TRƯỜNG ĐẠI HỌC SÀI GÒN
KHOA CÔNG NGHỆ THÔNG TIN
---o0o---

BÁO CÁO BÀI TẬP NHÓM

LAB - 02

Giảng viên hướng dẫn: TS. Đỗ Như Tài
Môn học: Trí tuệ nhân tạo nâng cao
Nhóm thực hiện: 7
Danh sách thành viên:

3122410489 – Lê Huỳnh Trúc Vy
3122410495 – Trần Mỹ Yên
3120410470 – Lê Quốc Thái
3122410174 – Thái Minh Khang

TP. HỒ CHÍ MINH, THÁNG 10 NĂM 2025

MỤC LỤC

BẢNG PHÂN CÔNG 4

SOLVING A MAZE USING A GOAL-BASED AGENT (TÌM KIẾM: GIẢI MÊ CUNG BẰNG TÁC NHÂN DỰA TRÊN MỤC TIÊU) 5

I. MỤC TIÊU. 5

II. GIỚI THIỆU. 5

III. NHIỆM VỤ. 6

1. Nhiệm vụ 1: 7

2. Nhiệm vụ 2: Tìm kiếm không thông tin trước: Tìm kiếm theo chiều rộng và chiều sâu 9

3. Nhiệm vụ 3: 14

4. Nhiệm vụ 4: 24

5. Nhiệm vụ nâng cao: 26

6. Nhiệm vụ nâng cao: 30

INTELLIGENT AGENTS: REFLEX-BASED AGENTS FOR THE VACUUM-CLEANER WORLD (TÁC NHÂN THÔNG MINH: TÁC NHÂN DỰA TRÊN PHẢN XẠ CHO THẾ GIỚI MÁY HÚT BỤI) 33

I. MỤC TIÊU: 33

II. GIỚI THIỆU 33

III. MÔ TẢ PEAS CHO GIAI ĐOẠN DỌN DẸP 33

IV. CHƯƠNG TRÌNH TÁC NHÂN CHO TÁC NHÂN NGẪU NHIÊN ĐƠN GIẢN . 34

V. NHIỆM VỤ: 34

1. Nhiệm vụ 1: 34

2. Nhiệm vụ 2: Triển khai tác nhân phản xạ đơn giản 35

3. Nhiệm vụ 3: 38

4. Nhiệm vụ 4: 49

5. Nhiệm vụ 5: 51

6. Nhiệm vụ nâng cao: Cảm biến bụi bẩn không hoàn hảo 53

7. Nhiệm vụ nâng cao: 56

CREATE A SIMPLE REFLEX-BASED LUNAR LANDER AGENT (XÂY DỰNG AGENT LUNAR LANDER DỰA TRÊN PHẢN XẠ ĐƠN GIẢN) 59

I. Mục tiêu. 59

II. Giới thiệu. 59

III. Cơ sở lý thuyết. 60

1. Khái niệm Agent 60

2. Simple Reflex Agent 60

3. Môi trường Lunar Lander 60

4. Nguyên lý hoạt động của Simple Reflex Agent trong Lunar Lander 61
IV. Kết luận 62

BẢNG PHÂN CÔNG

Mã số sinh viên	Họ tên	Công việc
3122410489	Lê Huỳnh Trúc Vy (Nhóm trưởng)	Maze.ipynb (Task 1, Advanced task) Lunar_lander.ipynb (Tasks 1, More Advanced) Robot_vacuum.ipynb (More Advanced)
3122410495	Trần Mỹ Yên	Maze.ipynb (Task 2, Advanced task) Lunar_lander.ipynb (Tasks 2, Advanced task)
3122410174	Thái Minh Khang	Maze.ipynb (Task 4, More Advanced) Lunar_lander (Tasks 4)
3120410470	Lê Quốc Thái	Maze.ipynb (Task 3, More Advanced) Lunar_lander.ipynb (Tasks 3, Task 5)

SOLVING A MAZE USING A GOAL-BASED AGENT (TÌM KIẾM: GIẢI MÊ CUNG BẰNG TÁC NHÂN DỰA TRÊN MỤC TIÊU)

I. MỤC TIÊU.

- Xây dựng bài toán tìm kiếm sử dụng các thành phần chính như trạng thái ban đầu, hành động và trạng thái mục tiêu trong môi trường xác định và quan sát được hoàn toàn.
- Triển khai và so sánh các thuật toán tìm kiếm bao gồm BFS, DFS, GBFS, A* và IDS để tìm đường trong các mê cung
- Phân tích hiệu suất thuật toán bằng cách đo lường chi phí đường đi, số nút mở rộng, độ sâu và dung lượng bộ nhớ trên nhiều loại mê cung khác nhau.
- Sử dụng các công cụ trực quan hóa để biểu diễn các đường đi trong mê cung, hỗ trợ gỡ lỗi và phân tích.

II. GIỚI THIỆU.

- Tác nhân phải sử dụng bản đồ được cung cấp để lập kế hoạch đường đi trong mê cung, từ vị trí bắt đầu S đến vị trí mục tiêu G. Đây là một bài tập lập kế hoạch cho một tác nhân dựa trên mục tiêu, vì vậy bạn không cần phải triển khai một môi trường; chỉ cần sử dụng bản đồ để tìm kiếm một đường đi. Một khi kế hoạch được lập ra, tác nhân trong một môi trường xác định (nghĩa là, hàm chuyển tiếp xác định với kết quả của mỗi cặp trạng thái/hành động đã được cố định và không có yếu tố ngẫu nhiên) có thể chỉ cần đi theo đường đi đó và không cần quan tâm đến tri giác. Điều này còn được gọi là một hệ thống vòng hở (open-loop system).
- Giai đoạn thực thi là rất đơn giản và có thể được thực hiện bằng một tác nhân phản xạ dựa trên mô hình mà bỏ qua tất cả tri giác và chỉ đi theo kế hoạch. Chúng ta không triển khai giai đoạn này trong bài tập này.
- Với giả định rằng tác nhân có một bản đồ hoàn chỉnh và chính xác, môi trường là quan sát được hoàn toàn, rời rạc, xác định và đã biết.
- Lưu ý:

- Quan sát được hoàn toàn (Fully observable) có nghĩa là tác nhân có thể thấy trạng thái của nó và các hành động có sẵn. Điều đó có nghĩa là tri giác chứa trạng thái hiện tại hoàn chỉnh**.** Ở đây, trong quá trình lập kế hoạch, tác nhân luôn thấy tọa độ x và y của nó trên bản đồ và cũng tìm kiếm khi nó đã đạt đến trạng thái mục tiêu.
- Rời rạc (Discrete) có nghĩa là chúng ta có một tập hợp hữu hạn các trạng thái. Mê cung có một tập hợp hữu hạn các ô vuông mà tác nhân có thể ở.
- Xác định (Deterministic) có nghĩa là hàm chuyển tiếp không chứa yếu tố ngẫu nhiên. Một hành động trong một trạng thái sẽ luôn tạo ra cùng một kết quả. Đi về phía nam từ trạng thái bắt đầu luôn sẽ dẫn đến cùng một ô vuông.
- Đã biết (Known) có nghĩa là tác nhân biết hàm chuyển tiếp hoàn chỉnh. Tác nhân có bản đồ và do đó biết vị trí của nó thay đổi như thế nào khi nó đi theo một hướng.

- Các triển khai thuật toán tìm kiếm trên cây đến từ các khóa học cấu trúc dữ liệu và có mục tiêu khác với tìm kiếm trên cây trong AI. Các thuật toán này giả định rằng đã có sẵn một cây trong bộ nhớ. Quan tâm đến việc tạo động một cây tìm kiếm với mục đích tìm ra một đường đi tốt/tốt nhất từ nút gốc đến trạng thái mục tiêu. Lý tưởng nhất, muốn chỉ tìm kiếm một phần nhỏ của mê cung, nghĩa là tạo ra một cây tìm kiếm với số nút ít nhất có thể.

III. NHIỆM VỤ.

Mục tiêu là:

- Triển khai các thuật toán tìm kiếm sau để giải các mê cung khác nhau:
  + Tìm kiếm theo chiều rộng (Breadth-first search – BFS)
  + Tìm kiếm theo chiều sâu (Depth-first search – DFS)
  + Tìm kiếm tham lam theo tốt nhất (Greedy best-first search – GBFS)
  + Tìm kiếm A* (A* search)
- Chạy từng thuật toán trên các mê cung:
  + small maze
  + medium maze
  + large maze

+ open maze
+ wall maze
+ loops maze
+ empty maze
+ empty 2_maze

- Đối với mỗi bài toán và mỗi thuật toán tìm kiếm, báo cáo các thông số sau trong bảng:
  + Lời giải và chi phí đường đi của nó
  + Tổng số node đã được mở rộng
  + Độ sâu tối đa của cây tìm kiếm
  + Kích thước tối đa của frontier
- Hiển thị mỗi lời giải bằng cách đánh dấu mọi ô (hoặc trạng thái) đã đi qua và các ô nằm trên đường đi cuối cùng.

1. Nhiệm vụ 1:

1. Trạng thái ban đầu (Initial state): $s_0 \in S$ là ô nơi tác nhân bắt đầu (ví dụ: tọa độ (r_0, c_0))
2. Hành động (Actions)
   - Tập hành động A= {Up, Down, Left, Right} (hoặc một tập con nếu cấm chéo).
   - Hành động có thể khả dụng tại một trạng thái nếu ô đích không phải là tường và nằm trong biên giới mê cung.
3. Mô hình chuyển tiếp (Transition model)
   - Hàm chuyển tiếp $T(s, a) = s'$ trả về trạng thái mới khi thực hiện hành động a ở trạng thái s.
   - Với đồ thị mê cung 4 - lân cận: nếu $s = (r, c)$ và a = Up thì $s' = (r - 1, c)$ nếu ô đó rỗng; nếu là tường hoặc ngoài biên giới thì hành động không hợp lệ (ta có thể định nghĩa $T(s, a)$ = null hoặc không đưa hành động đó vào danh sách khả dụng)
   - Nếu có chi phí khác nhau, thì chuyển tiếp có thêm chi phí $c(s, a', s')$
4. Trạng thái mục tiêu (Goal state)

- Tập mục tiêu $G \subseteq S$. Thông thường G là một ô đích đơn g (ví dụ: tọa độ (r_g, c_g))
- Mục tiêu đạt được khi $s = g$

5. Chi phí đường đi (Path cost)
   - Hàm chi phí $C(\text{path}) = \sum c(s_i, a_i, s_i+1)$
   - Với chi phí từng bước đồng nhất: mỗi bước có chi phí 1 => chi phí đường đi là số bước trên đường đi.
   - Nếu di chuyển chéo hoặc ô có chi phí khác nhau thì dùng tổng các chi phí từng cạnh.
6. n: kích thước không gian trạng thái - số trạng thái có thể đạt được (số ô trống có thể đi).
   - Phương pháp thực tế: đếm số ô không phải tường trong grid (số ô "rỗng" hoặc "free") $n=\#\{(r, c) \mid$ ô (r, c) là free$\}$
   - Nếu muốn loại trừ các ô không thể tiếp cận từ trạng thái bắt đầu (bị cách ly), thì chạy một flood fill/BFS từ s_0 và đếm các ô đạt được - đó là kích thước không gian trạng thái có thể tiếp cận
7. d: độ sâu có nghiệm tối ưu - số bước trong đường đi ngắn nhất từ s_0 tới mục tiêu (nếu chi phí 1/bước thì tương đương chi phí)
   - Nếu chi phí mỗi bước là 1, thì d là độ dài đường đi ngắn nhất (số bước).
   - Cách chính xác: chạy BFS (vì BFS cho đồ thị không trọng số trả về đường ngắn nhất) từ $s0$ tới g. Gía trị trả về là d.
   - Làm công thức / giới hạn: một lower bound có thể là khoảng cách Manhattan nếu không có chướng ngại $d \geq |r0 - rg| + |c0 - cg|$
8. m: độ sâu cực đại của cây tìm kiếm.
   - Có thể hiểu là giới hạn sâu nhất mà thuật toán có thể mở rộng (ví dụ giới hạn vòng lận, hoặc số bước tối đa).
   - m là độ sâu lớn nhất có thể xuất hiện trong cây tìm kiếm. Nếu ta cho phép chỉ đường đi không lặp (simple path), thì tối đa có n - 1 bước (đi qua tất cả các ô khác): $m \leq n - 1$
9. b: hệ số phân nhánh tối đa - số hành động khả dụng lớn nhất từ một trạng thái.

- Với lân cận 4 hướng (Up/Down/Left/Right), ta có $b_{max} = 4$
- Tuy nhiên ở biên hoặc gần tường số hành động khả dụng ít hơn. Vì thế:

$$b = \max_{s \in S} \deg(s)$$

trong đó deg(s) = số ô kề với *s* và đi được.

- Ta có thể tính *b* bằng cách duyệt tất cả các ô rỗng và lấy giá trị lớn nhất của count(neighbors).
- Để ước lượng trung bình, ta có thể tính:

$$\bar{b} = \frac{1}{n} \sum_{s \in S} \deg(s)$$

2. Nhiệm vụ 2: Tìm kiếm không thông tin trước: Tìm kiếm theo chiều rộng và chiều sâu

a. Tổng quan

- Nhiệm vụ này tập trung vào việc triển khai hai trong số các thuật toán tìm kiếm không thông tin trước quan trọng nhất: Tìm kiếm theo Chiều rộng (BFS - Breadth-First Search) và Tìm kiếm theo Chiều sâu (DFS - Depth-First Search). Các thuật toán này được sử dụng để tìm đường đi từ một điểm bắt đầu (`S`) đến một điểm đích (`G`) trong một mê cung.

b. Cấu trúc triển khai

- Để thực hiện tìm kiếm, một cấu trúc dữ liệu cơ bản đã được tạo:
- Lớp Node: Đại diện cho một nút trong cây tìm kiếm. Mỗi nút lưu trữ ba thông tin chính:
  + state: Vị trí hiện tại của tác nhân (dưới dạng một cặp (hàng, cột)).
  + parent: Nút cha trong cây tìm kiếm, cho phép chúng ta xây dựng lại đường đi sau khi tìm thấy lời giải.
  + action: Hành động đã thực hiện để đến được trạng thái này.
- Các hàm trợ giúp cũng được triển khai để xử lý các tác vụ chung:
- find_start_and_goal(maze): Quét qua mê cung để xác định vị trí ban đầu và vị trí mục tiêu.

