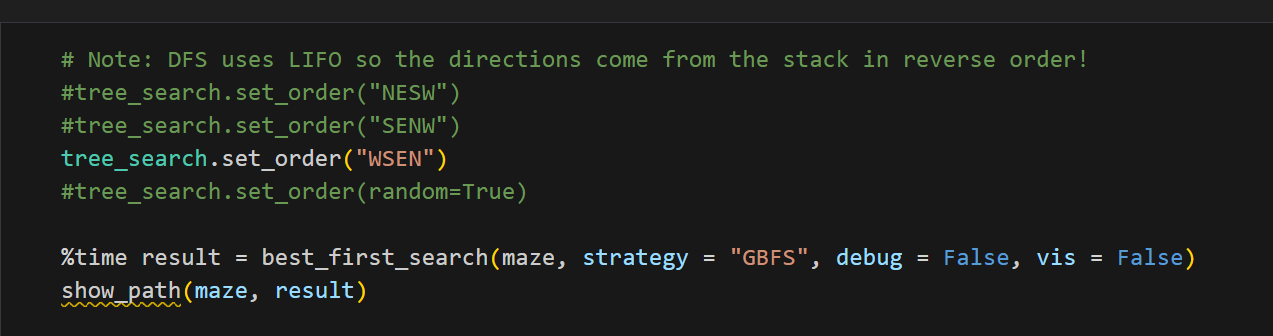
* **Cách 1: duyệt theo “NESW”**

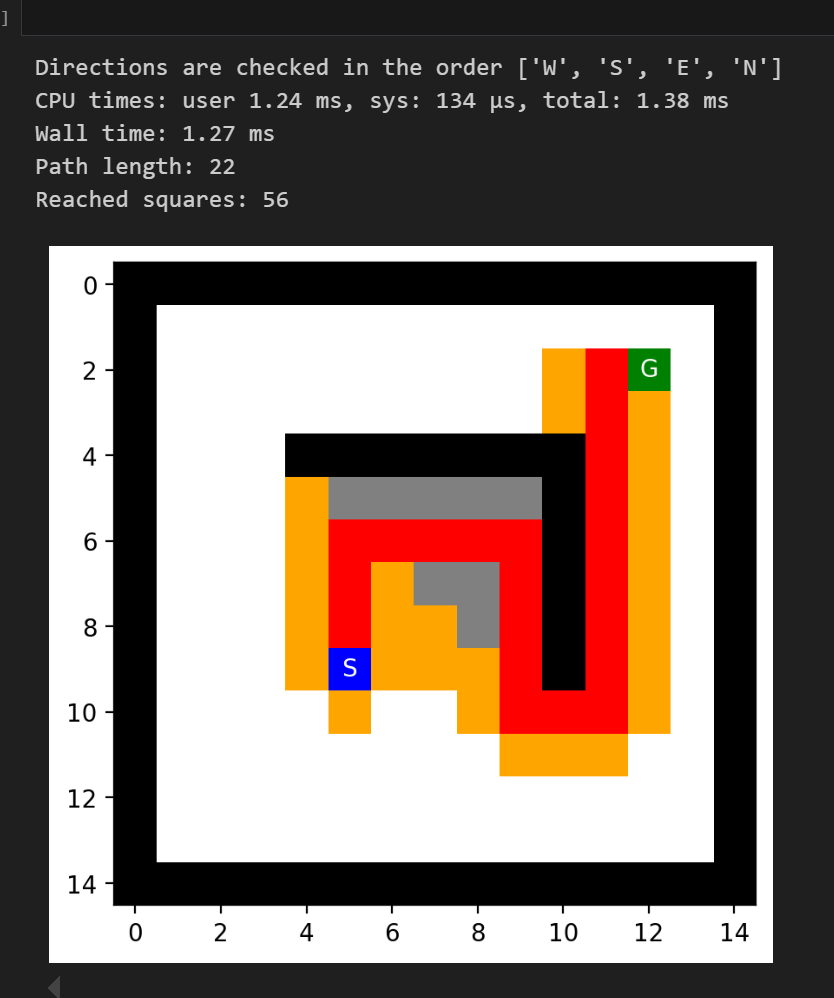
****

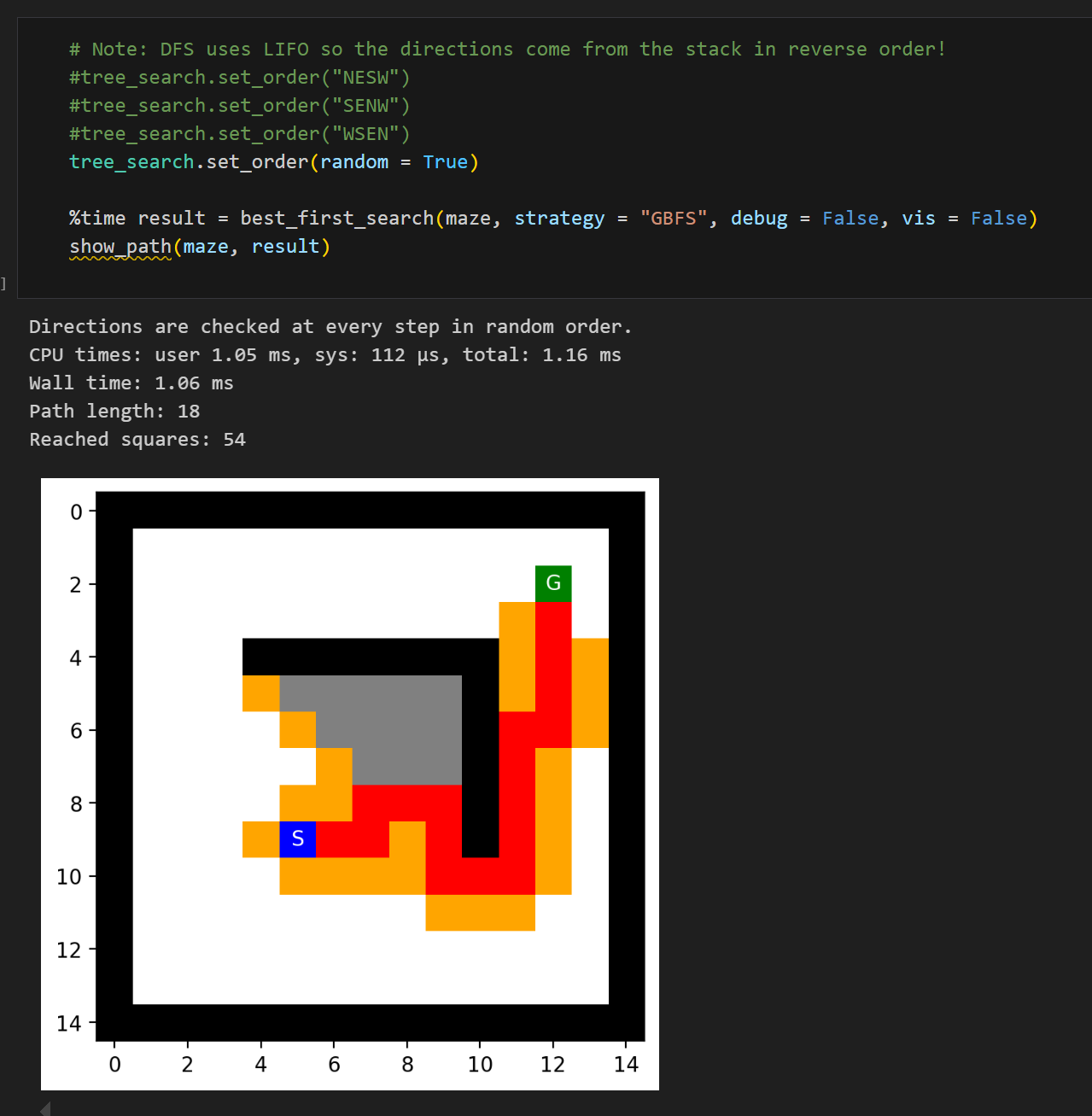
* Thiết lập thứ tự tìm kiếm theo hướng: Bắc 🡪 Đông 🡪 Nam 🡪 Tây
* Tiến hành đo thời gian thực thi, dùng “GBFS” làm hàm tìm kiếm.

****

* Kết quả cho thấy với cách duyệt này GBFS mất khoảng 2.5 đến gần 3 mili giây để tìm được đường dài 16 bước sau khi duyệt qua 61 ô.
* **Cách 2: duyệt theo “WSEN”**

****

* Như cách làm ở trên nhưng thay hướng duyệt bằng Tây 🡪 Nam 🡪 Đông 🡪 Bắc.****
* Kết quả hiển thị cách này mất khoảng 1.2 đến 1.4 mili giây để duyệt qua 56 ô và tìm được đường đi dài 22 bước.
* **Cách 3: duyệt ngẫu nhiên**

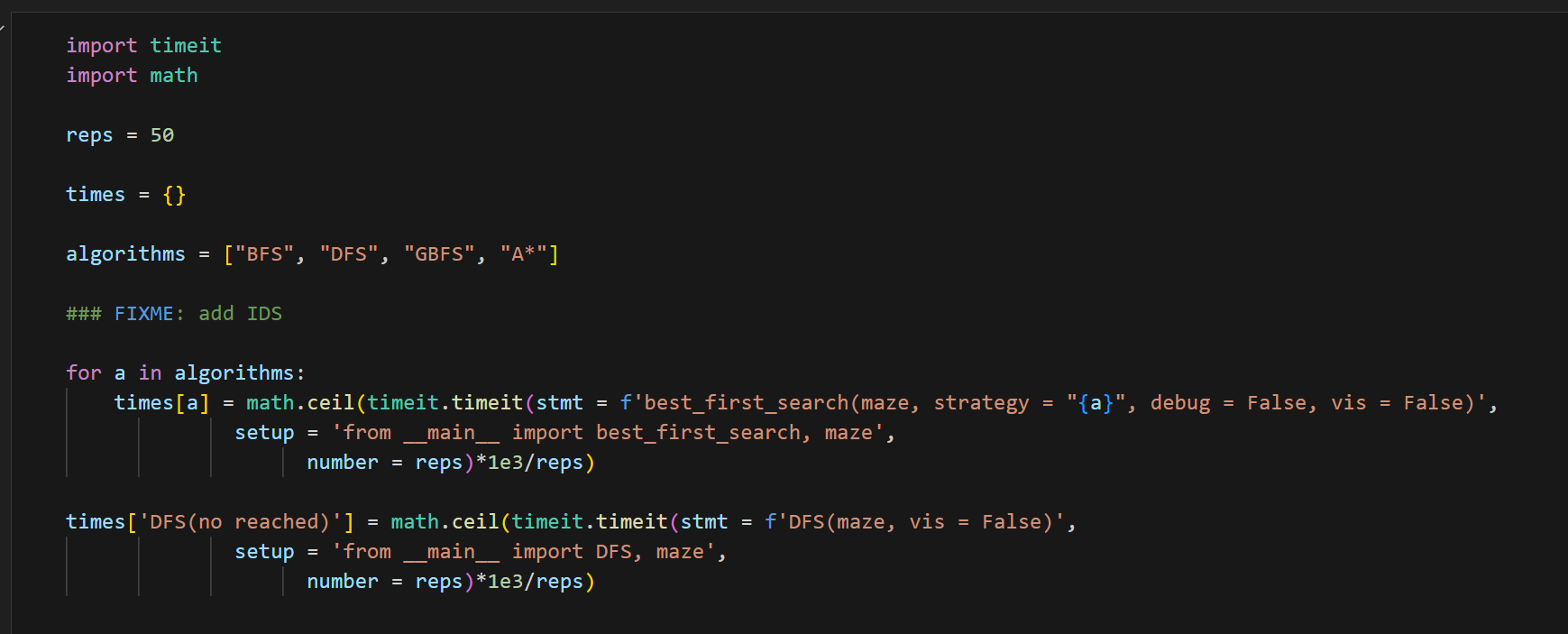
****

* Với cách duyệt theo hướng ngẫu nhiên, kết quả cho thấy để duyệt qua 54 ô và tìm được đường dài 18 bước đã mất khoảng 1.2 mili giây. (Kết quả có thể thay đổi sau những lần duyệt theo hướng ngẫu nhiên).