- reconstruct_path(node): Theo dõi các nút cha từ nút đích trở ngược về nút gốc để xây dựng lại đường đi hoàn chỉnh.

c. Triển khai Thuật toán Tìm kiếm theo Chiều rộng (BFS)

- Thuật toán BFS được triển khai trong hàm bfs_search(maze). Đây là một thuật toán tìm kiếm hoàn chỉnh và tối ưu (đối với các bước có cùng chi phí).
- Frontier (Biên): BFS sử dụng một hàng đợi (collections.deque) để lưu trữ các nút cần được khám phá. Việc sử dụng hàng đợi đảm bảo rằng các nút được mở rộng theo thứ tự FIFO (First-In, First-Out), ưu tiên các nút ở độ sâu thấp hơn.
- Set Khám phá (explored): Một tập hợp (set) có tên explored được sử dụng để lưu trữ tất cả các trạng thái đã được ghé thăm. Điều này ngăn chặn việc lặp lại trạng thái và rơi vào vòng lặp vô hạn, giúp thuật toán hiệu quả hơn.
- Logic Hoạt động:
  + Khởi tạo hàng đợi với nút bắt đầu và thêm trạng thái bắt đầu vào tập hợp explored.
  + Trong khi hàng đợi không rỗng, lấy ra nút đầu tiên.
  + Kiểm tra xem trạng thái của nút đó có phải là mục tiêu hay không. Nếu có, đường đi được xây dựng lại và trả về.
  + Nếu không phải, tìm tất cả các trạng thái láng giềng hợp lệ (không phải là tường và chưa được khám phá).
  + Đối với mỗi láng giềng hợp lệ, tạo một nút mới, thêm nó vào cuối hàng đợi và thêm trạng thái của nó vào tập hợp explored.
- Phân tích và Ưu điểm:
  + Hoàn chỉnh: BFS luôn tìm thấy lời giải nếu có tồn tại.
  + Tối ưu: BFS luôn tìm thấy đường đi ngắn nhất (về số bước) từ nút bắt đầu đến nút đích.
  + Nhược điểm: BFS có thể tốn rất nhiều bộ nhớ. Vì nó phải lưu trữ tất cả các nút ở một độ sâu nhất định trước khi chuyển sang độ sâu tiếp theo, bộ nhớ sử dụng có thể tăng theo cấp số nhân với kích thước mê cung.

d. Triển khai Thuật toán Tìm kiếm theo Chiều sâu (DFS)

- Thuật toán DFS được triển khai trong hàm dfs_search(maze). DFS là một chiến lược tìm kiếm không thông tin trước đi sâu vào một nhánh của cây tìm kiếm càng xa càng tốt trước khi quay lui.
- Khác với BFS, DFS sử dụng một ngăn xếp (frontier) để lưu trữ các nút cần được khám phá. Bằng cách sử dụng phương thức pop() thay vì popleft(), chúng ta đảm bảo rằng các nút được xử lý theo thứ tự LIFO (Last-In, First-Out). Điều này cho phép thuật toán đi sâu vào các nút con mới được thêm vào trước khi xử lý các nút ở độ sâu nông hơn.
- Xử lý bộ nhớ và chu trình:
  + Bộ nhớ: Để tận dụng ưu điểm tiết kiệm bộ nhớ của DFS, việc triển khai này không sử dụng một set explored toàn cục. Thay vào đó, nó sử dụng một set path_states chỉ để theo dõi các trạng thái trên đường đi hiện tại từ gốc đến nút đang xét. Điều này giúp tránh rơi vào các vòng lặp vô hạn mà không cần lưu trữ thông tin của toàn bộ các nút đã được ghé thăm trong quá trình tìm kiếm.
  + Hoàn chỉnh: Mặc dù DFS có thể không hoàn chỉnh trên các không gian trạng thái vô hạn, trong môi trường mê cung hữu hạn và có kiểm tra chu trình, nó được đảm bảo sẽ tìm thấy lời giải nếu có.
  + Tối ưu: DFS không tối ưu về chi phí đường đi. Nó có thể tìm thấy một đường đi rất dài đến mục tiêu trước khi khám phá một đường đi ngắn hơn nhiều.

e. Thảo luận

- Việc thực hiện các thuật toán tìm kiếm như BFS (Tìm kiếm theo chiều rộng) và DFS (Tìm kiếm theo chiều sâu) mà không sử dụng cấu trúc dữ liệu để theo dõi các nút đã được thăm (reached hoặc explored) sẽ dẫn đến những vấn đề nghiêm trọng khi gặp các vòng lặp (chu trình) trong mê cung hoặc đồ thị.
- Vấn đề với BFS và DFS khi không có “Reached”
- Vòng lặp vô hạn: Nếu có một vòng lặp, thuật toán sẽ liên tục đi qua các nút trong vòng lặp đó và không bao giờ kết thúc.

+ BFS: Hàng đợi (queue) sẽ liên tục được thêm vào các nút đã thăm lại, làm cho kích thước hàng đợi tăng lên không giới hạn.
+ DFS: Ngăn xếp (stack) sẽ tăng lên không giới hạn khi thuật toán đi sâu vào vòng lặp, dẫn đến lỗi tràn bộ nhớ (stack overflow).

- Lặp lại công việc: Các thuật toán sẽ liên tục thăm lại và xử lý các nút đã được khám phá trước đó, gây lãng phí tài nguyên tính toán.
- Không tìm thấy giải pháp: Thuật toán có thể bị kẹt trong một vòng lặp và không bao giờ khám phá được phần còn lại của mê cung hoặc đồ thị, ngay cả khi có đường đi đến đích.
- Tầm quan trọng của cấu trúc dữ liệu "Reached": Cấu trúc dữ liệu reached (thường là một tập hợp - set) là bắt buộc để giải quyết các vấn đề trên. Nó hoạt động như một bộ nhớ cho thuật toán:
  + Mỗi khi một nút mới được khám phá, nó sẽ được thêm vào tập hợp reached.
  + Trước khi xử lý một nút, thuật toán sẽ kiểm tra xem nút đó đã có trong reached chưa.
  + Nếu nút đó đã có, thuật toán sẽ bỏ qua nó, ngăn chặn việc thăm lại và mắc kẹt trong vòng lặp.
- Nhờ có cấu trúc này, BFS và DFS có thể tìm kiếm hiệu quả và đảm bảo rằng chúng sẽ chấm dứt ngay cả trong các đồ thị có chu trình.
- Việc triển khai đã được thiết kế để chính xác và đầy đủ theo yêu cầu của bài toán, tuân thủ các nguyên tắc cốt lõi của thuật toán Tìm kiếm theo chiều rộng (BFS). Tuy nhiên, nó không phải là triển khai tối ưu nhất trong mọi tình huống. Nó ưu tiên sự rõ ràng và tính đúng đắn hơn là tối ưu hóa hiệu suất cực đoan.
- Đánh giá triển khai:
  + Tính đúng đắn và đầy đủ: Thuật toán BFS được triển khai một cách chuẩn mực, sử dụng một hàng đợi (collections.deque) để khám phá các nút và một tập hợp đã thăm (explored) để ngăn chặn việc lặp lại các nút. Lớp Node giúp lưu lại đường đi một cách hiệu quả để có thể

tái tạo lại sau khi tìm thấy đích. Điều này đảm bảo rằng thuật toán sẽ tìm ra đường đi ngắn nhất nếu có.

+ Điểm chưa tối ưu: Mặc dù đúng, việc lưu trữ toàn bộ lịch sử đường đi trong mỗi Node có thể tốn kém bộ nhớ, đặc biệt là trong các mê cung lớn. Đối với các bài toán quy mô lớn, việc tối ưu hóa bộ nhớ có thể là một mối quan tâm lớn.

- Độ phức tạp về thời gian và không gian:
- Để phân tích độ phức tạp, ta sử dụng các ký hiệu sau:
  + V: số lượng đỉnh (các ô trống có thể đi qua trong mê cung)
  + E: số lượng cạnh (các kết nối giữa các ô)
  + b: Hệ số phân nhánh (số lượng đường đi có thể có từ mỗi nút, thường là 4)
  + d: Độ sâu của đường đi ngắn nhất
  + m: Chiều sâu lớn nhất của không gian tìm kiếm

*Triển khai BFS

- Độ phức tạp thời gian :O(V+E)
- Trong mê cung dạng lưới, mỗi đỉnh có một số lượng cạnh cố định (tối đa 4), nên E tỷ lệ với V. Do đó, độ phức tạp là O(V). BFS phải thăm tất cả các ô có thể đi qua trong trường hợp xấu nhất.
- Độ phức tạp không gian: O(V)
  + Không gian lưu trữ chủ yếu đến từ hàng đợi và tập hợp đã thăm (explored).
  + Trong trường hợp xấu nhất (mê cung rỗng), hàng đợi có thể chứa tất cả các nút ở một cấp độ duy nhất, dẫn đến không gian bộ nhớ tăng theo cấp số mũ với độ sâu (O(b^d)).Tuy nhiên, trong mê cung, số nút bị giới hạn bởi V, nên độ phức tạp không gian là O(V).

Tiêu chí	BFS(với ‘reached’	DFS (với ‘reached’)
Độ phức tạp không gian	O(V) hoặc O(b^d)	O(m)

Lý do	Hàng đợi của BFS có thể chứa một số lượng lớn nút ở cùng một cấp độ. Trong trường hợp xấu nhất, nó phải lưu trữ tất cả các nút ở cấp độ cuối cùng, khiến chi phí không gian tăng theo cấp số nhân với độ sâu của giải pháp (O(b^ d)) và có thể rất lớn đối với các bài toán có giải pháp sâu.	Ngăn xếp của DFS chỉ lưu trữ các nút trên đường đi hiện tại từ gốc đến nút đang được thăm, không phải tất cả các nút đã thăm. Chi phí không gian chỉ tăng tuyến tính với độ sâu lớn nhất của không gian tìm kiếm (O(m)), thường nhỏ hơn đáng kể so với BFS.

- Tóm lại: BFS đảm bảo tìm thấy đường đi ngắn nhất nhưng phải trả giá bằng việc sử dụng nhiều bộ nhớ hơn. Ngược lại, DFS sử dụng ít bộ nhớ hơn đáng kể, làm cho nó trở thành lựa chọn tốt hơn cho các bài toán có chiều sâu lớn, mặc dù nó không đảm bảo tìm thấy đường đi tối ưu.

3. Nhiệm vụ 3:

1. Mục tiêu

- Nhiệm vụ của task này là cài đặt và so sánh hai thuật toán tìm kiếm có thông tin (informed search) là Greedy Best-First Search (GBFS) và A* Search. Cả hai thuật toán được áp dụng để tìm đường đi ngắn nhất từ điểm bắt đầu (S) đến điểm kết thúc (G) trong một mê cung. Hàm heuristic được sử dụng để ước tính khoảng cách từ một ô bất kỳ đến đích là khoảng cách Manhattan.

2. Phương pháp cài đặt

- Để cài đặt hai thuật toán, một cấu trúc giải thuật tìm kiếm dựa trên hàng đợi ưu tiên (Priority Queue) đã được xây dựng.
- Node: Một lớp Node được tạo để lưu trữ thông tin của mỗi ô trong quá trình tìm kiếm, bao gồm:

  + position: Tọa độ (hàng, cột) của ô.

+ parent: Node cha đã dẫn đến node hiện tại (dùng để truy vết đường đi).

+ g_cost: Chi phí thực tế từ điểm bắt đầu đến node hiện tại (số bước đã đi).

+ h_cost: Chi phí ước tính (heuristic) từ node hiện tại đến đích, được tính bằng khoảng cách Manhattan.

+ f_cost: Tổng chi phí g_cost + h_cost, chỉ dùng cho A*.

+ Hàm Heuristic: Hàm manhattan_distance được cài đặt để tính h_cost.

h(n) = |n.x - goal.x| + |n.y - goal.y|

- Cấu trúc dữ liệu:

+ Frontier: Sử dụng một hàng đợi ưu tiên (min-heap từ thư viện heapq của Python) để lưu trữ các node sẽ được khám phá. Node có độ ưu tiên cao nhất (giá trị nhỏ nhất) sẽ được lấy ra trước.

+ Explored Set: Một set được sử dụng để lưu các vị trí đã được khám phá, nhằm tránh việc xử lý lặp lại và các vòng lặp vô hạn.

* Sự khác biệt giữa hai thuật toán:

- Greedy Best-First Search: Độ ưu tiên của một node trong frontier được quyết định hoàn toàn bởi giá trị heuristic: priority = h(n). Thuật toán này có xu hướng "tham lam", luôn ưu tiên mở rộng node mà nó tin là gần đích nhất.

- A* Search: Độ ưu tiên được quyết định bởi tổng chi phí f(n) = g(n) + h(n). A* cân bằng giữa chi phí đã đi (g(n)) và chi phí ước tính còn lại (h(n)), giúp nó tìm ra đường đi tối ưu.

3. Kết quả thực nghiệm

- Các thuật toán đã được chạy trên nhiều loại mê cung khác nhau để đánh giá hiệu suất và tính chính xác. Các chỉ số được so sánh bao gồm:

- Độ dài đường đi: Số bước đi từ S đến G.

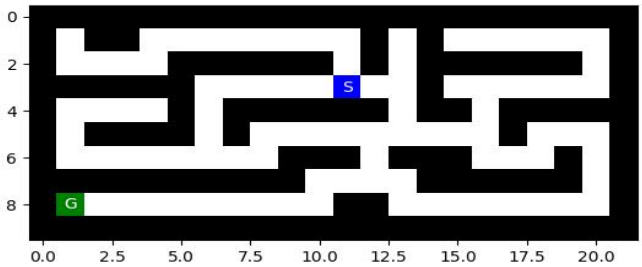
- Số nút đã duyệt: Số lượng ô đã được lấy ra khỏi frontier và xử lý. Chỉ số này thể hiện mức độ hiệu quả của việc tìm kiếm.