# **e) A\* Search**

* Vì A\* Search là thuật toán tìm được đi ít chi phí nhất và ưu tiên nút gần đích nhất nên kết quả tương đối giống nhau về đường đi tìm được và thời gian mất khoảng từ 1 đến 2 mili giây.

# **f) So sánh thời gian giữa các thuật toán (BFS, DFS, GBFS, A\*)**

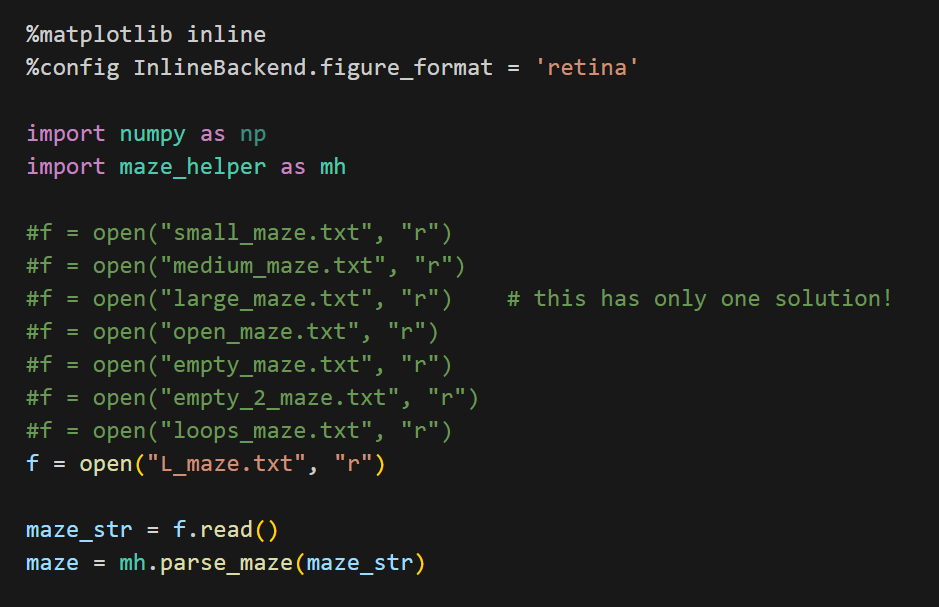


* Sử dụng thư viện timeit để đo thời gian thực thi đoạn code nhỏ, tránh sai số do môi trường.
* Dùng **reps = 50** để lấy thời gian trung bình mỗi thuật toán sau 50 lần lặp.
* Tạo một dictionary **times**  để lưu kết quả là thời gian của các thuật toán cần đo.
* Kết quả đo được trả về dưới 2 dạng bảng dữ liệu và biểu đồ cột với trục X chứa tên các thuật toán, trục Y thể hiện giá trị thời gian trung bình tương ứng theo mili giây (ms).

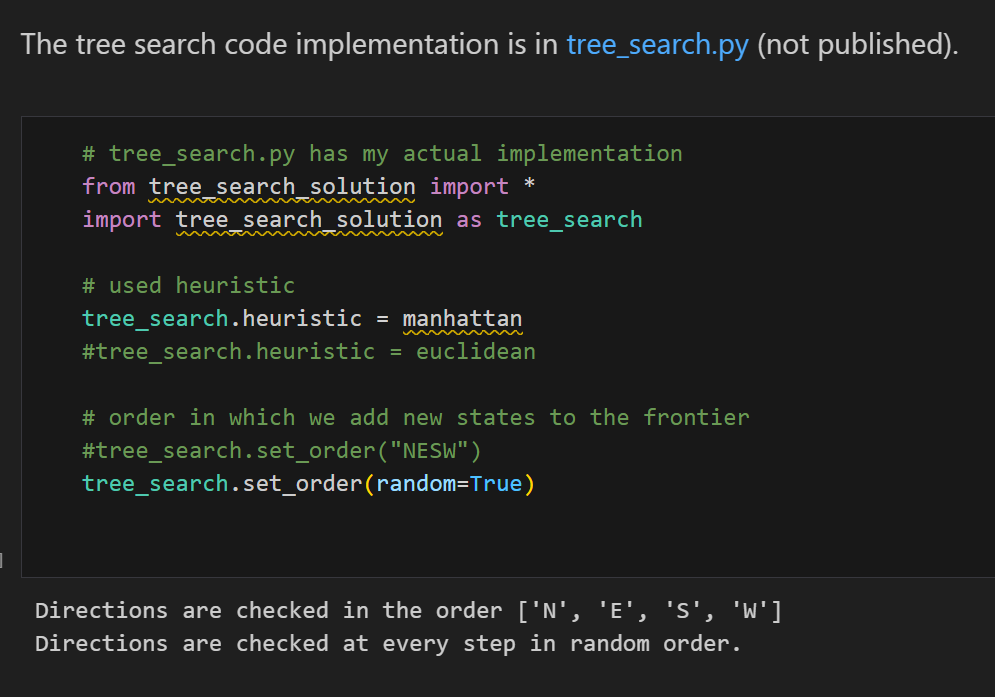
****

* **Nhận xét:** Sự khác biệt là rất nhỏ vì mê cung và và bài toán thử khá nhỏ. Biểu đồ hiển thị thuật toán DFS nhanh hơn các thuật toán khác trong đợt thử nghiệm này, nhưng điều đó không có nghĩa DFS sẽ luôn tối ưu. Đối với những mê cung lớn và phức tạp, A\* thường hiệu quả hơn nhờ vào heuristic.

# **2. Mazes: Compare BFS with A-Star Search using the L Maze. (So sánh BFS với A\* Search bằng L Maze)**

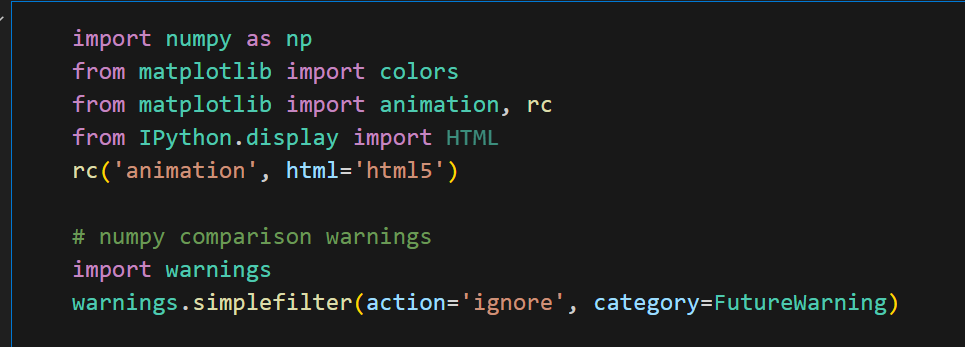


* Sử dụng **%matplotlib inline** hiển trị đồ thị trong notebook, **%config InlineBackend.figure\_format = ‘retina’** tăng độ phân giải hình ảnh và làm đồ thị sắc nét hơn.
* Import các thư viện cần thiết như numpy, file **maze\_helper** chứa các hàm đọc/ hiển thị mê cung được cung cấp sẵn.
* Tiến hành mở và đọc file chứa mê cung **“L\_maze.txt”**.
* Chuyển đổi nội dung file thành chuỗi ký tự và biến chuỗi này thành cấu trúc dữ liệu mê cung (thường là mảng 2D, chứa tường, đường đi và đích đến).

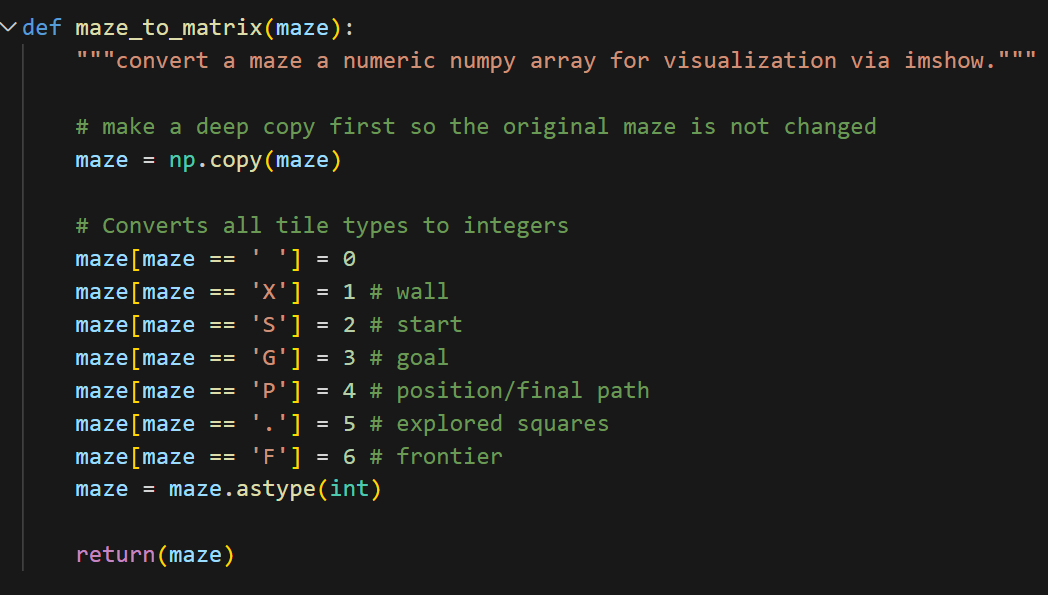


* Import module giải thuật tìm kiếm, sử dụng hàm heuristic và chọn manhattan để tiến hành tìm kiếm theo hướng ngẫu nhiên.

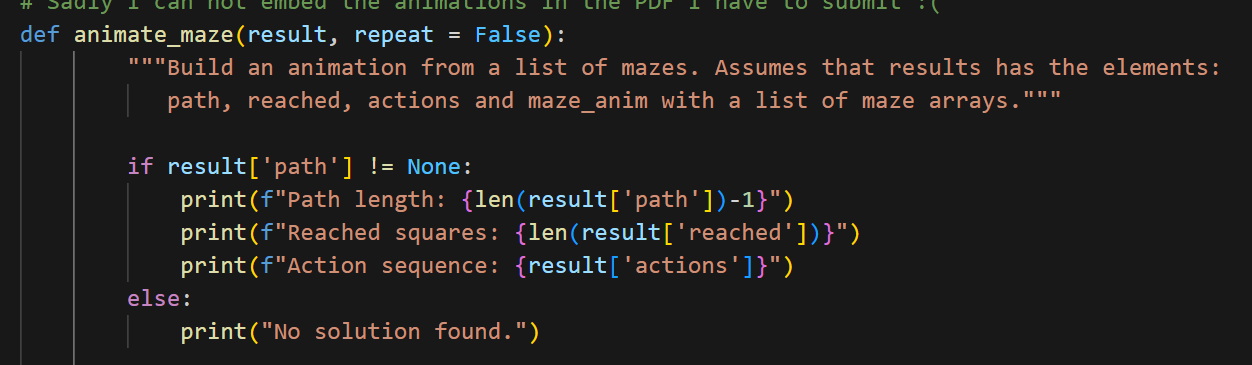
Animation Code



* Import các thư viện trong đó **matplotlib.colors** dùng định nghĩa bảng màu cho hiển thị mê cung. **matplotlib.animation + rc (‘animation’, html=’html5’)** là cấu hình để hiển thị animation trực tiếp trong notebook dưới dạng HTML5.



* Hàm **maze\_to\_matrix** sẽ nhận một mảng ký tự trong mê cung. Sau đó, chuyển mỗi loại ô sang mã số nguyên để **“imshow”** vẽ bằng màu.
* Hàm **animate\_maze** dùng để:



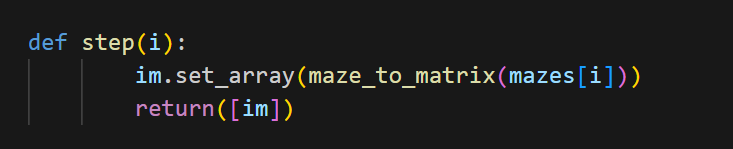
+ In thông tin đường đi: chiều dài đường đi, số ô đã duyệt, chuỗi hành động.