3.1. Small Maze (small_maze.txt)

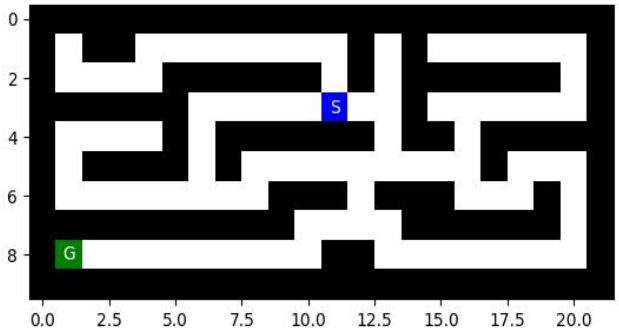
- Mê cung nhỏ với một vài chướng ngại vật đơn giản.

Thuật toán	Độ dài đường đi	Số nút đã duyệt

Greedy BFS	29	40
A* Search	19	54



Hình 3.1. Hình ảnh mô phỏng thuật toán Greedy BFS trong dữ liệu small_maze.txt

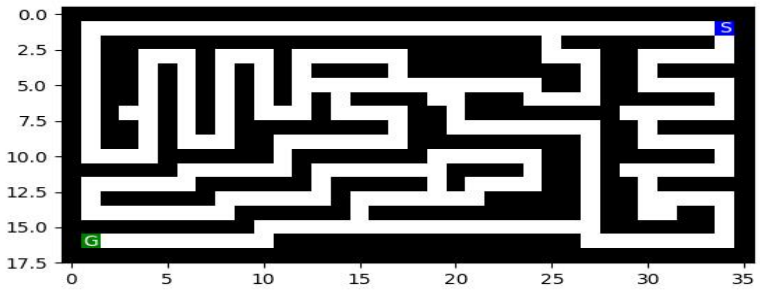


Hình 3.2. Hình ảnh mô phỏng thuật toán A Search trong dữ liệu small_maze.txt*

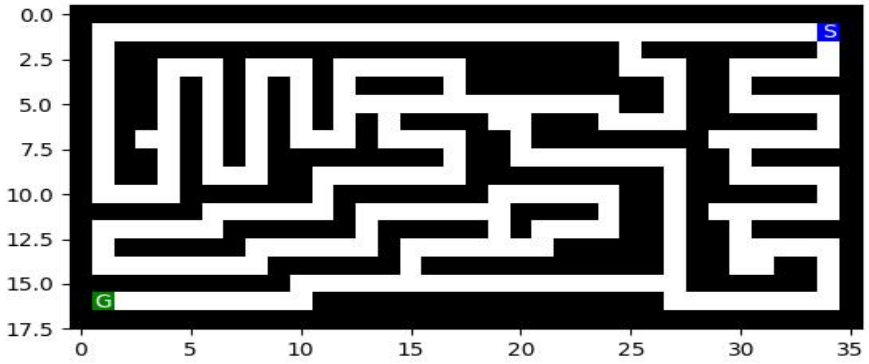
3.2. Medium Maze (medium_maze.txt)

- Mê cung cỡ trung bình với một con đường dài, quanh co. Đây là một ví dụ điển hình để thấy sự khác biệt giữa GBFS và A*.

Thuật toán	Độ dài đường đi	Số nút đã duyệt
Greedy BFS	74	79
A* Search	68	222



Hình 3.3. Hình ảnh mô phỏng thuật toán Greedy BFS trong dữ liệu medium_maze.txt

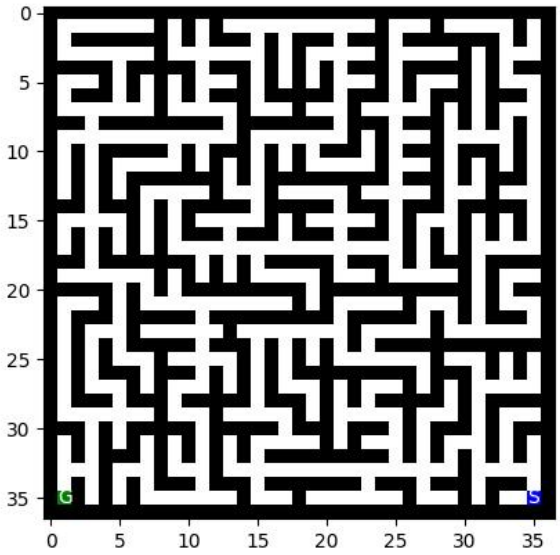


Hình 3.4. Hình ảnh mô phỏng thuật toán A Search trong dữ liệu medium_maze.txt*

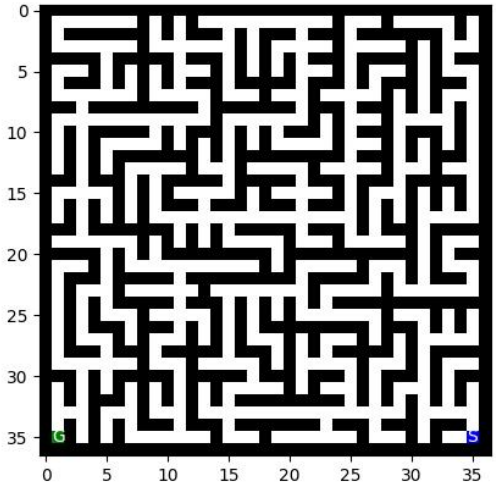
3.3. Large Maze (large_maze.txt)

- Mê cung lớn và phức tạp, đòi hỏi khả năng tìm kiếm hiệu quả.

Thuật toán	Độ dài đường đi	Số nút đã duyệt
Greedy BFS	210	467
A* Search	210	550



Hình 3.5. Hình ảnh mô phỏng thuật toán Greedy BFS trong dữ liệu large_maze.txt

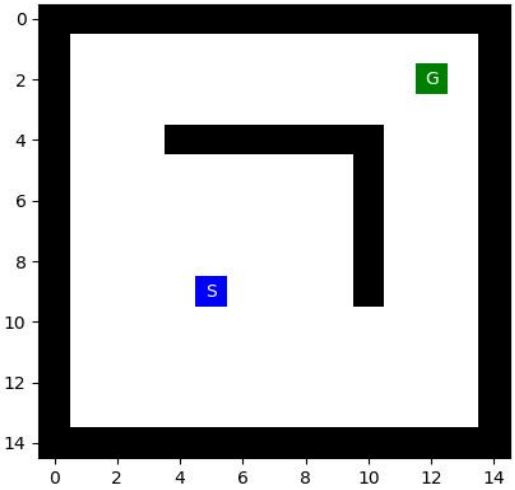


Hình 3.6. Hình ảnh mô phỏng thuật toán A Search trong dữ liệu large_maze.txt*

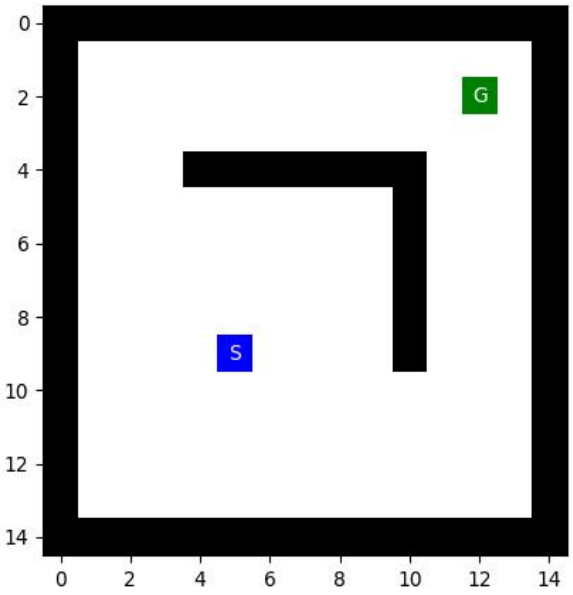
3.4. L Maze (L_maze.txt)

- Một mê cung được thiết kế đặc biệt với một chướng ngại vật lớn hình chữ L, là một "cái bẫy" cổ điển cho thuật toán GBFS.

Thuật toán	Độ dài đường đi	Số nút đã duyệt
Greedy BFS	24	35
A* Search	16	54



Hình 3.7. Hình ảnh mô phỏng thuật toán Greedy BFS trong dữ liệu l_maze.txt

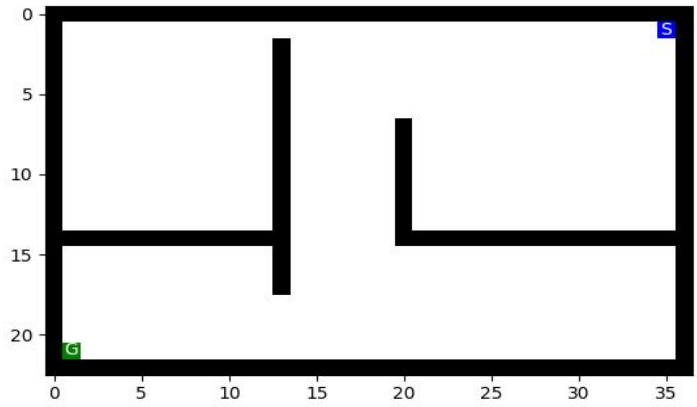


Hình 3.8. Hình ảnh mô phỏng thuật toán A Search trong dữ liệu l_maze.txt*

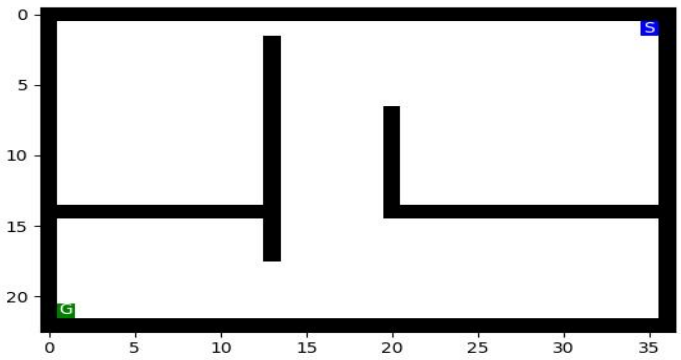
3.5. Open Maze (open_maze.txt)

- Một mê cung gần như trống, có rất ít chướng ngại vật.

Thuật toán	Độ dài đường đi	Số nút đã duyệt
Greedy BFS	68	90
A* Search	54	536



Hình 3.9. Hình ảnh mô phỏng thuật toán Greedy BFS trong dữ liệu open_maze.txt

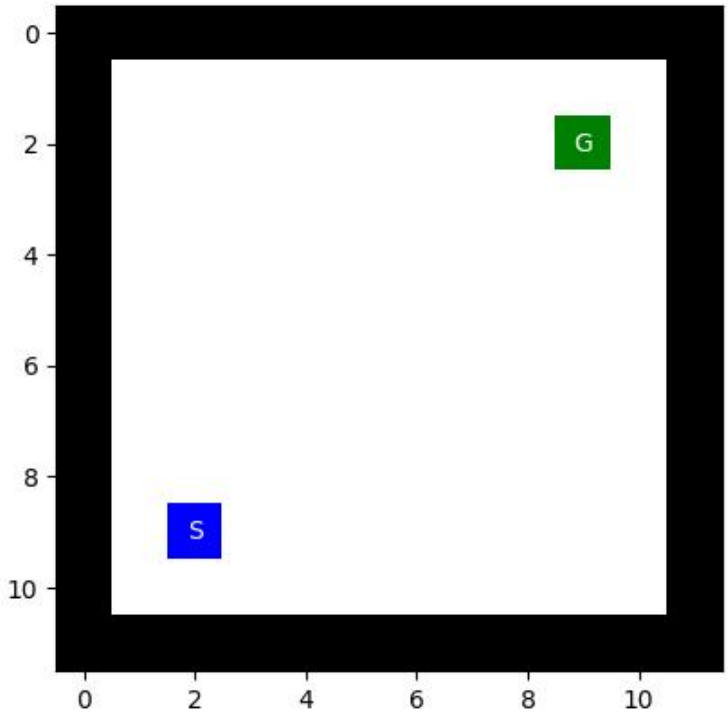


Hình 3.10. Hình ảnh mô phỏng thuật toán A Search trong dữ liệu open_maze.txt*

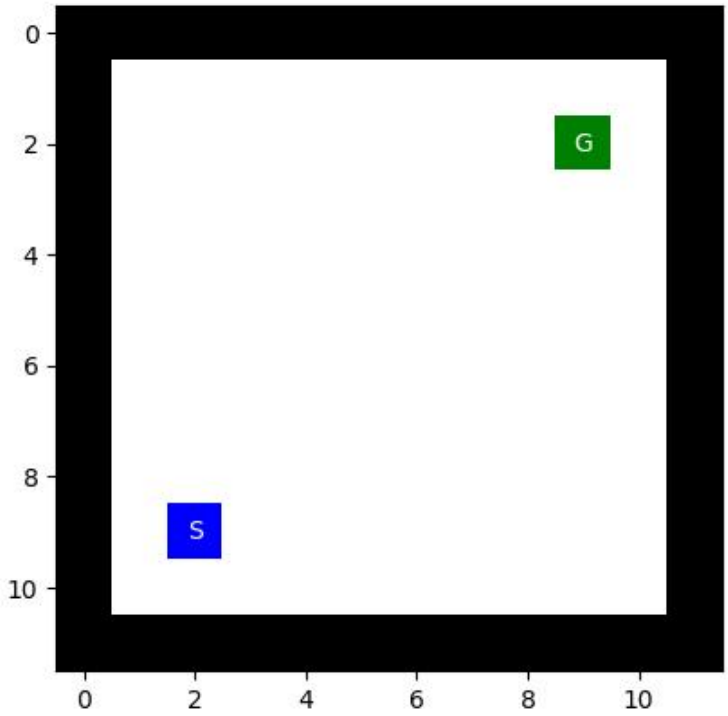
3.6. Empty Maze (empty_maze.txt)

- Một mê cung trống.

Thuật toán	Độ dài đường đi	Số nút đã duyệt
Greedy BFS	14	15
A* Search	14	64



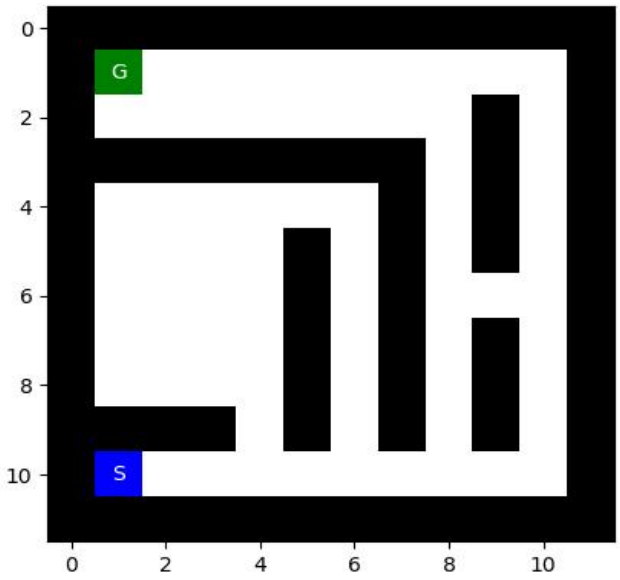
Hình 3.11. Hình ảnh mô phỏng thuật toán Greedy BFS trong dữ liệu empty_maze.txt



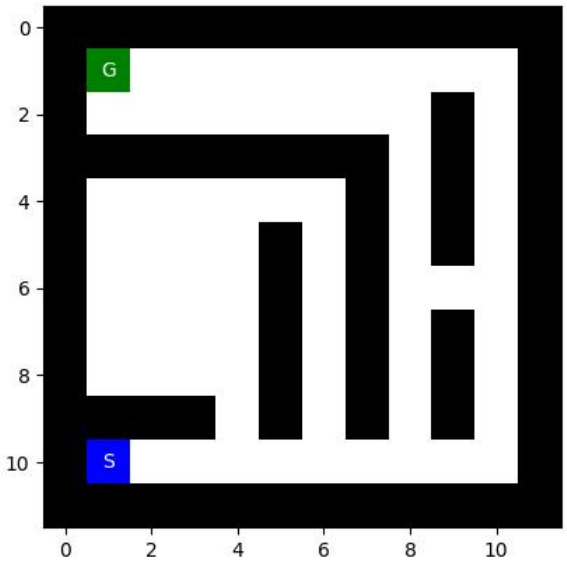
Hình 3.11. Hình ảnh mô phỏng thuật toán A Search trong dữ liệu empty_maze.txt*

3.7. Loops Maze (loops_maze.txt)

Thuật toán	Độ dài đường đi	Số nút đã duyệt
Greedy BFS	23	52
A* Search	23	59



Hình 3.13. Hình ảnh mô phỏng thuật toán Greedy BFS trong dữ liệu loops_maze.txt



Hình 3.14. Hình ảnh mô phỏng thuật toán A Search trong dữ liệu loops_maze.txt*

4. Thảo luận

Tính đầy đủ và tối ưu (Completeness and Optimality)

Greedy Best-First Search:

- Completeness (Tính đầy đủ): Không đầy đủ. Trong một số trường hợp, GBFS có thể đi vào một nhánh vô hạn (mặc dù trong cài đặt này, explored set đã ngăn chặn điều đó) hoặc không bao giờ tìm ra lối thoát khỏi một khu vực phức tạp mặc dù có lời giải.
- Optimality (Tính tối ưu): Không tối ưu. Như đã thấy trong các ví dụ medium_maze và L_maze, GBFS không quan tâm đến chi phí đã đi, vì vậy nó thường tìm thấy đường đi không phải là ngắn nhất.

A* Search:

- Completeness (Tính đầy đủ): Đầy đủ. Miễn là chi phí mỗi bước đi là dương, A* sẽ luôn tìm thấy lời giải nếu nó tồn tại.
- Optimality (Tính tối ưu): Tối ưu. A* đảm bảo tìm được đường đi ngắn nhất vì nó sử dụng một hàm heuristic hợp lệ (admissible). Khoảng cách Manhattan không bao giờ đánh giá quá cao chi phí thực tế để đến đích, do đó điều kiện tối ưu được thỏa mãn.
- Độ phức tạp thời gian và không gian (Time and Space Complexity)
- Gọi V là tổng số ô trong mê cung.

- Time Complexity: O(V log V)
- Trong trường hợp xấu nhất, cả hai thuật toán có thể phải đưa gần như tất cả các ô vào hàng đợi ưu tiên. Mỗi thao tác push hoặc pop trên hàng đợi có độ phức tạp là O(log V). Do đó, tổng độ phức tạp thời gian là O(V log V).
- Space Complexity: O(V)
- Trong trường hợp xấu nhất, cả frontier và explored set có thể phải lưu trữ tất cả các ô trong mê cung. Do đó, độ phức tạp không gian là tuyến tính theo số ô.

5. Kết luận

Qua quá trình cài đặt và thử nghiệm, ta có thể rút ra các kết luận sau:

- A* Search là một thuật toán mạnh mẽ, luôn đảm bảo tìm ra đường đi ngắn nhất nếu lời giải tồn tại, với điều kiện heuristic là hợp lệ. Tuy nhiên, nó có thể phải trả giá bằng việc khám phá nhiều nút hơn.
- Greedy Best-First Search thường nhanh hơn A* trong việc tìm ra một lời giải *bất kỳ*, đặc biệt trong các không gian mở. Tuy nhiên, nó không đáng tin cậy và thường không tìm ra đường đi tối ưu, dễ bị "đánh lừa" bởi các chướng ngại vật.
- Sự lựa chọn giữa A* và GBFS phụ thuộc vào yêu cầu của bài toán: nếu tính tối ưu là bắt buộc, A* là lựa chọn vượt trội; nếu chỉ cần tìm một đường đi nhanh nhất có thể mà không quan tâm đến độ dài, GBFS có thể là một phương án thay thế.

4. Nhiệm vụ 4:

So sánh đường đi (path cost)

- BFS và A* luôn tìm được đường đi ngắn nhất (Small: 20, Medium: 69, Large: 211).
- DFS thường ra đường đi dài hơn rất nhiều (Small: 50, Medium: 131, Large: 211), vì nó đi sâu ngẫu nhiên và dễ chọn nhánh "không tối ưu".
- GBS (Greedy Best-First Search) có kết quả dao động:
- Small: 30 (dài hơn BFS/A*).
- Medium: 153 (rất dài).

- Large: 211 (tối ưu được, nhưng do may mắn heuristic dẫn đúng).
  → Kết luận: GBS không đảm bảo tối ưu, trong khi BFS và A* thì đảm bảo (A* thường hiệu quả hơn BFS).

Số nút được mở rộng (nodes expanded)

- DFS mở rộng ít nút hơn BFS trong Small và Medium, nhưng không ổn định (Large DFS 427 > BFS 620).
- GBS thường mở ít nút hơn BFS và gần bằng DFS (Small: 40, Medium: 158, Large: 454).
- A* consistently mở rộng ít nút hơn BFS nhưng nhiều hơn DFS/GBS, đổi lại tìm được đường đi tối ưu.
  → BFS mở rộng cực nhiều vì duyệt "ngang" theo lớp, A* cân bằng giữa heuristic và cost nên ít tốn kém hơn.

Độ sâu tối đa (max depth)

- DFS luôn có độ sâu rất lớn (Small: 49, Medium: 130, Large: 222) vì nó đi theo nhánh sâu nhất trước.
- BFS và A* thì độ sâu tương ứng với độ dài đường đi ngắn nhất (Small: 19, Medium: 68, Large: 210).
- GBS có độ sâu thay đổi (Small: 29, Medium: 152, Large: 210), cho thấy nó có thể "lạc đường" sâu không cần thiết.

Bộ nhớ tiêu thụ (max memory, max frontier)

- BFS tiêu tốn bộ nhớ lớn nhất (Small: 188, Medium: 544, Large: 1246) vì nó lưu trữ cả một lớp frontier. Frontier size cũng rất cao (Large: 623).
- DFS và GBS dùng bộ nhớ ít nhất (Large DFS: 498, GBS: 479).
- A* dùng bộ nhớ vừa phải (Large: 565), cao hơn DFS/GBS nhưng vẫn ít hơn BFS.

Tổng kết và bài học

- BFS: đảm bảo đường đi ngắn nhất, nhưng cực tốn bộ nhớ và mở rộng nhiều nút → chỉ phù hợp với bài toán nhỏ.
- DFS: dùng ít bộ nhớ, mở rộng nhanh, nhưng đường đi thường dài, thậm chí không tối ưu → chỉ thích hợp khi muốn kiểm tra có lời giải hay không.

- GBS: chạy nhanh, tiết kiệm tài nguyên nhưng dễ bị "lạc đường" vì heuristic không tính cost, nên kết quả không ổn định.
- A*: cân bằng nhất, luôn tìm được đường đi tối ưu và mở rộng ít nút hơn BFS, bộ nhớ cũng hợp lý hơn. Đây là thuật toán tốt nhất trong 4 cái cho bài toán tìm đường mê cung.

Tóm lại

- BFS = đúng, nhưng chậm & tốn RAM.
- DFS = nhanh & ít RAM, nhưng sai/lệch đường đi.
- GBS = tiết kiệm nhưng không đáng tin.
- A* = tối ưu nhất, trade-off hợp lý.

Qua việc cài đặt và so sánh các chiến lược tìm kiếm khác nhau (DFS, BFS, UCS), tôi rút ra nhiều bài học quan trọng. Trước hết, DFS có ưu điểm là dễ cài đặt, tốn ít bộ nhớ, nhưng thường đi quá sâu vào một nhánh, dẫn đến đường đi tìm được không tối ưu, thậm chí có thể rơi vào ngõ cụt. Điều này đặt ra câu hỏi: *Liệu có cách nào luôn tìm được đường đi ngắn nhất không?* Kết quả là BFS giải quyết được vấn đề này, bởi BFS duyệt theo từng lớp nên luôn tìm được đường đi ngắn nhất (tính theo số bước). Tuy nhiên, nhược điểm của BFS là phải lưu trữ rất nhiều trạng thái trung gian, gây tốn kém bộ nhớ.

Khi thử với UCS, tôi học được rằng việc đưa chi phí (cost) vào trong tìm kiếm giúp thuật toán trở nên linh hoạt hơn. UCS vẫn đảm bảo tìm được đường đi tối ưu, nhưng thay vì chỉ tính theo số bước như BFS, nó có thể áp dụng cho nhiều bài toán phức tạp hơn, nơi mỗi bước đi có chi phí khác nhau. Trong trường hợp chi phí đồng đều, UCS hoạt động gần giống BFS.

Từ những quan sát này, có thể thấy mỗi thuật toán có ưu và nhược điểm riêng: DFS nhanh và gọn nhưng thiếu tin cậy, BFS đảm bảo tối ưu nhưng nặng về bộ nhớ, còn UCS cân bằng được tính đúng đắn và khả năng mở rộng cho các bài toán có trọng số. Đây là kinh nghiệm quan trọng khi lựa chọn chiến lược tìm kiếm phù hợp cho từng loại bài toán.

5. Nhiệm vụ nâng cao:

1. Nhiệm vụ nâng cao: IDS và đa mục tiêu
    a. IDS:

- Triển khai IDS (tìm kiếm sâu dần lặp lại) bằng cách sử dụng triển khai DFS. Thử nghiệm IDS trên các mê cung ở trên. Có thể gặp một số vấn đề với các mê cung có không gian mở. Nếu không thể giải quyết các vấn đề, hãy báo cáo và thảo luận về nguyên nhân gây ra các vấn đề đó.
- Tổng quan: Nhiệm vụ này yêu cầu triển khai thuật toán Tìm kiếm Sâu dần Lặp lại (Iterative Deepening Search - IDS), một chiến lược tìm kiếm không thông tin trước kết hợp ưu điểm của cả BFS và DFS.
  + Ưu điểm:
    - Hoàn chỉnh: Giống như BFS, IDS luôn đảm bảo tìm thấy lời giải nếu có tồn tại.
    - Tối ưu: Giống như BFS, nó tìm thấy đường đi có độ dài ngắn nhất (tối ưu về chi phí).
    - Hiệu quả về Bộ nhớ: Giống như DFS, nó chỉ lưu trữ một nhánh duy nhất của cây tìm kiếm tại một thời điểm, giúp tiết kiệm bộ nhớ đáng kể so với BFS.
  + Nhược điểm :
    - Nó có thể lãng phí thời gian vì các nút ở các tầng trên của cây tìm kiếm sẽ được mở rộng nhiều lần. Tuy nhiên, trong thực tế, nhược điểm này không quá nghiêm trọng, vì số lượng nút tăng theo cấp số nhân với độ sâu, do đó phần lớn công việc được thực hiện ở lần lặp cuối cùng.
- Cấu trúc triển khai: Thuật toán IDS được xây dựng dựa trên hai hàm chính:
  + dfs_limited_search(maze, start_pos, goal_pos, limit): Đây là trái tim của thuật toán. Nó thực hiện một tìm kiếm DFS thông thường, nhưng với một giới hạn độ sâu (limit).
    - Nó sử dụng một ngăn xếp (collections.deque) để lưu trữ các nút cần khám phá, cùng với độ sâu của chúng.
    - Hàm này sẽ dừng khám phá một nhánh ngay khi độ sâu của nó vượt quá giới hạn đã cho.

- Kiểm tra chu trình: Để tránh lặp vô hạn trong một lần lặp DFS cụ thể, một tập hợp (explored_in_iteration) được sử dụng để theo dõi các trạng thái đã được thăm trong lần tìm kiếm đó. Tập hợp này được khởi tạo lại cho mỗi lần lặp DFS mới.

+ ids_search(maze): Đây là hàm điều khiển chính. Nó thực hiện một vòng lặp vô hạn, trong đó nó liên tục gọi dfs_limited_search với giới hạn độ sâu tăng dần (từ 0, 1, 2, ...).
  - Vòng lặp sẽ tiếp tục cho đến khi dfs_limited_search trả về một đường đi (tức là tìm thấy lời giải).
  - Khi lời giải được tìm thấy, IDS sẽ dừng và trả về đường đi.