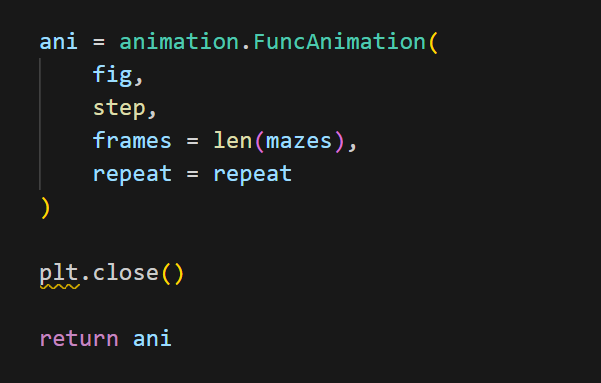
+ Chuẩn bị dữ liệu vẽ : tạo bảng màu, xác định toạ độ S và G để đặt chữ, chuyển mỗi trạng thái mê cung sang ma trận số.



+ Thiết lập figure ban đầu: tạo đối tượng hình vẽ, hiển thị trạng thái đầu tiên. Vẽ vị trị S và G tương ứng.



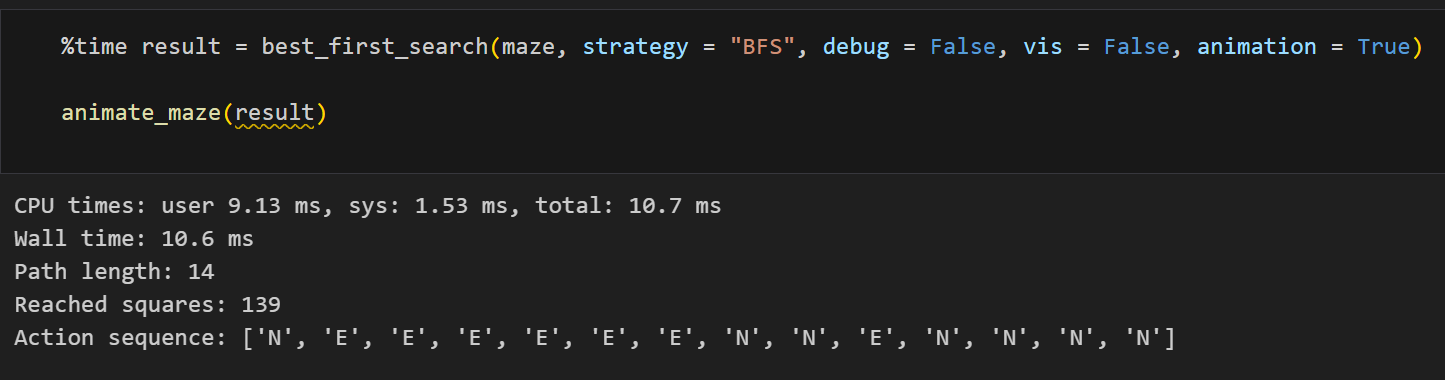
+ Cập nhật từng frame: Mỗi lần gọi sẽ thay thế dữ liệu ảnh để hiện frame thứ i.



+ Tạo animation: **FuncAnimation** tạo hoạt hình từ danh sách maze. **plt.close()** để không hiển thị tĩnh ngay lập tức.

**BFS**

Tìm kiếm theo chiều rộng là một thuật toán tối ưu. BFS là một thuật toán tìm kiếm không có thông tin và không biết đích đến ở đâu. Nó mở rộng tìm kiếm theo các vòng tròn đồng tâm xung quanh điểm bắt đầu cho đến khi đạt đến đích.

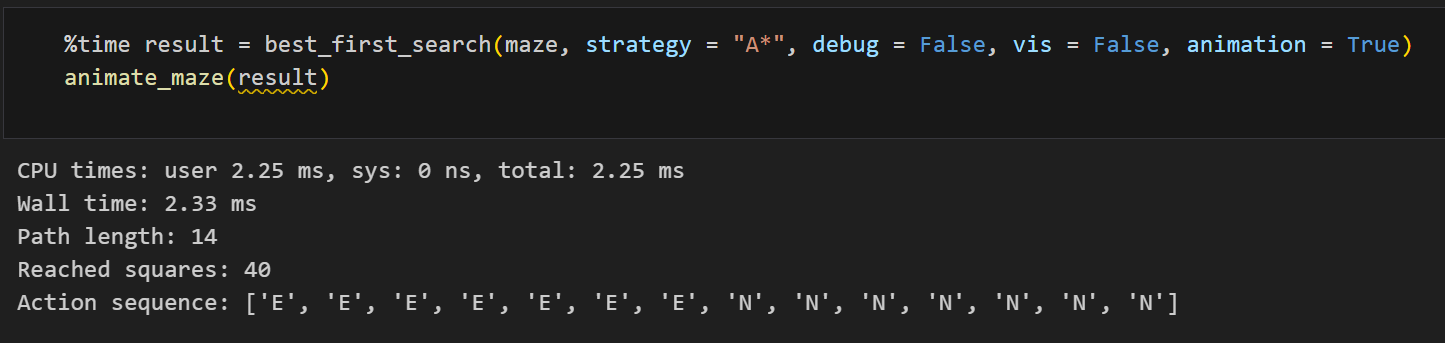




* Sử dụng BFS để tìm đường đi trong mê cung, tiến hành đo thời gian thực thi, tạo hoạt hình mô phỏng từng bước và báo cáo chi tiết đường đi.
* Kết quả cho thấy thuật toán mất khoảng 10.6 mili giây để duyệt qua 139 ô và tìm được đường đi dài 14 bước (kết quả có thể khác nhau sau những lần thực nghiệm do duyệt theo hướng đi ngẫu nhiên).

**A\* Search**

Tìm kiếm A\* là một thuật toán tìm kiếm có thông tin. Nó thu thập thông tin về vị trí của mục tiêu bằng cách sử dụng hàm heuristic. Giá trị của hàm heuristic có xu hướng giảm dần khi chúng ta đến gần mục tiêu. Thuật toán được coi là tối ưu nếu A\* là một heuristic chấp nhận được.





* Sử dụng A\* để tìm đường đi trong mê cung, tiến hành đo thời gian thực thi, tạo hoạt hình mô phỏng từng bước và báo cáo chi tiết đường đi.
* Kết quả cho thấy thuật toán mất khoảng 2.33 mili giây để duyệt qua 40 ô và tìm được đường đi dài 14 bước (kết quả có thể khác nhau sau những lần thực nghiệm do duyệt theo hướng đi ngẫu nhiên).

\*\* So sánh:

* BFS và A\* đều là những thuật toán tối ưu vì chúng luôn tìm được đường đi ngắn nhất từ điểm xuất phát đến đích.
* BFS đã duyệt qua 139 trạng thái và phải lưu cây tìm kiếm có kích thước tương ứng thì A\* chỉ cần duyệt qua 40 trạng thái 🡪 Tiết kiệm bộ nhớ hơn nhiều.
* Về thời gian tìm kiếm A\* nhanh hơn BFS vì khám phá ít trạng thái hơn BFS
* Hiệu quả của A\* phụ thuộc vào chất lượng heuristic. Với bài toán mê cung dạng lưới, Manhattan distance (tổng khoảng cách theo trục dọc + ngang) là heuristic rất tốt.
* Nếu không gian tìm kiếm quá lớn, ta có thể cân nhắc Weighted A\* hoặc IDS để giảm áp lực bộ nhớ.

# **3. Maze\_example-Animation\_full**

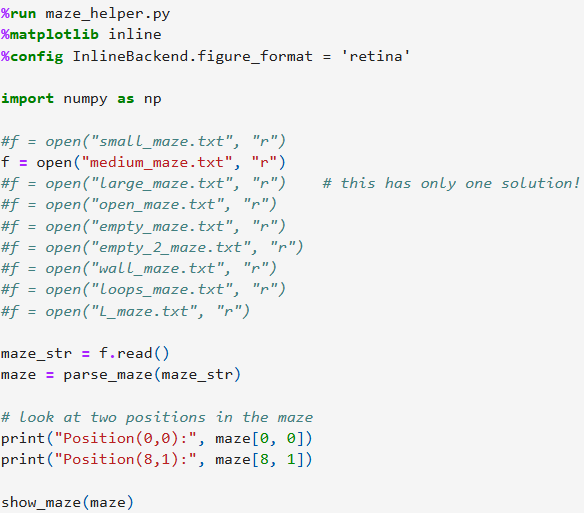
- Xử lý khi có ràng buộc bằng nhau (Breaking ties):

+ Thứ tự duyệt các nút con (theo các hướng đi có sẵn) ảnh hưởng rất nhiều đến DFS và cách xử lý các ô trống. Tôi chọn duyệt các hướng theo thứ tự ngẫu nhiên, điều này khiến thuật toán mang tính ngẫu nhiên (stochastic)!

+ Đối với các trường hợp bằng nhau về giá trị h(n) và f(n), cần phải có cách xử lý thống nhất. Tôi sử dụng nút được thêm gần nhất (most recently added node), với mục đích cố gắng tiếp tục di chuyển theo cùng một hướng.

- Chu trình (Cycles): Chu trình được tránh bằng cách sử dụng cấu trúc dữ liệu reached. Với DFS, ta có thể dùng reached để kiểm tra, hoặc nếu không dùng reached thì có thể kiểm tra chu trình chỉ bằng thông tin về cha (parents).

- Tìm kiếm có giới hạn độ sâu (Depth-limited search): Để hoạt động tốt trong các không gian mở, cần áp dụng thêm một số ý tưởng từ tìm kiếm tốt nhất trước (best-first search).



- %run maze\_helper.py: chạy file maze\_helper để nạp các hàm hỗ trợ (như parse\_maze, show-maze,…)

- f= open(…): mở file mê cung cần phân tích

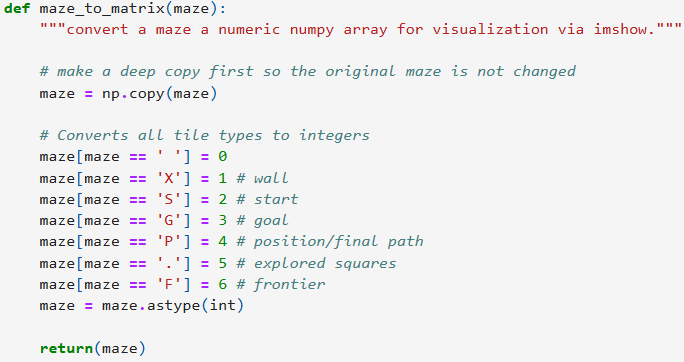
- f.read(): đọc toàn bộ nội dung file mê cung vào chuỗi maze\_str

- parse\_maze: chuyển chuỗi thành mảng 2D

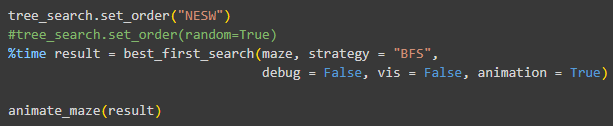
- hàm tạo animation từ quá trình tìm đường đi trong mê cung



- hàm chuyển mê cung dạng kí tự sang mảng số nguyên. Sau đó ép kiểu về int

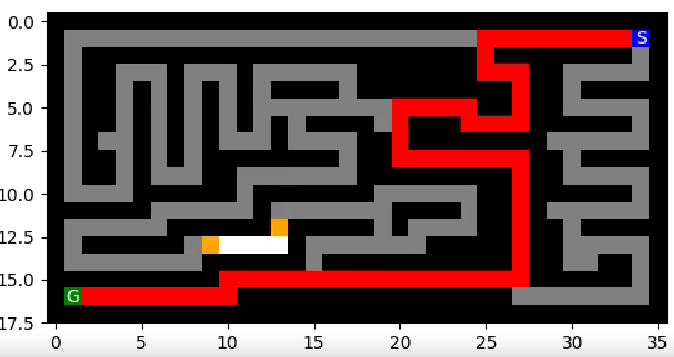


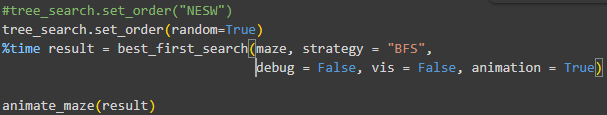
\* BFS:



- duyệt nút theo hướng cố định: Bắc->Đông->Nam->Tây

- đường đi tìm được: ổn định, luôn cùng một kết quả





- duyệt nút ngẫu nhiên

- đường đi tìm được: có thể thay đổi mỗi lần chạy(nếu có nhiều đường đi ngắn nhất cùng tồn tại)



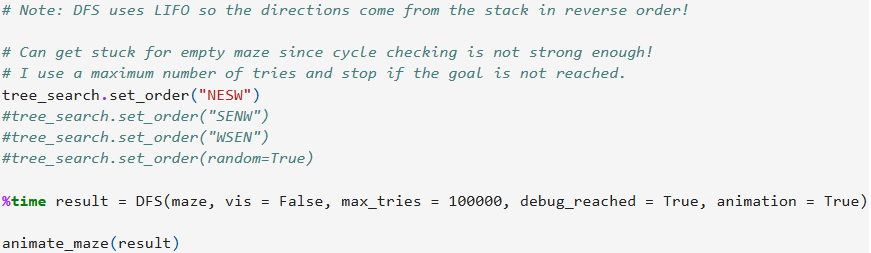
\* DFS không sử dụng cấu trức dữ liệu

- Điều này có độ phức tạp về không gian là O(bm) thay vì O(b^m) khi ta tái sử dụng thuật toán tìm kiếm theo cây (tree search) từ BFS!