- Các vấn đề trong môi trường không gian mở:
  + Hiệu suất Thời gian không tối ưu: Mặc dù lý thuyết chỉ ra rằng IDS hiệu quả về thời gian, việc triển khai này có thể gặp khó khăn với các mê cung lớn và ít chướng ngại vật. Do số lượng nút tăng theo cấp số nhân, việc lặp lại tìm kiếm từ đầu ở mỗi lần tăng giới hạn có thể dẫn đến việc thăm lại một lượng lớn các nút đã được khám phá ở các lần lặp trước, làm tăng đáng kể thời gian tìm kiếm.
  + Vấn đề với việc kiểm tra chu trình: Việc chỉ kiểm tra chu trình trong một lần lặp DFS duy nhất (explored_in_iteration) có thể không đủ. Trong một không gian mở, một đường đi có thể đi xa và sau đó quay trở lại một nút đã được thăm ở một lần lặp DFS trước đó, nhưng không nằm trong đường đi hiện tại. Mặc dù IDS vẫn sẽ tìm thấy lời giải, nhưng nó có thể làm việc không hiệu quả.

b. Đa mục tiêu

Tạo một vài mê cung có nhiều mục tiêu bằng cách thêm một hoặc hai mục tiêu nữa vào mê cung cỡ trung bình. Tác nhân sẽ hoàn thành nhiệm vụ khi tìm thấy một trong các mục tiêu.

Giải mê cung bằng các triển khai DFS, BFS và IDS. Chạy các thử nghiệm để cho thấy triển khai nào tìm thấy giải pháp tối ưu và triển khai nào không. Thảo luận lý do tại sao lại như vậy.

- Bài toán: Nhiệm vụ này mở rộng bài toán tìm đường đi trong mê cung sang một kịch bản thực tế hơn: tác nhân phải tìm thấy **một trong nhiều mục tiêu khả thi**. Tác nhân hoàn thành nhiệm vụ ngay khi nó đến được bất kỳ ô nào được đánh dấu là 'G'.Để thực hiện điều này, các thuật toán tìm kiếm ban đầu (BFS, DFS, IDS) đã được sửa đổi. Thay vì tìm một mục tiêu duy nhất, chúng sẽ kiểm tra xem trạng thái hiện tại có nằm trong một danh sách các mục tiêu hay không.
- Triển khai thuật toán đã sửa đổi:
  + **Hàm find_start_and_goals(maze):** Thay thế hàm cũ, hàm này giờ đây không chỉ tìm vị trí bắt đầu ('S') mà còn thu thập tất cả các vị trí mục tiêu ('G') vào một danh sách.
  + **Sửa đổi điều kiện dừng:** Trong mỗi thuật toán, điều kiện if current_state == goal_pos: được thay thế bằng if current_state in goal_pos_list:. Điều này cho phép thuật toán dừng ngay khi tìm thấy bất kỳ mục tiêu nào.
- Phân tích hiệu suất và tính tối ưu:

Thuật toán	Tính tối ưu	Lý do
BFS	Có	BFS là thuật toán tìm kiếm hoàn chỉnh và tối ưu (đối với chi phí bước bằng nhau). Nó mở rộng các nút theo thứ tự từng lớp, đảm bảo rằng nó sẽ tìm thấy mục tiêu gần nhất đầu tiên, tính theo số bước. Điều này làm cho nó trở thành lựa chọn lý tưởng cho các bài toán đa mục tiêu khi mục tiêu là tìm đường đi ngắn nhất đến bất kỳ đích nào.
DFS	Không	DFS đi sâu vào một nhánh cho đến khi nó tìm thấy một mục tiêu hoặc đạt đến điểm cụt. Trong một mê cung có nhiều mục tiêu, DFS có thể vô tình tìm thấy một mục tiêu nằm trên một nhánh

		rất dài và quanh co trước khi khám phá một mục tiêu khác gần hơn nhiều nhưng nằm trên một nhánh khác. Do đó, DFS không đảm bảo tìm thấy đường đi ngắn nhất.
IDS	Có	IDS về cơ bản là một chuỗi các tìm kiếm DFS giới hạn độ sâu. Nó lặp lại, tăng giới hạn độ sâu sau mỗi lần lặp, bắt đầu từ độ sâu 0. Bằng cách này, IDS khám phá các nút theo thứ tự giống hệt như BFS, đảm bảo rằng nó sẽ tìm thấy mục tiêu đầu tiên (và do đó là mục tiêu gần nhất) ngay khi nó đạt đến độ sâu của mục tiêu đó. Tuy nhiên, nó đạt được tính tối ưu này với hiệu quả bộ nhớ cao hơn nhiều so với BFS.

6. Nhiệm vụ nâng cao:

a. Intersection as State.

Giải pháp

- Để BFS và IDS (phiên bản điều chỉnh) vẫn tối ưu trên đồ thị có trọng số, chúng ta phải thay đổi cách chúng hoạt động để ưu tiên tổng chi phí đường đi (g(n)) thay vì số bước đi.
- Thay thế BFS bằng Uniform Cost Search (UCS):
- Uniform Cost Search (UCS), hay còn được biết đến là thuật toán Dijkstra, là một biến thể của BFS được thiết kế cho đồ thị có trọng số.
- Thay vì sử dụng hàng đợi (Queue) thông thường, UCS sử dụng hàng đợi ưu tiên (Priority Queue).
- Các nút trong hàng đợi được sắp xếp theo tổng chi phí từ điểm bắt đầu (g(n)). Nút có g(n) thấp nhất sẽ luôn được ưu tiên xử lý.
- Bằng cách này, UCS đảm bảo rằng khi nó tìm thấy đích, đó chắc chắn là con đường có tổng chi phí thấp nhất.
- Thực chất, UCS chính là thuật toán A* mà không có heuristic (h(n) = 0).
- Điều chỉnh IDS:

+ Phiên bản IDS tiêu chuẩn duyệt theo depth_limit (giới hạn số cạnh).

+ Để nó tối ưu trên đồ thị có trọng số, chúng ta phải thay đổi nó thành duyệt theo cost_limit (giới hạn tổng chi phí).

+ Thuật toán sẽ chạy một loạt các lượt tìm kiếm theo chiều sâu (Depth-First Search) với giới hạn chi phí tăng dần.

+ Lần 1: Tìm đường đi có tổng chi phí <= C.

+ Lần 2: Nếu không tìm thấy, tìm đường đi có tổng chi phí <= C + delta. và cứ thế tiếp tục.

+ Thuật toán này phức tạp hơn để cài đặt và thường kém hiệu quả hơn UCS vì nó phải duyệt lại các trạng thái nhiều lần. Vì vậy, trong phần code bên dưới, chúng ta sẽ tập trung vào việc cài đặt UCS, là giải pháp chuẩn và hiệu quả hơn.

Kết quả

Kiểm thử trên dữ liệu mê cung trong file l_maze.txt

Thuật toán	Độ dài đường đi	Số nút đã duyệt
Uniform Cost Search	16	148
A* Search	16	54

b. Weighted A* search và Unknown Maze

Qua việc triển khai Weighted A* và Unknown Maze Agent, tôi rút ra nhiều bài học quan trọng. Weighted A* giúp thấy rõ cách tăng trọng số heuristic ảnh hưởng đến hiệu quả tìm kiếm: trọng số lớn hơn ưu tiên việc đi thẳng về phía goal, làm giảm số node mở rộng nhưng đôi khi không tối ưu tuyệt đối. Kết quả cho thấy Weighted A* cân bằng tốt giữa độ dài đường đi và bộ nhớ sử dụng, thể hiện sức mạnh của heuristic trong các thuật toán informed search. Ngược lại, Unknown Maze Agent minh họa thách thức khi môi trường chưa biết trước: agent phải khám phá từng bước, vừa phát hiện maze vừa định hướng bằng GPS. Điều này cho thấy tầm quan trọng của exploration và exploitation, cũng như việc quản lý frontier trong môi trường online. Tôi học được cách theo dõi số node mở rộng, độ sâu tối đa và kích thước frontier, để đánh giá hiệu quả thuật toán. So sánh hai phương pháp giúp nhận ra rằng thông tin đầy đủ (map) giúp thuật toán planning hiệu quả hơn, trong khi không biết trước môi trường làm tăng chi phí tìm kiếm.

Weighted A* nhấn mạnh tầm quan trọng của heuristic design, còn Unknown Maze Agent dạy về adaptive search và online planning. Hai đoạn code này cũng giúp củng cố kiến thức về cấu trúc node, parent, path reconstruction, và cách đo lường hiệu suất thuật toán. Tổng thể, việc thử nghiệm cả môi trường offline và online giúp hiểu sâu hơn về sự trade-off giữa thời gian, bộ nhớ và tính tối ưu trong các thuật toán tìm kiếm.

INTELLIGENT AGENTS: REFLEX-BASED AGENTS FOR THE VACUUM-CLEANER WORLD (TÁC NHÂN THÔNG MINH: TÁC NHÂN DỰA TRÊN PHẢN XẠ CHO THẾ GIỚI MÁY HÚT BỤI)

I. MỤC TIÊU:

- Thiết kế và xây dựng một môi trường mô phỏng mô phỏng các đầu vào cảm biến, hiệu ứng của các cơ cấu chấp hành, và đo lường hiệu suất.
- Áp dụng các khái niệm cơ bản về AI bằng cách triển khai hàm tác nhân cho các tác nhân phản xạ đơn giản và dựa trên mô hình, phản ứng với các nhận thức từ môi trường.
- Thực hành cách môi trường và hàm tác nhân tương tác với nhau.
- Phân tích hiệu suất của tác nhân thông qua các thí nghiệm kiểm soát trên các cấu hình môi trường khác nhau.
- Phát triển các chiến lược xử lý sự không chắc chắn và thông tin không hoàn chỉnh trong các hệ thống tác nhân tự động.

II. GIỚI THIỆU

Triển khai một môi trường mô phỏng cho robot hút bụi tự động, một tập hợp các chương trình tác nhân dựa trên phản xạ khác nhau, và thực hiện một nghiên cứu so sánh cho việc dọn dẹp một phòng duy nhất. Tập trung vào **giai đoạn dọn dẹp** bắt đầu khi robot được kích hoạt và kết thúc khi tất cả các ô bẩn trong phòng đã được làm sạch. Việc điều khiển robot quay trở lại trạm sạc sau khi phòng đã sạch sẽ do người khác đảm nhiệm.

III. MÔ TẢ PEAS CHO GIAI ĐOẠN DỌN DẸP

- Chỉ số hiệu suất (Performance Measure): Mỗi hành động tiêu tốn 1 đơn vị năng lượng. Hiệu suất được đo bằng tổng số đơn vị năng lượng đã sử dụng để làm sạch toàn bộ phòng.
- Môi trường (Environment): Một phòng gồm n x n ô vuông, với n = 5. Bụi bẩn được đặt ngẫu nhiên trên mỗi ô với xác suất p = 0.2. Để đơn giản, bạn có

thể giả sử tác nhân biết kích thước và bố cục phòng (tức là biết n). Ban đầu, tác nhân được đặt trên một ô ngẫu nhiên.

- Cơ cấu chấp hành (Actuators): Tác nhân có thể làm sạch ô hiện tại (hành động suck) hoặc di chuyển đến một ô liền kề bằng cách đi north (bắc), east (đông), south (nam), hoặc west (tây).
- Cảm biến (Sensors): Bốn cảm biến va chạm, mỗi cảm biến cho một hướng: bắc, đông, nam, tây; một cảm biến bụi báo cáo có bụi trên ô hiện tại hay không.

IV. CHƯƠNG TRÌNH TÁC NHÂN CHO TÁC NHÂN NGẪU NHIÊN ĐƠN GIẢN

Chương trình tác nhân là một hàm nhận thông tin cảm biến (percepts hiện tại) làm đối số. Các đối số là:

- Một dictionary với các giá trị boolean cho bốn cảm biến va chạm north, east, west, south. Ví dụ, nếu tác nhân ở góc bắc-tây, bumpers sẽ là {"north" : True, "east" : False, "south" : False, "west" : True}.
- Cảm biến bụi trả về giá trị boolean.

Tác nhân trả về hành động được chọn dưới dạng một chuỗi (string).

V. NHIỆM VỤ:

1. Nhiệm vụ 1:

1. Mục tiêu

Mục tiêu của Task 1 là xây dựng một môi trường mô phỏng (simulation environment) cho robot hút bụi tự động trong bài toán Vacuum Cleaner World, tuân theo mô tả PEAS đã cho. Môi trường cần:

- Lưu trạng thái sạch/bẩn của từng ô trong phòng.
- Cung cấp thông tin cảm biến (bumpers, dirt sensor) cho agent.
- Cập nhật trạng thái môi trường dựa trên hành động của agent.
- Đo lường hiệu suất dựa trên số bước hành động cần để dọn sạch toàn bộ phòng.

2. Thiết kế mô hình môi trường

Theo mô tả, ta có các thành phần chính:

- Performance Measure: Hiệu suất đo bằng tổng số hành động (mỗi hành động tốn 1 đơn vị năng lượng) cho đến khi tất cả ô sạch.
- Environment:
    - Phòng có kích thước n × n (với n = 5).
    - Mỗi ô có xác suất p = 0.2 ban đầu bị bẩn.
    - Vị trí ban đầu của agent được chọn ngẫu nhiên.
- Actuators:
    - north, south, east, west: di chuyển lên/xuống/trái/phải.
    - suck: làm sạch ô hiện tại.
- Sensors:
    - Bumpers: 4 cảm biến va chạm tường (True nếu agent ở biên).
    - Dirt sensor: True nếu ô hiện tại bẩn.

3. Cấu trúc thuật toán môi trường

1. Khởi tạo môi trường
    - Sinh ma trận n × n với giá trị True (dirty) hoặc False (clean).
    - Đặt agent ở vị trí ngẫu nhiên.
2. Vòng lặp mô phỏng
    - Sinh cảm biến (bumpers + dirt sensor).
    - Gọi hàm agent để nhận action.
    - Cập nhật trạng thái môi trường:
        - Nếu "suck" → làm sạch ô hiện tại.
        - Nếu "move" → di chuyển nếu không có tường.
    - Tăng bộ đếm bước.
3. Điều kiện dừng
    - Tất cả các ô sạch.
    - Hoặc vượt quá số bước tối đa (max_steps).

Kết quả quan sát:

- Agent có thể dọn sạch phòng sau một số bước nhất định.
- Do agent chọn hành động ngẫu nhiên nên thường không tối ưu, có thể mất nhiều bước hoặc không dọn sạch hết trong giới hạn max_steps.

2. Nhiệm vụ 2: Triển khai tác nhân phản xạ đơn giản

Tác nhân phản xạ đơn giản là một trong những loại tác nhân trí tuệ nhân tạo cơ bản nhất. Nó hoạt động dựa trên các quy tắc điều kiện-hành động (condition-action rules) và không có bất kỳ bộ nhớ nào về môi trường hay lịch sử hoạt động. Điều này có nghĩa là quyết định của tác nhân tại một thời điểm bất kỳ chỉ phụ thuộc vào tri giác hiện tại của nó.

a. Tác nhân phản xạ đơn giản

Tác nhân phản xạ đơn giản di chuyển ngẫu nhiên nhưng phản ứng với cảm biến cản bằng cách không đâm vào tường và phản ứng với bụi bẩn bằng cách hút. Triển khai chương trình tác nhân này dưới dạng một hàm.

Lưu ý: Tác nhân không thể trực tiếp sử dụng các biến trong môi trường. Nó chỉ nhận các tri giác làm tham số cho hàm tác nhân. Sử dụng cùng một cấu trúc hàm như hàm simple_randomized_agent ở trên.

Logic của tác nhân theo ba bước chính:

- Kiểm tra bụi bẩn: Quy tắc đầu tiên và quan trọng nhất là "Nếu cảm biến bụi bẩn báo bẩn, hãy hút". Điều này được thực hiện bằng cách kiểm tra percept['is_dirty']. Nếu True, tác nhân sẽ trả về hành động 'Suck'. Đây là một quy tắc ưu tiên cao, vì mục tiêu chính của tác nhân là làm sạch.
- Xử lý chướng ngại vật: Nếu không có bụi bẩn, tác nhân sẽ kiểm tra các hướng di chuyển có thể. percept['can_move'] cung cấp thông tin về các chướng ngại vật (ví dụ: tường). Bằng cách duyệt qua dictionary này, tác nhân xác định được các hướng an toàn để di chuyển, tránh đâm vào tường.
- Di chuyển ngẫu nhiên: Từ danh sách các hướng hợp lệ (valid_moves), tác nhân sẽ chọn một hướng ngẫu nhiên để di chuyển. Việc lựa chọn ngẫu nhiên này thể hiện sự thiếu mục tiêu hoặc chiến lược lâu dài của tác nhân, một đặc điểm cốt lõi của tác nhân phản xạ đơn giản. Nếu không có hướng di chuyển nào hợp lệ (ví dụ: tác nhân bị kẹt trong một góc), nó sẽ thực hiện hành động 'NoOp' (không làm gì cả).

Mặc dù đơn giản và dễ triển khai, tác nhân phản xạ này có những hạn chế đáng kể, đặc biệt khi phải đối mặt với một căn phòng lớn hoặc phức tạp:

- Hiệu suất không tối ưu: Do di chuyển ngẫu nhiên, tác nhân có thể đi lòng vòng ở một khu vực, bỏ sót các khu vực khác, hoặc lặp lại các đường đi

không hiệu quả. Nó không có cách nào để biết liệu nó đã làm sạch toàn bộ căn phòng hay chưa.

- Không xử lý được các lỗi cảm biến: Nếu cảm biến bụi bẩn thỉnh thoảng báo sai (ví dụ: một ô bẩn nhưng cảm biến báo sạch), tác nhân sẽ bỏ qua ô đó vĩnh viễn vì nó không có bộ nhớ để quay lại.
- Không có mục tiêu dài hạn: Tác nhân không có khái niệm về mục tiêu cuối cùng là làm sạch toàn bộ căn phòng. Nó chỉ phản ứng với từng tình huống tức thời mà không có kế hoạch.

b. Môi trường mô phỏng

Môi trường mô phỏng (simple_environment) đóng vai trò là "thế giới" nơi tác nhân hoạt động. Nó chịu trách nhiệm quản lý trạng thái của môi trường, cung cấp tri giác cho tác nhân, và thực thi các hành động mà tác nhân lựa chọn.

- Thiết kế vai trò: Môi trường này được thiết kế theo mô hình PEAS (Performance, Environment, Actuators, Sensors) để đảm bảo tính nhất quán với lý thuyết tác nhân AI.
    + P (Performance - Hiệu suất): Được đo bằng số hành động mà tác nhân đã thực hiện để làm sạch toàn bộ căn phòng.
    + E (Environment - Môi trường): Một lưới 2D (numpy array) với kích thước tùy chỉnh, nơi mỗi ô có thể ở trạng thái sạch (0) hoặc bẩn (1). Nó cũng theo dõi vị trí của robot.
    + A (Actuators - Bộ truyền động): Môi trường thực hiện các hành động mà tác nhân đưa ra, bao gồm Suck (hút bụi) và di chuyển theo các hướng up, down, left, right.
    + S (Sensors - Cảm biến): Môi trường cung cấp tri giác cho tác nhân tại mỗi bước. Tri giác bao gồm hai thông tin chính: trạng thái bụi bẩn tại ô hiện tại (is_dirty) và khả năng di chuyển theo từng hướng (can_move), mô phỏng cảm biến va chạm.
- Logic hoạt động: Mã của môi trường hoạt động theo một vòng lặp chính, đảm bảo sự tương tác liên tục giữa tác nhân và thế giới ảo.
    + Khởi tạo Môi trường: Khi hàm simple_environment được gọi, nó tạo ra một căn phòng mới với tỷ lệ bụi bẩn ban đầu được xác định ngẫu

nhiên. Vị trí ban đầu của robot cũng được thiết lập. Điều này tạo ra một kịch bản ban đầu độc đáo cho mỗi lần mô phỏng.

+ Vòng lặp Mô phỏng: Một vòng lặp while là trung tâm của mô phỏng. Nó tiếp tục chạy cho đến khi tất cả các ô bẩn được làm sạch hoặc số bước tối đa được đạt đến.
+ Ghi nhận Hiệu suất: Sau mỗi hành động, biến action_count tăng lên, cung cấp một thước đo định lượng về chi phí hoạt động của tác nhân.
+ Trực quan hóa và Gỡ lỗi: Tham số verbose=True cho phép hiển thị từng bước của mô phỏng, bao gồm cả trạng thái phòng được trực quan hóa bằng hàm display_room. Chức năng này rất quan trọng để xác minh rằng cả môi trường và tác nhân đều hoạt động đúng như mong đợi.

- Vai trò trong hệ thống: Môi trường đóng vai trò là một "công cụ thử nghiệm" để đánh giá các tác nhân khác nhau. Bằng cách giữ cho môi trường ổn định và tách biệt với logic của tác nhân, chúng ta có thể so sánh hiệu suất của các chiến lược AI khác nhau (như tác nhân phản xạ, tác nhân dựa trên mô hình,...) một cách công bằng và có hệ thống.

3. Nhiệm vụ 3:

1. Mục tiêu

Mục tiêu của Task 3 là thiết kế và triển khai một tác nhân thông minh hơn, có khả năng làm sạch toàn bộ căn phòng một cách có hệ thống. Không giống như tác nhân ngẫu nhiên hoặc tác nhân phản xạ đơn giản, tác nhân này sử dụng một mô hình nội bộ (internal model) để lưu trữ trạng thái, giúp nó hiểu được vị trí của mình và lập kế hoạch di chuyển một cách hiệu quả.

2. Thiết kế Tác nhân

Tác nhân được thiết kế để hoạt động theo một logic hai giai đoạn rõ ràng, dựa trên trạng thái nội bộ được duy trì giữa các lần gọi hàm:

Trạng thái (State): Trạng thái của tác nhân được lưu trong một dictionary, bao gồm:

'localized': Cho biết tác nhân đã xác định được vị trí của mình hay chưa.

'location': Tọa độ (x, y) hiện tại của tác nhân.

'direction': Hướng di chuyển chính hiện tại ('east' hoặc 'west').

'room_size': Kích thước n của phòng.

Giai đoạn 1: Định vị (Localization)

Nếu chưa định vị ('localized' là False), tác nhân sẽ di chuyển một cách có chủ đích về góc Tây-Bắc (trên-trái) của căn phòng.

Nó di chuyển liên tục về phía west cho đến khi chạm tường, sau đó di chuyển liên tục về phía north cho đến khi chạm tường.

Khi đã ở góc, nó cập nhật trạng thái 'localized' thành True và đặt vị trí của mình là (0, 0).

Giai đoạn 2: Dọn dẹp có hệ thống (Systematic Cleaning)

Sau khi định vị, tác nhân ưu tiên hành động 'suck' nếu ô hiện tại bẩn.

Nếu ô hiện tại sạch, nó sẽ di chuyển theo một mẫu "cày ruộng" (boustrophedon):

Di chuyển sang 'east' cho đến hết hàng.

Di chuyển xuống 'south' một ô.

Di chuyển sang 'west' cho đến hết hàng.

Di chuyển xuống 'south' một ô, và lặp lại.

Sau mỗi lần di chuyển, tác nhân cập nhật tọa độ 'location' trong trạng thái của mình.

Chiến lược này đảm bảo rằng mọi ô trong phòng sẽ được đi qua ít nhất một lần một cách có trật tự.

3. Kết quả Chạy thử

Dưới đây là kết quả của một lần chạy thử nghiệm với môi trường phòng 5x5, p=0.3 và seed=42 để đảm bảo tính nhất quán.

```
Step 1
[['.' 'D' 'D' '.' '.']
['.' '.' 'D' '.' 'D']
['.' 'D' 'D' 'R' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
```

```
----------
Agent position (row, col): (2, 3), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}
Action: west
Step 2
[['.' 'D' 'D' '.' '.']
['.' '.' 'D' '.' 'D']
['.' 'D' 'R' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
----------
Agent position (row, col): (2, 2), Dirty: True, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}
Action: west
Step 3
[['.' 'D' 'D' '.' '.']
['.' '.' 'D' '.' 'D']
['.' 'R' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
----------
Agent position (row, col): (2, 1), Dirty: True, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}
Action: west
Step 4
[['.' 'D' 'D' '.' '.']
['.' '.' 'D' '.' 'D']
['R' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
```

```
----------
Agent position (row, col): (2, 0), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': True, 'east': False}
Action: north
Step 5
[['.' 'D' 'D' '.' '.']
['R' '.' 'D' '.' 'D']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
----------
Agent position (row, col): (1, 0), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': True, 'east': False}
Action: north
Step 6
[['R' 'D' 'D' '.' '.']
['.' '.' 'D' '.' 'D']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
----------
Agent position (row, col): (0, 0), Dirty: False, Bumpers: {'north': True, 'south': False, 'west': True, 'east': False}
Action: suck
Step 7
[['R' 'D' 'D' '.' '.']
['.' '.' 'D' '.' 'D']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
```

```
----------
Agent position (row, col): (0, 0), Dirty: False, Bumpers: {'north': True, 'south': False, 'west': True, 'east': False}
Action: east
Step 8
[['.' 'R' 'D' '.' '.']
['.' '.' 'D' '.' 'D']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
----------
Agent position (row, col): (0, 1), Dirty: True, Bumpers: {'north': True, 'south': False, 'west': False, 'east': False}
Action: suck
Step 9
[['.' 'R' 'D' '.' '.']
['.' '.' 'D' '.' 'D']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
----------
Agent position (row, col): (0, 1), Dirty: False, Bumpers: {'north': True, 'south': False, 'west': False, 'east': False}
Action: east
Step 10
[['.' '.' 'R' '.' '.']
['.' '.' 'D' '.' 'D']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
```

Agent position (row, col): (0, 2), Dirty: True, Bumpers: {'north': True, 'south': False, 'west': False, 'east': False}

Action: suck

Step 11

[['.' '.' 'R' '.' '.']

['.' '.' 'D' '.' 'D']

['.' 'D' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (0, 2), Dirty: False, Bumpers: {'north': True, 'south': False, 'west': False, 'east': False}

Action: east

Step 12

[['.' '.' '.' 'R' '.']

['.' '.' 'D' '.' 'D']

['.' 'D' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (0, 3), Dirty: False, Bumpers: {'north': True, 'south': False, 'west': False, 'east': False}

Action: east

Step 13

[['.' '.' '.' '.' 'R']

['.' '.' 'D' '.' 'D']

['.' 'D' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

```
----------
Agent position (row, col): (0, 4), Dirty: False, Bumpers: {'north': True, 'south': False, 'west': False, 'east': True}
Action: south
Step 14
[['.' '.' '.' '.' '.']
['.' '.' 'D' '.' 'R']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
----------
Agent position (row, col): (1, 4), Dirty: True, Bumpers: {'north': False, 'south': False, 'west': False, 'east': True}
Action: suck
Step 15
[['.' '.' '.' '.' '.']
['.' '.' 'D' '.' 'R']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
----------
Agent position (row, col): (1, 4), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': False, 'east': True}
Action: west
Step 16
[['.' '.' '.' '.' '.']
['.' '.' 'D' 'R' '.']
['.' 'D' 'D' '.' '.']
['.' '.' '.' '.' '.']
['.' '.' '.' '.' '.']]
```

Agent position (row, col): (1, 3), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}

Action: west

Step 17

[['.' '.' '.' '.' '.']

['.' '.' 'R' '.' '.']

['.' 'D' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (1, 2), Dirty: True, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}

Action: suck

Step 18

[['.' '.' '.' '.' '.']

['.' '.' 'R' '.' '.']

['.' 'D' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (1, 2), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}

Action: west

Step 19

[['.' '.' '.' '.' '.']

['.' 'R' '.' '.' '.']

['.' 'D' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (1, 1), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}

Action: west

Step 20

[['.' '.' '.' '.' '.']

['R' '.' '.' '.' '.']