- Tuy nhiên, vì ta không thể kiểm tra được tất cả các chu trình, nên vẫn có khả năng gặp vòng lặp vô hạn. Dưới đây là một số cách giải quyết khả thi:

+Dừng sau một số lần thử cố định và trả về “không có lời giải” (max\_tries).

+ IDS (Iterative Deepening Search) giải quyết được vấn đề này.



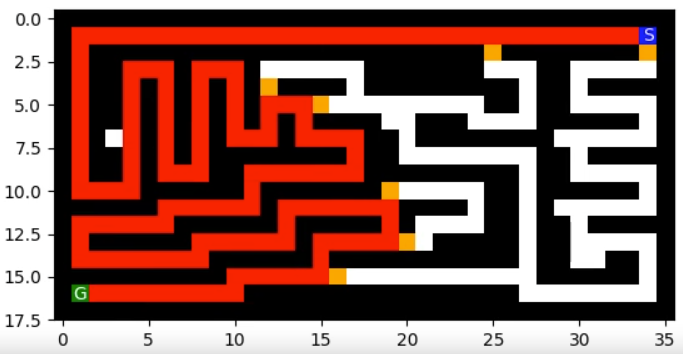
- duyệt theo thứ tự: Bắc -> Đông -> Nam -> Tây

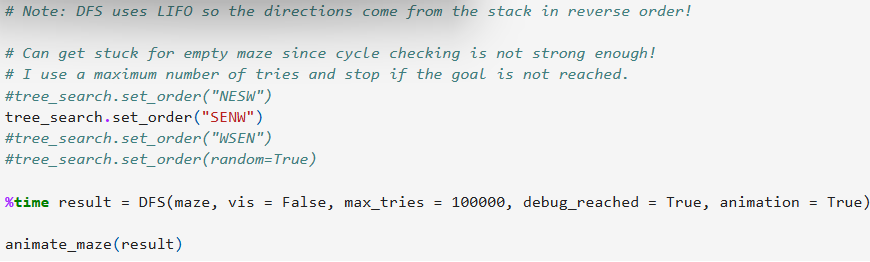
- DFS là LIFO nên nút Tây sẽ được thăm đầu tiên sau đó đến Nam, Đông cuối cùng là Bắc

- đường đi luôn ưu tiên theo mẫu nhất định

- ưu điểm: kết quả ổn định

- nhược điểm: có thê đi nhiều vòng lặp hoặc bị kẹt trong vòng lặp





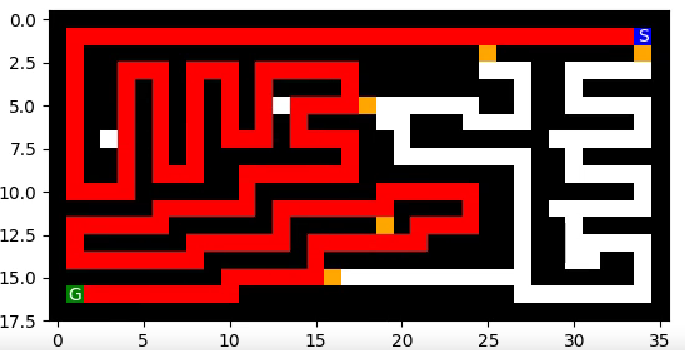
- duyệt theo thứ tự: Nam -> Đông -> Bắc -> Tây

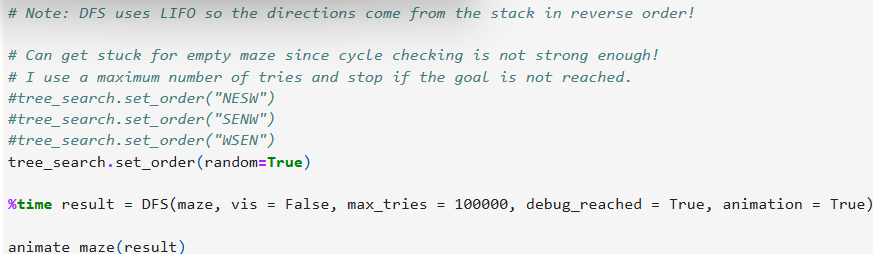
- Tương tự như ví dụ trên, nút Tây được thăm đầu tiên, sau đó Đông, Bắc, Nam

- đường đi thay đổi so với NESW

- ưu điểm: tìm đường đi khác, có thể tìm ra lối gần hơn trong 1 số trường hợp

- nhược điểm: có thể đi nhiều vòng lặp hoặc mắc kẹt trong vòng lặp



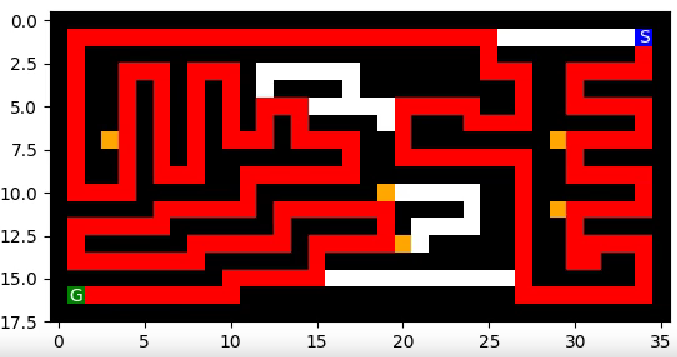


- duyệt theo thứ tự ngẫu nhiên ở mỗi lần mở rộng

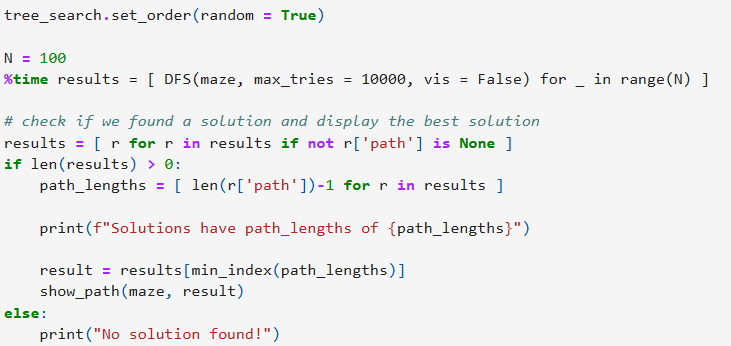
- mỗi lần chạy có thể ra đường đi khác nhau

- ưu điểm: có thể tìm được lời giải nhanh hơn

- nhược điểm: không ổn định, có thể tốn nhiều thời gian hơn



\* Chạy DFS ngẫu nhiên nhiều lần và chọn lời giải tốt nhất.



- đặt thứ tự duyệt con là ngẫu nhiên

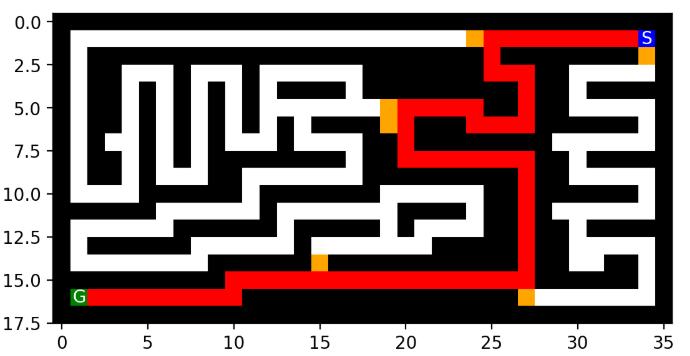
- N=100: thực hiện 100 lần chạy DFS ngẫu nhiên trên cùng một mê cung

- max\_tries=10000: giới hạn số bước thử để tránh vòng lặp vô hạn

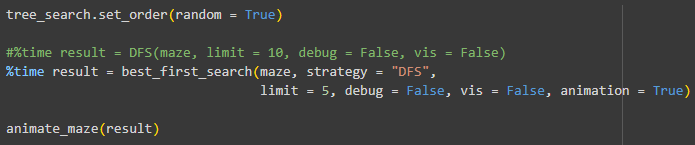
- vis = false: không bật chế độ hiển thị chi tiết trong khi tìm kiếm

- path\_lenght: tính độ dài đường đi của từng kết quả (len(r[‘path’])-1)

- result = results[min\_index(path\_lengths)]: tìm lời giải có đường đi ngắn nhất bằng cách lấy index của giá trị nhỏ nhất trong path\_lenght

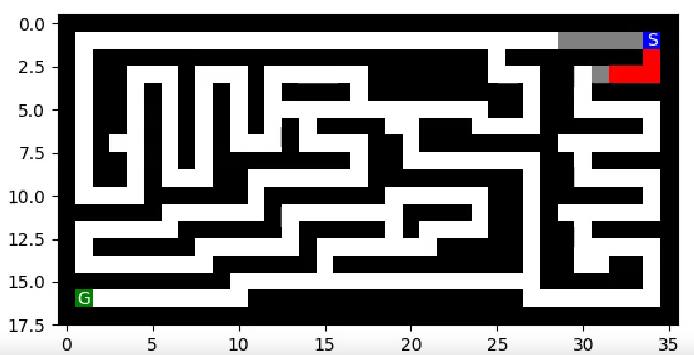


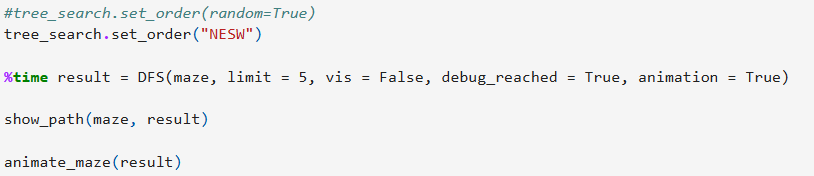
\* DFS giới hạn độ sâu (Depth-limited DFS): DFS không lưu lại cây tìm kiếm đã được duyệt để trực quan hóa, vì vậy ta sử dụng cài đặt tree search để hiển thị.



- thứ tự duyêt ngẫu nhiên cho các hướng

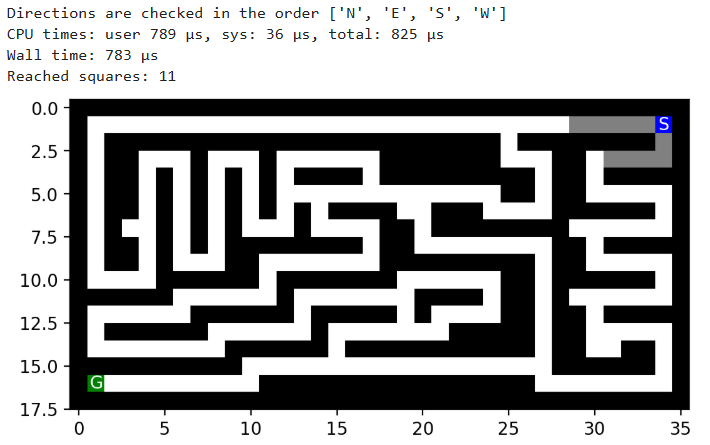
- best\_first\_search với strategy = “DFS” và limit = 5: là chạy DFS nhưng dưới khung tree search và giới hạn độ sâu =5 để có thể dễ dàng trực quan hóa hơn

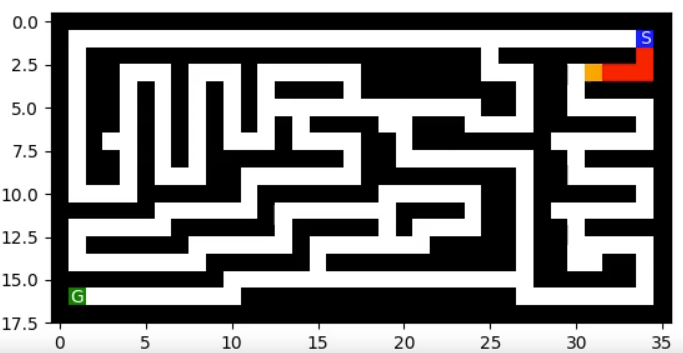




- ở đây khoogn dùng ngẫu nhiên nữa mà cố định qua các hướng Bắc->Đông->Nam->Tây

- vì DFS dùng LIFO nên thứ tự duyệt sẽ là Tây->Nam->Đông->Bắc

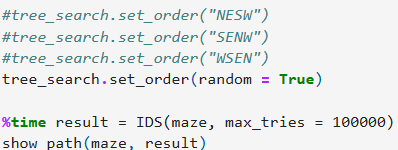




\* IDS(Iterative Deepening Search: tìm kiếm lặp sâu)

- IDS khi kết hợp với DFS không lưu lại các ô đã duyệt, vì vậy các vùng màu xám (đã duyệt) sẽ không được hiển thị!

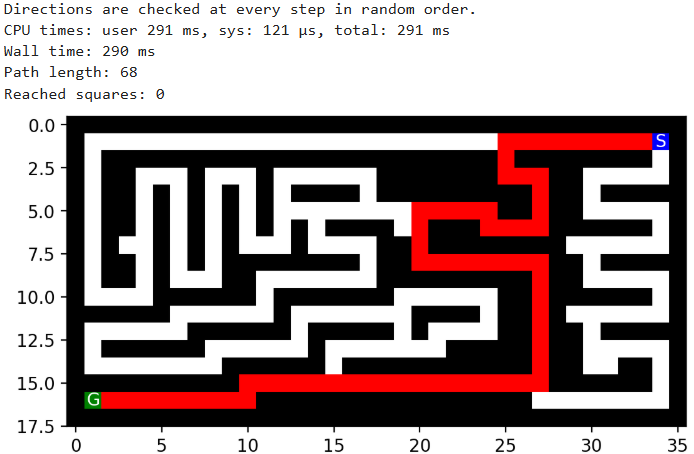
- IDS phụ thuộc vào cơ chế kiểm tra chu trình (cycle checking) của DFS, do đó nó cũng bị ảnh hưởng bởi các vấn đề tương tự.

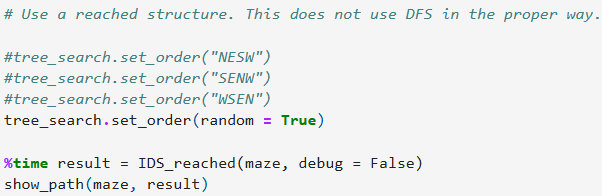


- chọn thứ tự duyệt các hướng ngẫu nhiên khi mở rộng nút

- IDS sẽ chạy nhiều lần DFS giới hạn dộ sâu với limit tăng dần từ 1,2,3… cho tới khi tìm thấy lời giải hoặc đạt tới max\_tries

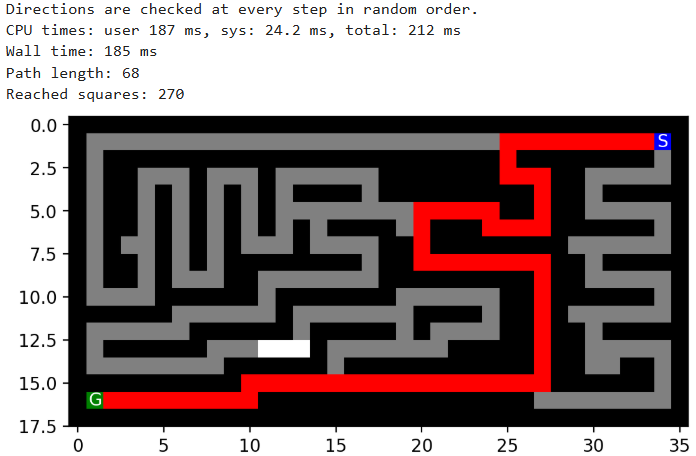
- max\_tries =10000: giới hạn số bước duyệt trong vòng lặp



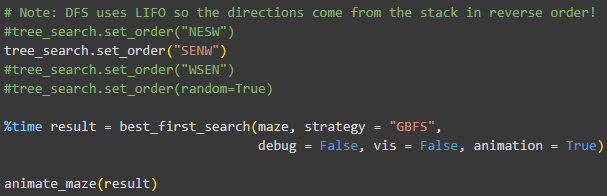


- ở đây sử dụng cấu trúc reached.

- duyệt các hướng ngẫu nhiên

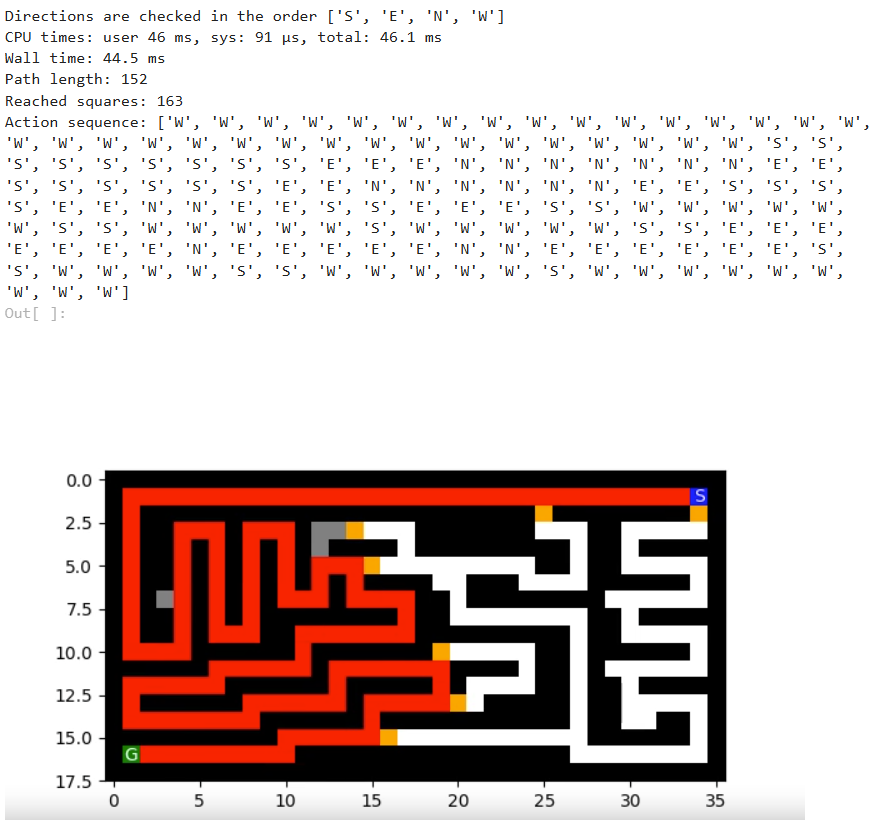


\*GBFS(Greedy Best-First Search): tìm kiếm tốt nhất trước tham lam

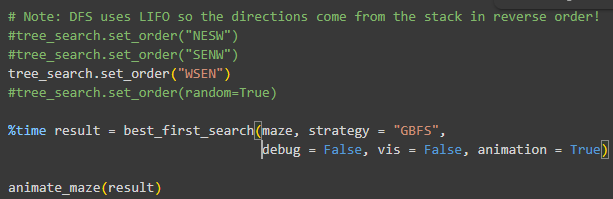


- duyệt các nút con theo thứ tự Nam->Đông->Bắc->Tây

- GBFS sẽ chọn nút có heuristic(h(n)) nhỏ nhất. Nhưng nếu có nhiều h(n) bằng nhau sẽ thứ tự duyệt sẽ theo lệnh set\_order

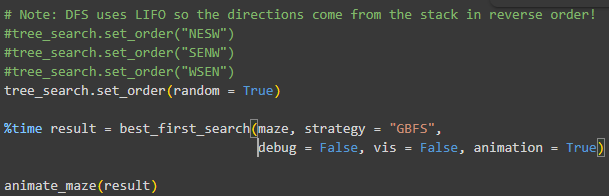


- Tương tự như trên, chỉ khác hướng duyệt là Tây-> Nam-> Đông-> Bắc





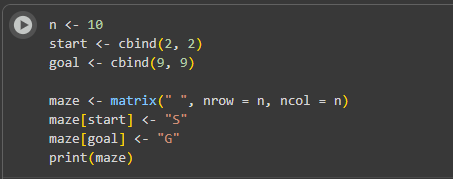
- Ví dụ này thì duyệt các nút theo thứ tự ngẫu nhiên





# **4. Explore heuristic.**

a) Khởi tạo mê cung

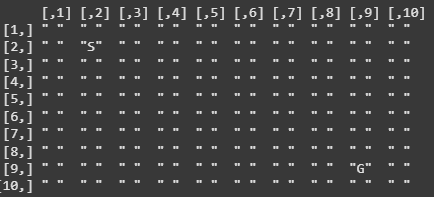


- Ban đầu tạo 1 mê cung có kích thươc là 10x10

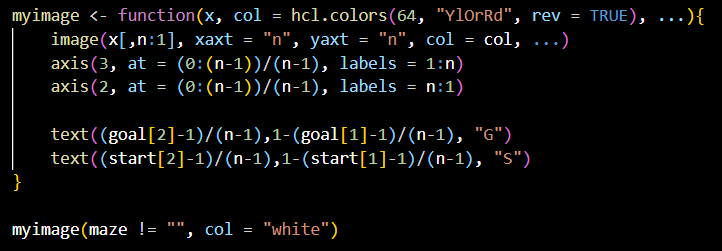
- Kế tiếp đặt điểm bắt đầu ở ô (2,2) và đích ở ô (9,9) (start <- cbind(2,2) và goal <- cbind(9,9))

-  : Tạo ra 1 ma trận rỗng chứa khoảng trắng “ ” để biểu diễn ô trống

- maze[start] <- "S" và maze[goal] <- "G": đánh dấu vị trí S và G trong mê cung.  
Mê cung in ra:



b) Hàm vẽ mê cung



- x[, n:1]: đảo ngược thứ tự dòng(từ dưới lên trên) để hiển thị đúng hướng như ta đọc mê cung

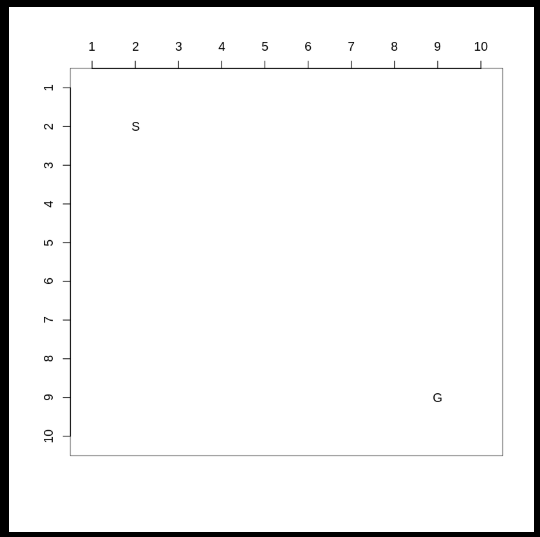
- xaxt = ”n”, yaxt = ”n”: tắt trục x và y mặc định

- image: sẽ biến ma trận thành một hình ảnh dạng lưới màu

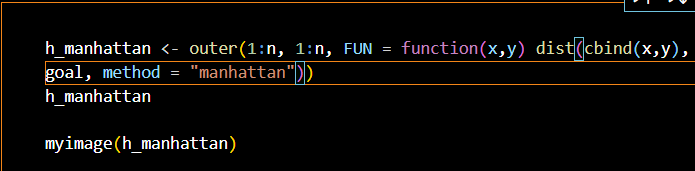
- axis(): thêm thứ tự trên trục của biểu đồ