['.' 'D' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (1, 0), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': True, 'east': False}

Action: south

Step 21

[['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']

['R' 'D' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (2, 0), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': True, 'east': False}

Action: east

Step 22

[['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']

['.' 'R' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (2, 1), Dirty: True, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}

Action: suck

Step 23

[['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']

['.' 'R' 'D' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (2, 1), Dirty: False, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}

Action: east

Step 24

[['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' 'R' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

Agent position (row, col): (2, 2), Dirty: True, Bumpers: {'north': False, 'south': False, 'west': False, 'east': False}

Action: suck

Step 25

[['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' 'R' '.' '.']

['.' '.' '.' '.' '.']

['.' '.' '.' '.' '.']]

```
----------

All squares are clean! Task finished.

Total actions taken (Performance): 24
```

4. Phân tích Kết quả

Kết quả chạy thử cung cấp một minh chứng rõ ràng và chi tiết về cách tác nhân phản xạ dựa trên mô hình hoạt động hiệu quả trong thực tế.

Hành vi có thể dự đoán và Logic hai giai đoạn:

Giai đoạn Định vị (Steps 1-6): Log cho thấy rõ ràng tác nhân bắt đầu ở một vị trí ngẫu nhiên (2, 3). Nó ngay lập tức thực hiện chiến lược định vị: di chuyển liên tục về hướng west (Steps 1-4) cho đến khi va vào tường (bumpers: {'west': True}), sau đó tiếp tục di chuyển về hướng north (Steps 5-6) cho đến khi va vào tường phía bắc. Tại cuối Step 6, nó đã đến được góc tham chiếu (0, 0) và hoàn thành giai đoạn định vị.

Giai đoạn Dọn dẹp (Steps 7-24): Ngay từ Step 7, sau khi đã xác định được vị trí, tác nhân bắt đầu quá trình dọn dẹp có hệ thống. Nó di chuyển sang east trên hàng đầu tiên (Steps 7, 9, 11, 12), chuyển hướng south khi đến cuối hàng (Step 13), sau đó di chuyển sang west trên hàng thứ hai (Steps 15, 16, 18, 19). Mẫu hình "cày ruộng" này được lặp lại một cách hoàn hảo, đảm bảo không một ô nào bị bỏ sót.

Hiệu quả Vượt trội:

Tác nhân đã hoàn thành nhiệm vụ dọn dẹp toàn bộ căn phòng chỉ trong 24 bước. Đây là một hiệu suất cực kỳ cao. Con số này bao gồm 6 bước để định vị, các bước di chuyển qua các ô và các hành động 'suck' cần thiết.

So với một tác nhân ngẫu nhiên có thể mất hàng trăm bước hoặc không bao giờ hoàn thành, sự hiệu quả của tác nhân dựa trên mô hình là không thể bàn cãi. Nó loại bỏ hoàn toàn các bước đi lãng phí và các hành động không cần thiết.

Phân tích Chi tiết Hành vi:

Ưu tiên dọn dẹp: Tác nhân luôn ưu tiên hành động 'suck' khi phát hiện bụi bẩn trước khi tiếp tục di chuyển. Ví dụ, tại Step 8, khi đến ô (0, 1) và thấy bẩn (Dirty: True), nó ngay lập tức thực hiện hành động 'suck'. Chỉ sau khi ô đó sạch (Step 9), nó mới tiếp tục di chuyển sang east. Điều này cho thấy logic if dirty: return 'suck' hoạt động chính xác.

Một quan sát nhỏ: Tại Step 6, sau khi vừa đến góc (0, 0) và hoàn thành định vị, tác nhân thực hiện một hành động 'suck' dù ô này không bẩn (Dirty: False). Đây có thể là một hành động thừa phát sinh từ cách logic được cấu trúc trong lượt chuyển tiếp giữa hai giai đoạn. Mặc dù đây là một sự thiếu hiệu quả nhỏ (một bước đi lãng phí), nó không ảnh hưởng đến tính đúng đắn chung hay khả năng hoàn thành nhiệm vụ của tác nhân.

Tính Toàn diện và Đáng tin cậy:

Quan trọng nhất, chiến lược di chuyển có hệ thống đảm bảo rằng mọi ô vuông trong phòng đều được tác nhân ghé thăm. Điều này mang lại sự đảm bảo về tính toàn diện (completeness) — tác nhân chắc chắn sẽ làm sạch toàn bộ căn phòng.

Sự kết hợp giữa hiệu quả và tính toàn diện làm cho tác nhân này trở nên cực kỳ đáng tin cậy để giải quyết bài toán đặt ra.

5. Kết luận

Việc triển khai thành công tác nhân phản xạ dựa trên mô hình trong Task 3 là một bước tiến lớn so với các tác nhân trước đó. Nó không chỉ hoàn thành nhiệm vụ dọn dẹp phòng một cách đáng tin cậy mà còn làm điều đó với hiệu suất cao hơn nhiều. Kết quả này nhấn mạnh tầm quan trọng của trạng thái nội bộ trong việc xây dựng các tác nhân thông minh có khả năng thực hiện các nhiệm vụ phức tạp một cách hiệu quả trong môi trường của chúng.

4. Nhiệm vụ 4:

Trong bài thực nghiệm này, ba loại agent (Randomized, Simple Reflex và Model-based Reflex) được đánh giá hiệu quả trong việc làm sạch môi trường vacuum với

các kích thước phòng khác nhau (5×5, 10×10, 100×100). Mỗi agent được chạy 100 lần với môi trường sinh ngẫu nhiên (xác suất bụi p=0.2).

Kết quả được so sánh dựa trên số hành động trung bình mà mỗi agent thực hiện cho đến khi phòng sạch hoặc đạt giới hạn bước.

Phân tích kết quả theo kích thước phòng

Room size 5×5:

Randomized agent cần trung bình khoảng 240 bước để hoàn thành.

Simple Reflex agent hoạt động hiệu quả hơn, chỉ cần khoảng 113 bước.

Model-based Reflex agent thể hiện tốt nhất, chỉ tốn khoảng 25 bước nhờ khả năng ghi nhớ trạng thái và định hướng di chuyển chính xác, tránh lặp lại ô đã thăm.

Room size 10×10:

Randomized agent mất trung bình khoảng 1000 bước do việc di chuyển ngẫu nhiên trở nên kém hiệu quả khi môi trường mở rộng.

Simple Reflex agent vẫn duy trì hiệu suất tương đối (≈705 bước), phản ứng tốt với bụi nhưng vẫn có nhiều hành động dư thừa.

Model-based Reflex agent vượt trội hơn hẳn, chỉ cần khoảng 122 bước, cho thấy mô hình lưu trữ trạng thái giúp giảm đáng kể số hành động lặp lại.

Room size 100×100:

Khi kích thước phòng rất lớn, cả ba agent đều cần nhiều bước, nhưng Model-based Reflex agent vẫn thể hiện hiệu quả vượt trội (≈12076 bước) so với Randomized (≈18149 bước) và Simple Reflex (≈16071 bước).

Điều này chứng tỏ khả năng ghi nhớ và định hướng theo mô hình giúp Model-based Reflex bao phủ không gian rộng hiệu quả hơn, trong khi hai agent còn lại dễ rơi vào tình trạng lặp lại hoặc bỏ sót vùng chưa dọn.

So sánh giữa ba agent:

Randomized agent**:** di chuyển ngẫu nhiên, đơn giản nhưng kém hiệu quả khi kích thước phòng tăng. Không có bộ nhớ nên thường lặp lại các ô đã thăm, làm tăng số bước đáng kể.

Simple Reflex agent**:** phản ứng nhanh với môi trường hiện tại, hoạt động tốt ở phòng nhỏ và trung bình, nhưng khi phòng mở rộng, việc không có cơ chế nhớ khiến nó hoạt động kém tối ưu.

Model-based Reflex agent**:** thể hiện đúng ưu thế của mô hình phản xạ có bộ nhớ. Agent này ghi nhận trạng thái môi trường, tránh di chuyển trùng lặp, chủ động tìm tới các vùng chưa dọn và vì vậy thực hiện ít hành động hơn đáng kể so với hai loại agent còn lại.

Mặc dù Model-based Reflex agent thể hiện kết quả tốt nhất, mô hình vẫn có thể được cải thiện thêm bằng cách:

Tích hợp heuristic hoặc thuật toán tìm đường tối ưu hơn.

Giảm hành động thừa khi khám phá không gian rộng.

Ngoài ra, có thể mở rộng nghiên cứu bằng cách:

Thử nghiệm với các xác suất bụi khác nhau (p).

Đánh giá thêm tiêu chí: tỷ lệ phòng sạch, năng lượng tiêu hao, hoặc thời gian hoàn thành.

Kết quả cho thấy Model-based Reflex agent là lựa chọn tối ưu nhất ở cả ba quy mô phòng. Randomized chỉ hiệu quả ở không gian nhỏ, Simple Reflex tốt ở mức trung bình, còn Model-based Reflex vượt trội nhờ khả năng ghi nhớ và định hướng hợp lý. Ở phòng rất lớn (100×100), dù cả ba đều gần đạt giới hạn bước, Model-based Reflex vẫn giữ được hiệu suất tốt hơn đáng kể — chứng minh rằng việc tích hợp mô hình và trí nhớ giúp agent làm việc thông minh và hiệu quả hơn, tương tự nguyên lý hoạt động của robot hút bụi hiện đại có bản đồ và lập kế hoạch di chuyển.

5. Nhiệm vụ 5:

1. Nếu phòng là hình chữ nhật nhưng không biết kích thước

- Tác nhân Ngẫu nhiên (Randomized Agent): Tác nhân này không quan tâm đến kích thước hay hình dạng của phòng. Nó sẽ tiếp tục di chuyển ngẫu nhiên và va vào tường như bình thường. Hiệu suất của nó vốn đã rất tệ, nên sẽ không có gì thay đổi lớn.

- Tác nhân Phản xạ Đơn giản (Simple Reflex Agent): Tác nhân này hoạt động khá tốt. Nó không cần biết kích thước phòng vì nó chỉ phản ứng với các bức tường ngay trước mặt. Nó sẽ đi lang thang trong phòng hình chữ nhật và

dọn dẹp một cách ngẫu nhiên. Nó vẫn không hiệu quả, nhưng nó sẽ không bị "hỏng".

- Tác nhân Dựa trên Mô hình (Model-based Reflex Agent): Sẽ thất bại hoàn toàn. Tác nhân này được lập trình với giả định về một căn phòng hình vuông `n x n`. Nếu kích thước thực tế khác đi (ví dụ 5x8 thay vì 5x5), kế hoạch di chuyển theo hàng của nó sẽ sai. Nó sẽ nghĩ rằng nó đã đến cuối hàng trong khi thực tế chưa, làm cho tọa độ nội bộ của nó bị lệch hoàn toàn so với vị trí thực. Kế hoạch dọn dẹp có hệ thống sẽ sụp đổ.

2. Nếu khu vực dọn dẹp có hình dạng bất thường (ví dụ: có hành lang)

- Tác nhân Ngẫu nhiên: Giống như trên, nó không quan tâm. Nó sẽ đi lang thang một cách ngẫu nhiên trong mọi không gian mà nó có thể vào được.

- Tác nhân Phản xạ Đơn giản: Tác nhân này xử lý tốt các hình dạng bất thường. Đối với nó, một hành lang chỉ là một không gian hẹp với các bức tường. Nó sẽ đi lang thang qua hành lang và vào các phòng khác một cách tự nhiên. Đây là ưu điểm lớn của việc không có một kế hoạch cứng nhắc.

- Tác nhân Dựa trên Mô hình: Sẽ thất bại hoàn toàn. Kế hoạch di chuyển "cày ruộng" của nó chỉ hoạt động trên một hình chữ nhật trống. Khi gặp một bức tường bất ngờ ở giữa (như ở góc của một phòng hình chữ L) hoặc một hành lang hẹp, logic di chuyển của nó sẽ bị phá vỡ. Nó không có khả năng tự tìm đường trong các không gian phức tạp.

3. Nếu trong phòng có chướng ngại vật (đồ đạc)

- Tác nhân Ngẫu nhiên: Nó sẽ liên tục cố gắng di chuyển vào chướng ngại vật vì nó phớt lờ cảm biến va chạm. Điều này làm cho nó càng kém hiệu quả hơn.

- Tác nhân Phản xạ Đơn giản: Tác nhân này xem chướng ngại vật như những bức tường nhỏ. Nó sẽ va vào chúng, cảm nhận được, và thử một hướng đi khác. Nó có thể di chuyển xung quanh chướng ngại vật, nhưng cũng có thể bị kẹt trong một không gian hẹp giữa chướng ngại vật và tường.

- Tác nhân Dựa trên Mô hình: Sẽ thất bại hoàn toàn. Kế hoạch của nó yêu cầu các hàng di chuyển không bị cản trở. Một chướng ngại vật ở giữa đường đi sẽ làm hỏng toàn bộ lộ trình. Tác nhân không có logic để đi vòng qua một vật cản

bất ngờ; nó sẽ coi đó là bức tường cuối phòng và làm sai lệch toàn bộ bản đồ nội bộ của nó.

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

- Tác nhân Ngẫu nhiên: Không bị ảnh hưởng, vì nó vốn dĩ không sử dụng cảm biến này để ra quyết định.
- Tác nhân Phản xạ Đơn giản: Đây là một vấn đề lớn. Nếu cảm biến báo "sạch" ở một nơi "bẩn" (false negative), nó sẽ bỏ qua vết bẩn đó. Căn phòng có thể sẽ không bao giờ được dọn sạch hoàn toàn. Nếu cảm biến báo "bẩn" ở một nơi "sạch" (false positive), nó sẽ lãng phí một lượt để hút bụi một ô đã sạch. Điều này làm giảm hiệu suất.
- Tác nhân Dựa trên Mô hình: Tương tự như tác nhân phản xạ đơn giản. Lộ trình di chuyển của nó vẫn hoàn hảo, nhưng việc dọn dẹp sẽ không đáng tin cậy. Nó sẽ bỏ sót các vết bẩn (false negative) hoặc lãng phí năng lượng (false positive). Mục tiêu làm sạch hoàn toàn căn phòng sẽ không đạt được.