-text(x,y,”S”) và text(x,y,”G”): đánh dấu chữ “G” tại ô Goal và “S” tại ô Start

Biểu đồ in ra:



c) Heuristic Manhattan



- outer(a, b, FUN): tạo ra một ma trận có kích thước length(a) x length(b). Ở đây a =1:n, b=1:n, tức là xét tất cả các cặp tòa độ (x,y) trong lưới vuông kích thước n x n

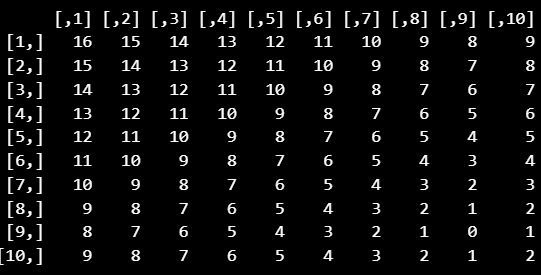
- FUN = function(x, y) dist(cbind(x, y), goal, method = "manhattan")\

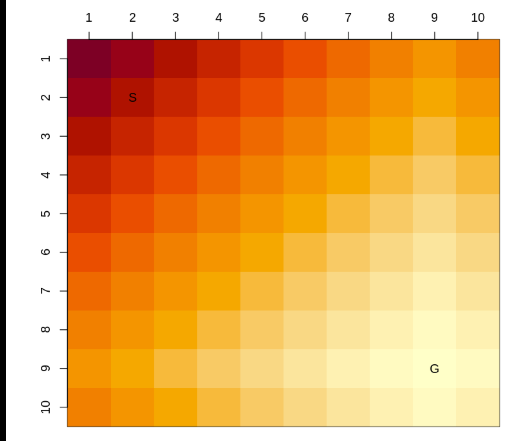
+ cbind(x,y) tạo thành một điểm có tọa độ (x,y)

+ goal là 1 điểm đích cố định

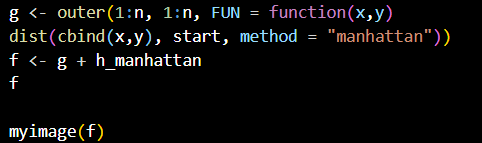
+ dict(…, method = “manhattan”): tính khoảng cavchs Manhattan(L1d- norm) giữa điểm (x,y) và goal

. Công thức khoảng cách Manhattan = |x - gx|+|y - gy|





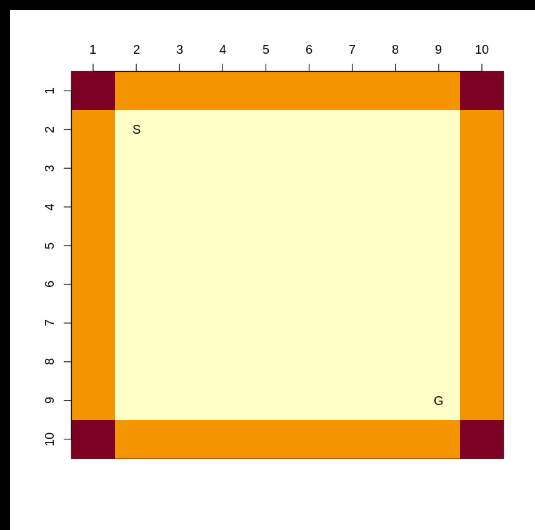
d) A\* với Manhatta heuistic

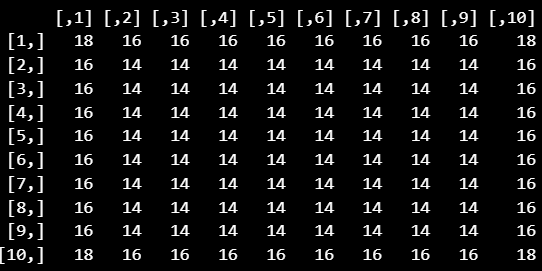


- Tương tự như hàm h\_manhattan ở trước, nhưng giờ ta tính khoảng cách Manhattan từ điểm (x,y) đến điểm xuất phát start

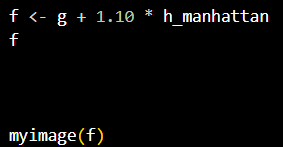
- g là ma trận nxn chứa khoảng cách Manhattan từ mỗi ô trong lưới đến start

- f là tổng khoảng cách từ (x,y) đến cả 2 điểm Start và Goal

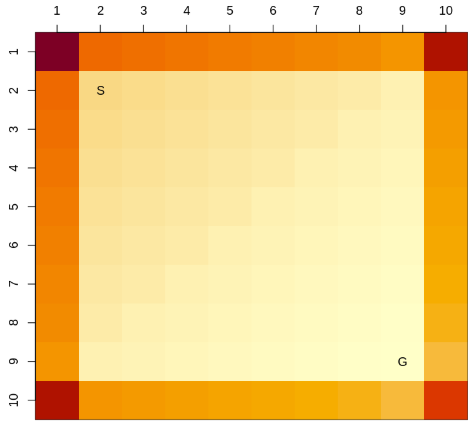
Biểu đồ được xuất ra:

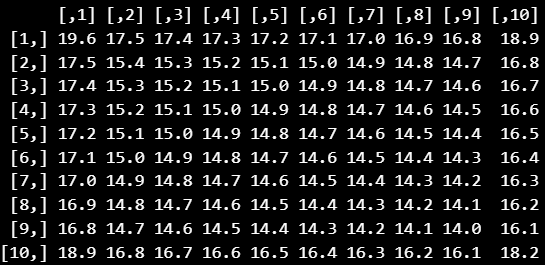


e) A\* có trọng số



- nhân thêm 1.10 -> làm cho heuristic “mạnh hơn”



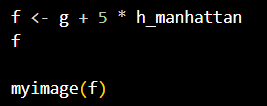


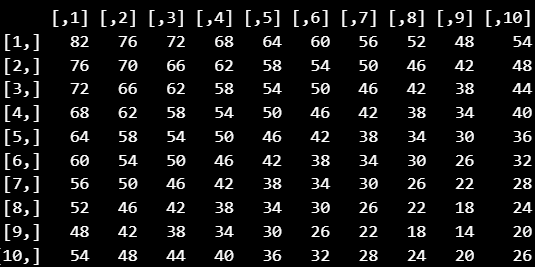
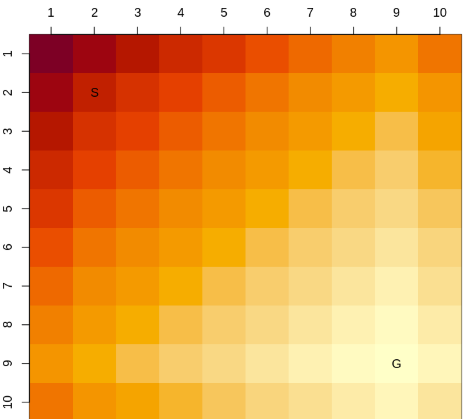
- cách này thường dùng:

+ tìm đường nhanh hơn vid heuristic ưu tiên mạnh mẽ hướng về goal

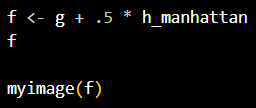
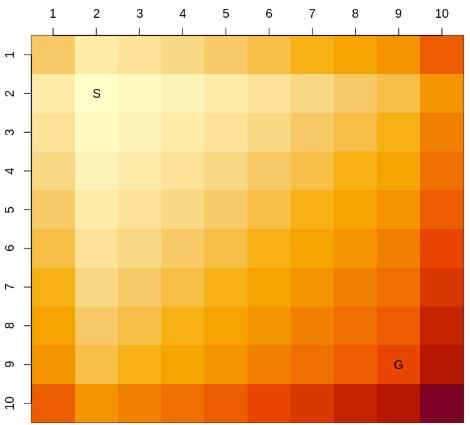
+ nhưng có thể làm kết quả không còn được tối ưu tyệt đối, ví heuristic không còn đảm bảo tính chấp nhận được

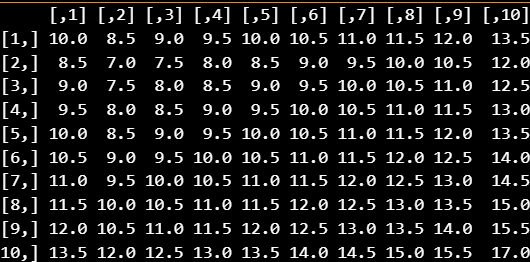
- với W = 5:





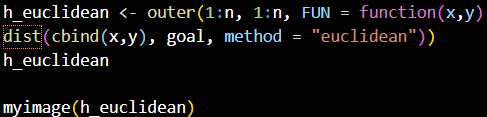
- với W = 0.5:





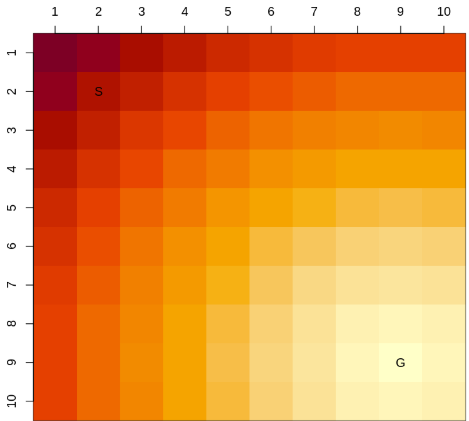
-> Sử dụng W<1 sẽ xấp xỉ việc tìm kiếm theo mã đồng đều (Uniform-cost Search/BFS) và sẽ ưu tiên khám phá khu vực xung quanh S trước. Vì một phần (fraction) của một heuristic khả chấp (admissible) thì vẫn còn khả chấp, nên ta vẫn được đảm bảo tính tối ưu (giống như trong BFS).

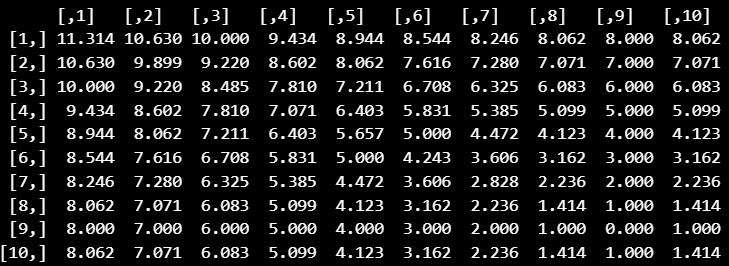
f) GBFS với khoảng cách Euclid cho h(n)



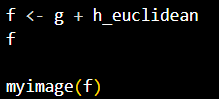
- Tương tự hàm Manhattan, ta tính khoảng cách bằng hàm Euclid từ điểm (x,y) đến goal:



- Biểu đồ:



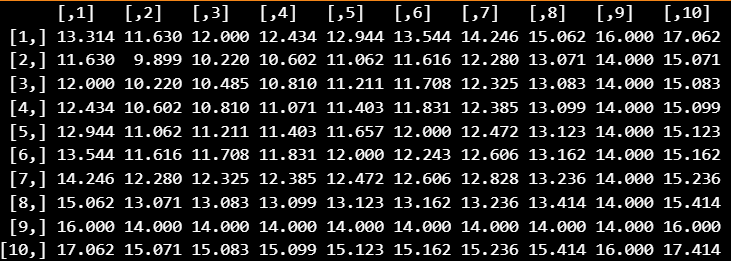
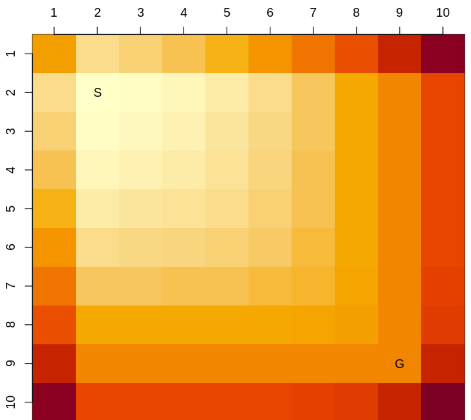
g) A\* với khoảng cách Euclid cho h(n)



- g: chi phí đi từ start đến (x,y)

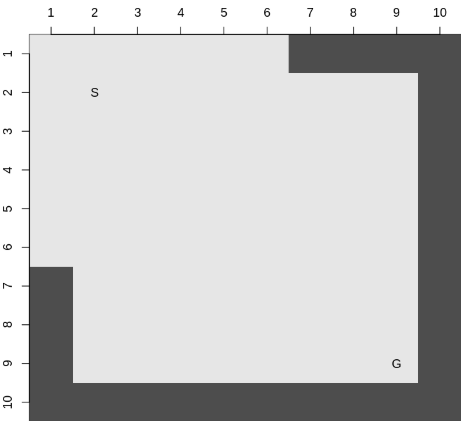
- h\_euclidean: chi phí ước lượng từ (x,y) đến goal theo khoảng cách Euclid

- f: tổng chi phí đi qua (x,y) nếu đi từ start->(x,y)->goal



- Dùng h\_euclidean, heuristic sẽ có dạng trong đều. So với Manhattan, đường đi dự đoán mượt hơn nhưng có thể không chính xác tuyệt đối nếu chi phí thật sự là Manhattan

- Thuật toán A\* sẽ mở rộng tất cả các nút có f(n) < C\* và một số nút có f(n) <= C\* được minh họa bởi biểu đồ bên dưới bằng màu xám nhạt

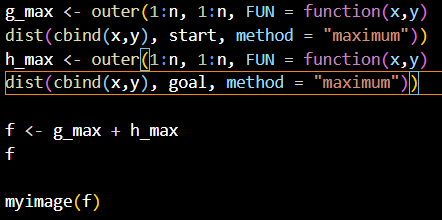




h) Di chuyển theo đường chéo với chi phí bằng một

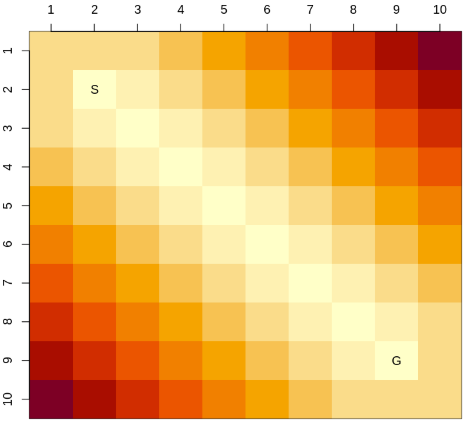
- Khi đó khoảng cách sẽ được đo bằng L∞,còn gọi là chuẩn cực đại (maximum norm), và được định nghĩa là:

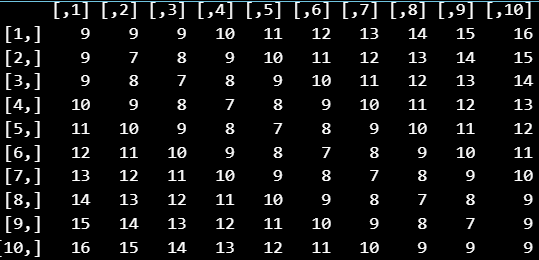
L∞(a,b)=max(∣ax​−bx∣,∣ay​−by∣).



- g\_max: chi phí đi từ start đến (x,y)

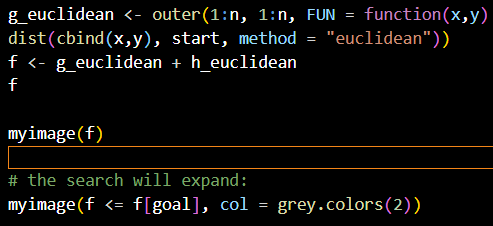
- h\_max: chi phí đi từ (x,y) đến goal

- f: là f-cost trong A\* với chuẩn L∞ đi từ start->(x,y)-> goal



i) Nếu có thể di chuyển bất kì khoảng cách nào theo bất kì hướng nào, ở bất kì góc nào

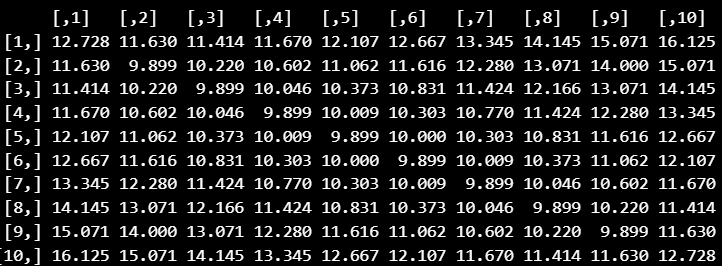
- Khoảng cách và chi phí sẽ được đo bằng khoảng cách Euclid. Các ô vuông lúc này chỉ là một lưới tìm kiếm để biểu diễn một vị trí, còn thực tế thì tất cả các vị trị ở giữa đều có thể di chuyển đến được. Một lưới lớn hơn sẽ cho hình ảnh trực quan chi tiết hơn

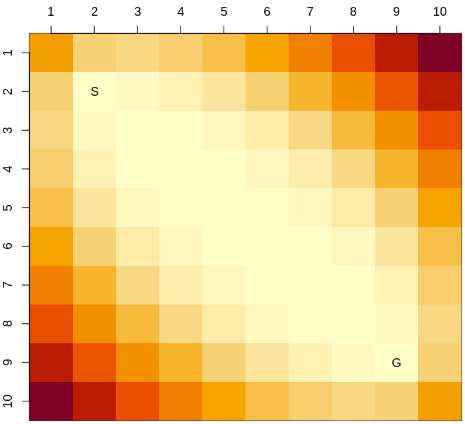
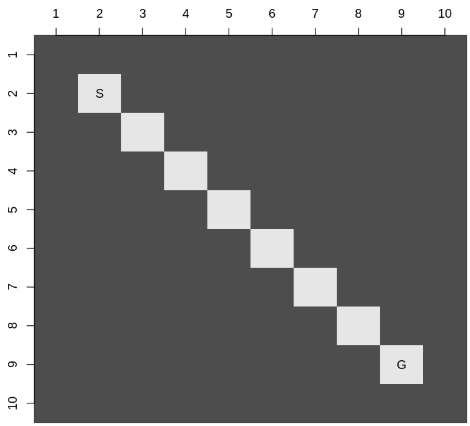


- g\_euclidean: khoảng cách Euclid từ mỗi ô (x,y) đến start

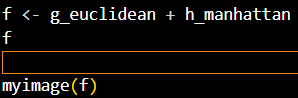


- f: tổng chi phí từ start -> (x,y) -> goal (theo mô hình di chuyển tự do) với chi phí Euclid



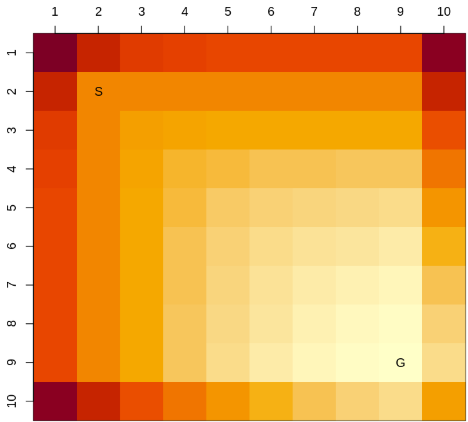


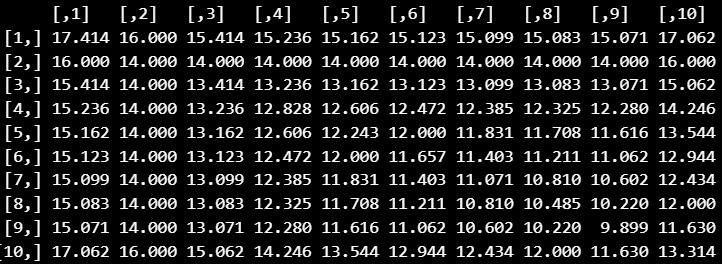
- Việc sử dụng khoảng cách Manhattan làm heuristic vẫn hoạt động tốt, nhưng A\* sẽ không được đảm bảo tối ưu, bởi vì đó không phải là một heuristic khả chấp — do khoảng cách Manhattan thường ước lượng vượt quá so với khoảng cách Euclid thực tế!



- g\_euclideean: chi phí từ start đến (x,y) bằng khoảng cách Euclid

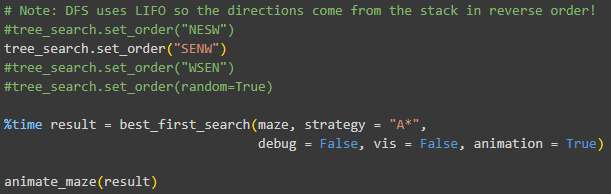
- h\_manhattan: chi phí từ (x,y) đến goal bằng khoảng cách Manhattan

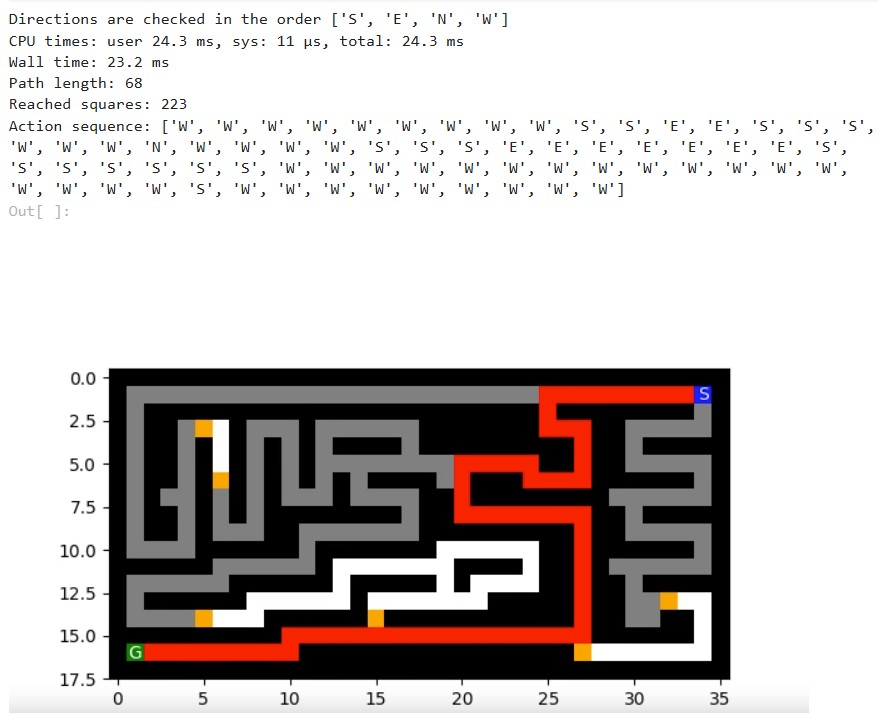
- f: tổng chi phí từ start -> (x,y) -> goal

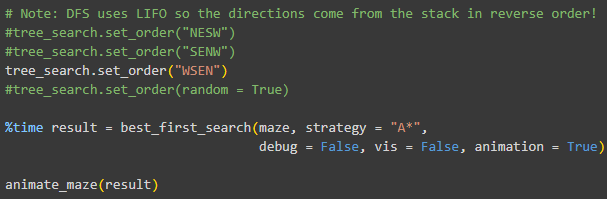


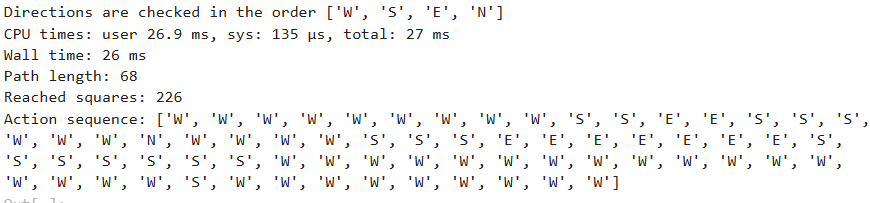
\* A\* Search

- thiết lập các hướng duyệt khác nhau sẽ có thể cho ra kết quả khác nhau

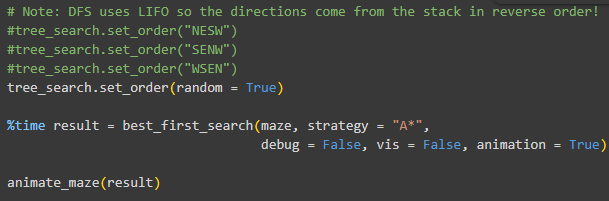


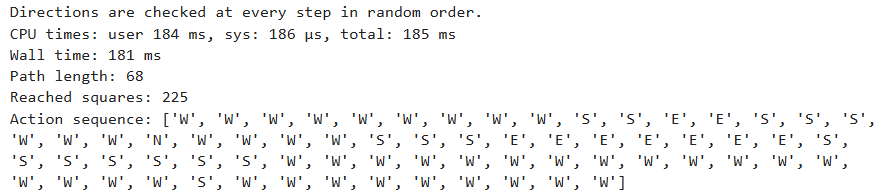








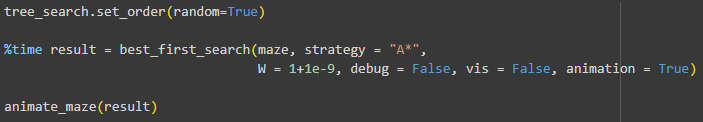


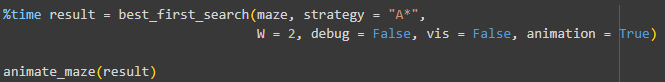


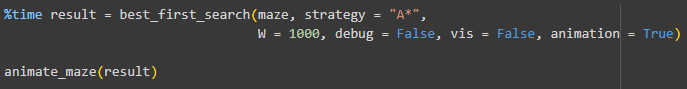


\* Weighted A\* Search(Trọng số của A\*)

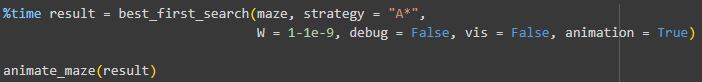
- W>1: thuật toán có xu hướng giống với GBFS (Greedy Best-First Search) và không còn đảm bảo tính tối ưu.

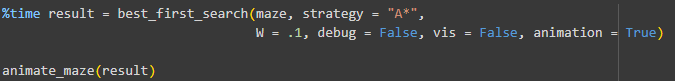






- W<1: sẽ có xu hướng giống với Uniform-Cost Search / BFS, và trong trường hợp này tính tối ưu được đảm bảo.





-> kết quả đường đi gần như tương tự nhau

# **5. File: Maze.ipynb.**

# **a) BÁO CÁO NHIỆM VỤ 1: ĐỊNH NGHĨA BÀI TOÁN TÌM KIẾM VÀ XÁC ĐỊNH KÍCH THƯỚC BÀI TOÁN.**

1. Đầu tiên định nghĩa các thành phần của bài toán tìm kiếm mê cung:

+ Trạng thái ban đầu: Vị trí xuất phát S, biểu diễn bằng tọa độ (row, col).

+ Hành động: Tập “North, South, East, West”, nghĩa là agent có thể di chuyển được lên, xuống, trái phải với điều kiện là ô trống không phải tường X.

+ Mô hình chuyển đổi: T(s, a) = s'. Ví dụ: đi North → (row-1, col).

+ Trạng thái mục tiêu: Vị trí đích G, kiểm tra bằng Goal\_Test(s) = True nếu s chứa G.

+ Chi phí đường đi: Mỗi bước có chi phí = 1, tổng chi phí là số bước từ S đến G.

+ Không gian trạng thái: Tập hợp các ô hợp lệ (không phải tường). Kích thước tối đa H × W.

1. Tiếp theo em sẽ nói về hàm **analyze\_maze\_size** (hàm analyze\_maze\_size để phân tích kích thước bài toán.):

A computer screen shot of text

AI-generated content may be incorrect.

+ Lấy kích thước mê cung bằng maze.shape.

+ Đếm số ô trống khác X để tính n (kích thước không gian trạng thái).

+ Tìm vị trí bắt đầu S và mục tiêu G.

+ Tính khoảng cách Manhattan để ước lượng độ sâu tối ưu d.

+ Xác định độ sâu tối đa m = số ô trống - 1.

+ Hệ số phân nhánh b = 4, vì mỗi ô có thể đi tối đa 4 hướng.

+ Hàm trả về các giá trị n, d, m, b để sử dụng cho phân tích.

+ Trong đó:  
+ n là số ô trống trong mê cung.  
+ d được ước tính bằng khoảng cách Manhattan từ S đến G.  
+ m là độ sâu tối đa trong trường hợp xấu nhất.  
+ b là hệ số phân nhánh tối đa bằng 4.

# **b) BÁO CÁO NHIỆM VỤ 2: TÌM KIẾM KHÔNG THÔNG TIN THEO CHIỀU RỘNG VÀ CHIỀU SÂU**

**Bước 1: Triển khai một hàm hỗ trợ**

**A computer screen shot of text

AI-generated content may be incorrect.**

+ Tạo hàm get\_neighbors(maze, pos) để tìm các vị trí có thể di chuyển. + Định nghĩa tập hành động ACTIONS với 4 hướng: North, South, East, West.  
+ Kiểm tra điều kiện hợp lệ: trong giới hạn mê cung và không phải tường.  
**Bước 2: Triển khai thuật toán BFS**

**A screen shot of a computer program

AI-generated content may be incorrect.**

+ Sử dụng deque (queue FIFO) cho frontier.  
+ Duy trì tập visited để theo dõi trạng thái đã thăm.  
+ Thu thập metrics: chi phí đường đi, số nút mở rộng, độ sâu tối đa, kích thước frontier.  
+ Trả về đường đi tối ưu (ngắn nhất).

**Bước 3: Triển khai thuật toán DFS**

**A screen shot of a computer program

AI-generated content may be incorrect.**

+ Sử dụng đệ quy.   
+Không dùng tập reached toàn cục để tiết kiệm bộ nhớ.  
+ Chỉ kiểm tra chu trình trên đường đi hiện tại.  
+ Đặt giới hạn độ sâu để tránh vòng lặp vô hạn.

**Bước 4: Tạo hàm kiểm thử và so sánh**

**A computer screen shot of a code

AI-generated content may be incorrect.**

+ Hàm print\_result() để hiển thị kết quả.  
+ So sánh hiệu suất giữa hai thuật toán.

**c) BÁO CÁO NHIỆM VỤ 3:  TÌM KIẾM CÓ THÔNG TIN Triển khai tìm kiếm tham lam theo chiều rộng tốt nhất và tìm kiếm A\*.  
Các công việc đã thực hiện**

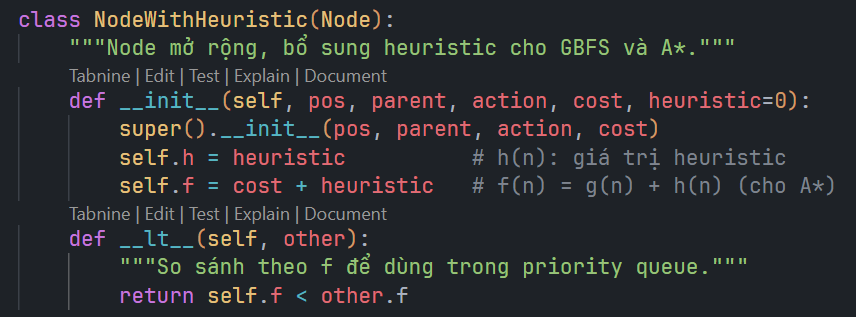
**Bước 1: Triển khai hàm Heuristic**

**A screen shot of a computer code

AI-generated content may be incorrect.**

+ Hàm manhattan\_distance(pos1, pos2): Tính khoảng cách Manhattan giữa 2 vị trí.  
+ Với công thức Công thức: |x1-x2| + |y1-y2|.  
+ Được sử dụng làm hàm heuristic h(n) cho cả GBFS và A\*.

**Bước 2: Tạo class NodeWithHeuristic**

****

+ Kế thừa từ class Node ban đầu  
+ Bổ sung thêm thuộc tính: h: giá trị heuristic h(n) và f: giá trị f(n) = g(n) + h(n) cho A\*.  
+ Triển khai \_\_lt\_\_() để so sánh trong priority queue.

**Bước 3: Triển khai Greedy Best-First Search (GBFS)**

**A screen shot of a computer program

AI-generated content may be incorrect.**

+ Với nguyên tắc ưu tiên nút có h(n) nhỏ nhất.  
+ Và cấu trúc Priority queue (heapq) với key = h(n). Tập reached để tránh thăm lại.  
+ Chỉ xem xét heuristic, bỏ qua chi phí thực tế g(n).

**Bước 4: Triển khai A\***

**A screen shot of a computer program

AI-generated content may be incorrect.**

+ Với nguyên tắc ưu tiên nút có  f(n) = g(n) + h(n) nhỏ nhất.  
+ Và cấu trúc Priority queue với key = f(n). Dictionary reached lưu chi phí tốt nhất g(n) cho mỗi trạng thái.  
+ Tối ưu hóa: kiểm tra và cập nhập đường đi tốt hơn. Và bỏ qua nút nếu đã có đường đi rẻ hơn.

**Bước 5: Hàm kiểm thử và so sánh**

**A screen shot of a computer code

AI-generated content may be incorrect.**+ print\_result(): Hiển thị kết quả từng thuật toán.  
+ Bảng so sánh đầy đủ các metrics.  
+ Phân tích ưu nhược điểm

# **d) BÁO CÁO NHIỆM VỤ 4:  SO SÁNH VÀ THẢO LUẬN**

**Các công việc đã thực hiện:**

**Bước 1: Xấy dựng các hàm so sánh**

+ Hàm run\_all(maze): Chạy tất cả 4 thuật toán (BFS, DFS, GBFS, A\*) trên cùng 1 mê cung.  
+ Hàm compare(results): Tạo bảng so sánh chi tiết và phân tích.  
+ Hàm visualize\_path(): Hiển thị đường đi trên mê cung.  
+ Hàm print\_result(): Hiển thị kết quả từng thuật toán.

**Bước 2: Thu thập đầy đủ các metrics  
Chi phí đường đi (số bước)**

+ Số nút mở rộng (nodes expanded).  
+ Độ sâu tối đa (max depth).  
+ Kích thước frontier tối đa (max frontier size).  
+ Ước tính nút tối đa trong bộ nhớ.

**6. File: Show\_all\_mazes.ipynb**

-Mục đích: File này được dùng để hiển thị và trực quan hóa các mê cung (mazes) khác nhau trong bài toán tìm kiếm đường đi. Nó giúp ta kiểm tra, quan sát cấu trúc mê cung và vị trí Start/Goal trước khi áp dụng các thuật toán tìm kiếm.

**Bước 1: Chuẩn bị môi trường**

A screen shot of a computer

AI-generated content may be incorrect.  
+ Sử dụng %matplotlib inline để vẽ trực tiếp trong notebook.  
+ %config InlineBackend.figure\_format = 'retina': cấu hình hình ảnh hiển thị rõ nét hơn (độ phân giải cao).  
+ import maze\_helper as mh: nạp file/module maze\_helper để xử lý và vẽ hiển thị mê cung.  
=> Cơ bản đoạn này chuẩn bị môi trường hiển thị đồ họa và nạp thư viện phụ trợ để làm việc với mê cung.

**Bước 2: Tìm danh sách đọc và hiển thị mê cung**

A screenshot of a computer program

AI-generated content may be incorrect.

+ Viết hàm list\_mazes(directory) duyệt qua thư mục, lấy tất cả file có đuôi .txt. Sắp xếp danh sách để hiển thị có thứ tự.  
+ mazes = list\_mazes("."): gọi hàm trên, lấy danh sách file mê cung trong thư mục hiện tại.  
+ for maze\_file in mazes:: lặp qua từng file mê cung:  
+ with open(maze\_file, "r") as f:: mở file đọc nội dung.  
+ maze\_str = f.read(): lấy toàn bộ dữ liệu mê cung dạng chuỗi.  
+ maze = mh.parse\_maze(maze\_str): dùng hàm trong maze\_helper để chuyển chuỗi thành cấu trúc mê cung.  
+ print(maze\_file): in tên file mê cung.  
+ mh.show\_maze(maze): hiển thị hình ảnh mê cung bằng thư viện phụ trợ.

=>Cơ bản đoạn này tìm tất cả các file mê cung .txt, đọc chúng, rồi vẽ từng cái ra.