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

- Tác nhân Ngẫu nhiên: Không bị ảnh hưởng, vì nó phớt lờ cảm biến này.
- Tác nhân Phản xạ Đơn giản: Nó sẽ cố gắng di chuyển xuyên tường. Môi trường sẽ ngăn nó lại, nhưng nó đã lãng phí một hành động. Điều này làm giảm hiệu suất nhưng không làm hỏng hoàn toàn tác nhân.
- Tác nhân Dựa trên Mô hình: THẢM HỌA. Đây là trường hợp tệ nhất. Trong giai đoạn định vị, nếu nó không "nhìn thấy" bức tường phía tây hoặc phía bắc, nó sẽ không bao giờ biết mình đang ở góc và sẽ bị kẹt trong một vòng lặp vô tận. Trong giai đoạn dọn dẹp, nếu nó đến cuối một hàng và cảm biến không hoạt động, tọa độ nội bộ của nó sẽ bị cập nhật sai. Từ đó, toàn bộ bản đồ trong đầu của nó sẽ bị lệch so với thực tế và tác nhân sẽ hoàn toàn bị "lạc".

6. Nhiệm vụ nâng cao: Cảm biến bụi bẩn không hoàn hảo

- Thay đổi môi trường mô phỏng của bạn để chạy các thử nghiệm cho bài toán sau: Cảm biến bụi bẩn có 10% khả năng đưa ra kết quả sai. Thực hiện các thử nghiệm để quan sát điều này làm thay đổi hiệu suất của ba triển khai (tác nhân phản xạ, tác nhân phản xạ dựa trên mô hình và tác nhân mục tiêu) như

thế nào. Tác nhân phản xạ dựa trên mô hình của bạn có thể không làm sạch được toàn bộ căn phòng, vì vậy cần đo lường hiệu suất một cách khác, đó là sự đánh đổi giữa chi phí năng lượng và số ô chưa được làm sạch.

- Thiết kế và triển khai một giải pháp cho tác nhân dựa trên mô hình để làm sạch tốt hơn. Chứng minh sự cải thiện này bằng các thử nghiệm.
- Một môi trường mô phỏng (MôiTrường) và bốn loại tác nhân (TácNhânPhảnXạ, TácNhânDựaTrênMôHình, TácNhânMụcTiêu, TácNhânDựaTrênMôHìnhCảiTiến) đã được triển khai để giải quyết bài toán này.

a. Môi trường

Môi trường đã được thay đổi để mô phỏng sự không chắc chắn của cảm biến:

- Hàm cảm_nhận_bụi_bẩn(): Mô phỏng cảm biến. Nó trả về trạng thái bẩn/sạch thực tế của ô hiện tại, nhưng với 10% xác suất, nó sẽ trả về kết quả ngược lại.
- Chi phí Năng lượng: Hàm di_chuyển_và_hút() tính toán chi phí năng lượng cho mỗi hành động. Hành động hút bụi tốn 11 đơn vị năng lượng (1 cơ bản + 10 cho hút), va chạm tường tốn thêm 5 đơn vị.

b. Các tác nhân

- Tác nhân Phản xạ: Hoạt động dựa trên tri giác sai lệch. Nếu cảm biến báo bẩn, nó hút. Nếu không, nó di chuyển ngẫu nhiên.
- Tác nhân Mục tiêu: Đây là tác nhân có "kiến thức thần thánh" vì nó biết trước tất cả các vị trí bẩn. Nó di chuyển trực tiếp đến các ô bẩn và hút bụi. Tuy nhiên, nó vẫn bị ảnh hưởng bởi lỗi cảm biến khi xác nhận vị trí bẩn.
- Tác nhân dựa trên Mô hình (Cơ bản): Triển khai một "bản đồ nội tâm" (bản_đồ_nội_tâm) để ghi nhớ trạng thái của các ô đã thăm. Tác nhân này sẽ cố gắng di chuyển đến các ô chưa khám phá. Logic cơ bản này không được tối ưu để xử lý lỗi cảm biến và có thể không hiệu quả.
- Tác nhân dựa trên Mô hình (Cải tiến):
    + Chiến lược Di chuyển: Để đảm bảo phủ hết toàn bộ diện tích phòng, tác nhân này sử dụng chiến lược di chuyển theo đường zig-zag. Nó di

chuyển theo một đường thẳng ngang, sau đó đi xuống một ô và đổi hướng di chuyển.

+ Xử lý Lỗi Cảm biến: Mặc dù cảm biến có thể báo sai, nhưng chiến lược di chuyển có hệ thống này đảm bảo robot sẽ đi qua mọi ô. Nếu cảm biến ban đầu báo sai một ô bẩn, tác nhân vẫn sẽ quay lại ô đó trong quá trình di chuyển tiếp theo, và lần cảm nhận tiếp theo có thể đúng. Điều này làm tăng khả năng làm sạch toàn bộ phòng.

- Kết quả:

Loại tác nhân	Chi phí năng lượng	Số ô chưa gạch	Đánh đổi (Tổng chi phí)
Phản xạ	495	26	3095
Dựa trên mô hình (Cơ bản)	715	7	1415
Mục tiêu	730	2	930
Dựa trên mô hình (Cải tiến)	200	46	4800

Dựa trên dữ liệu trong bảng, bạn có thể phân tích hiệu suất của từng tác nhân và lý giải tại sao chúng lại có kết quả như vậy trong môi trường có lỗi cảm biến.

+ Tác nhân Phản xạ: Có chi phí năng lượng tương đối thấp (495) nhưng lại có số ô chưa sạch rất cao (26). Điều này cho thấy chiến lược di chuyển ngẫu nhiên của nó không hiệu quả. Khi cảm biến báo sai, tác nhân này sẽ bỏ qua một ô bẩn và không bao giờ quay lại, vì nó không có bộ nhớ.

+ Tác nhân dựa trên mô hình (Cơ bản): Mặc dù chi phí năng lượng cao hơn (715), nó lại làm sạch hiệu quả hơn nhiều (chỉ còn 7 ô chưa sạch). Việc sử dụng một bản đồ nội tâm giúp nó ghi nhớ những vị trí đã đi qua và cố gắng khám phá những khu vực chưa biết, giảm thiểu việc

bỏ sót. Kết quả đánh đổi tổng thể của nó tốt hơn nhiều so với tác nhân phản xạ.

+ Tác nhân Mục tiêu: Tác nhân này đạt hiệu suất tốt nhất với số ô chưa sạch chỉ là 2. Tuy nhiên, chi phí năng lượng của nó cao (730). Điều này cho thấy ngay cả khi có "kiến thức thần thánh" (biết trước vị trí bụi bẩn), lỗi cảm biến vẫn có thể khiến nó bỏ sót một vài ô. Mặc dù vậy, đây vẫn là tác nhân hiệu quả nhất trong số các tác nhân được thử nghiệm.
+ Tác nhân dựa trên mô hình (Cải tiến): Tác nhân này có chi phí năng lượng cực thấp (200), nhưng lại thất bại thảm hại trong việc làm sạch (46 ô chưa sạch). Điều này cho thấy chiến lược di chuyển zig-zag mà bạn đã triển khai không xử lý tốt lỗi cảm biến. Rất có khả năng, khi cảm biến báo sai một ô bẩn thành sạch, tác nhân bỏ qua nó và tiếp tục lộ trình, không bao giờ quay lại.

7. Nhiệm vụ nâng cao:

1. Môi trường có vật cản (Obstacles)

Trong môi trường mở rộng này, ta bổ sung thêm các ô vuông ngẫu nhiên đóng vai trò là vật cản. Các vật cản có đặc điểm:

- Chúng cũng kích hoạt bumper sensor khi agent di chuyển vào.
- Agent không biết trước vị trí của vật cản.

Thí nghiệm đề xuất:

- So sánh hiệu suất (performance) của ba loại agent đã cài đặt trước đó trong môi trường có vật cản.
- Quan sát: tỉ lệ làm sạch, số bước bị va chạm, và độ hiệu quả năng lượng.

Phân tích thách thức:

- Với agent phản xạ đơn giản (simple reflex agent), khả năng thích ứng sẽ kém vì nó không có trí nhớ về vị trí đã gặp vật cản.
- Với agent có mô hình (model-based agent), cần cập nhật bản đồ môi trường động, từ đó tránh được việc lặp lại va chạm.
- Để cải thiện hiệu suất, agent cần xây dựng một bản đồ cục bộ (local map) ghi nhớ những ô bị chặn, và lập kế hoạch đường đi thay thế.

Hướng phát triển:

- Áp dụng kỹ thuật pathfinding như A* hoặc Dijkstra để tìm đường tránh vật cản.
- Tích hợp cơ chế học hỏi (reinforcement learning) để điều chỉnh hành vi khi phát hiện vật cản nhiều lần.

2. Agent trong môi trường không biết kích thước và có vật cản

Ở kịch bản khó hơn, ta giả định agent:

- Không biết kích thước môi trường (chỉ biết nó hình chữ nhật).
- Không biết vị trí ban đầu của mình.
- Không biết vị trí của vật cản.

Ý tưởng triển khai:

- Agent luôn di chuyển đến ô chưa được thăm hoặc chưa được làm sạch gần nhất.
- Điều này tương ứng với việc duyệt theo chiến lược depth-first search (DFS).

Khó khăn:

- Cần duy trì một cấu trúc dữ liệu để ghi nhớ: ô nào đã làm sạch, ô nào chưa.
- Đảm bảo agent không bị kẹt trong vòng lặp giữa các vật cản.

Giải pháp:

- Dùng graph search với bộ nhớ ánh xạ giữa các trạng thái (state memory).
- Khi gặp vật cản, loại bỏ cạnh/đường đi liên quan khỏi đồ thị.
- Kết hợp chiến lược khám phá (exploration) để dần mở rộng hiểu biết về môi trường.

3. Utility-based Agent trong môi trường động

Trong phiên bản nâng cao này, môi trường là một phòng 5×5, và:

- Mỗi ô có một xác suất cố định để trở nên bẩn trở lại.
- Môi trường được biểu diễn bằng một ma trận 2D chứa các giá trị xác suất.
- Utility của một trạng thái = số ô đang sạch trong phòng.
- Agent phải tối đa hóa utility trong suốt 100.000 bước, ứng với một lần sạc đầy pin.

Yêu cầu triển khai:

- Agent cần học xác suất ô bẩn lại, thay vì được biết trước.

- Trong mỗi bước, agent phải ước lượng utility kỳ vọng của các hành động khác nhau và chọn hành động tốt nhất.

Thách thức:

- Đây là một bài toán khó vì:
  - Agent phải vừa khám phá (explore) để học xác suất,
  - Vừa khai thác (exploit) để tối ưu số ô sạch.
- Bài toán tương tự với học Markov Decision Process (MDP) hoặc Reinforcement Learning.

Hướng giải quyết:

- Dùng ước lượng xác suất thực nghiệm: theo dõi số lần một ô bị bẩn trở lại trong một khoảng thời gian.
- Sử dụng chiến lược expected utility maximization: chọn hành động đem lại lợi ích lớn nhất theo kỳ vọng.
- Với môi trường phức tạp hơn, có thể áp dụng Q-learning hoặc value iteration để tối ưu hóa.

Kết luận

Phần mở rộng "More Advanced Implementation" giúp mô phỏng các tình huống thực tế hơn, nơi môi trường có tính không chắc chắn, phức tạp và thay đổi theo thời gian.

- Với vật cản, agent phải học cách ghi nhớ và tránh né.
- Với môi trường không xác định, agent cần chiến lược khám phá toàn diện.
- Với môi trường động có xác suất, agent phải áp dụng các khái niệm nâng cao về utility và học tăng cường.

CREATE A SIMPLE REFLEX-BASED LUNAR LANDER AGENT
(XÂY DỰNG AGENT LUNAR LANDER DỰA TRÊN PHẢN XẠ ĐƠN GIẢN)

I. Mục tiêu.

Mục tiêu của bài tập này là xây dựng một tác tử Simple Reflex-Based Agent trong môi trường Lunar Lander thuộc thư viện Gymnasium. Cụ thể:

- Hiểu cách hoạt động của môi trường Lunar Lander (trạng thái, hành động, phần thưởng).
- Xây dựng một agent phản ứng theo các quy tắc đơn giản dạng *if–then*, không sử dụng học tăng cường.
- Thử nghiệm agent để đánh giá khả năng hạ cánh và rút ra nhận xét về ưu, nhược điểm của phương pháp reflex agent.
- Làm nền tảng so sánh với các phương pháp nâng cao hơn như Reinforcement Learning (RL).

II. Giới thiệu.

Trong lĩnh vực trí tuệ nhân tạo, các tác tử (agent) được xây dựng để tương tác với môi trường, nhận thông tin từ môi trường và đưa ra hành động nhằm đạt mục tiêu nhất định. Một trong những loại agent cơ bản là Simple Reflex Agent – tác tử phản xạ đơn giản, chỉ đưa ra quyết định dựa vào trạng thái hiện tại theo một số quy tắc được định nghĩa sẵn. Môi trường Lunar Lander là một ví dụ điển hình thường được dùng để minh họa và thử nghiệm các thuật toán điều khiển agent. Trong môi trường này, agent cần điều khiển các động cơ để giúp tàu vũ trụ hạ cánh an toàn lên bề mặt Mặt Trăng. Đây là một môi trường có mức độ khó vừa phải, đủ để thể hiện rõ sự khác biệt giữa các loại agent.

III. Cơ sở lý thuyết.

1. Khái niệm Agent

- Agent là một thực thể có khả năng quan sát môi trường xung quanh thông qua cảm biến (sensors) và đưa ra hành động thông qua bộ chấp hành (actuators). Agent được thiết kế nhằm tối ưu hóa mục tiêu trong một môi trường cụ thể.
- Mô hình tổng quát của một agent được mô tả theo công thức:

$$\text{Agent} = f(\text{Percept}) \rightarrow \text{Action}$$

- Trong đó:
  - Percept: thông tin nhận được từ môi trường tại một thời điểm.
  - Action: hành động mà agent đưa ra để tác động trở lại môi trường.

2. Simple Reflex Agent

Simple Reflex Agent là loại tác tử cơ bản nhất, hoạt động dựa trên quy tắc IF–THEN (nếu – thì). Agent này chỉ dựa vào trạng thái hiện tại để ra quyết định, mà không quan tâm đến lịch sử hành động trước đó.

Ví dụ:

- Nếu góc tàu lệch sang trái quá nhiều → bật động cơ bên phải.
- Nếu vận tốc rơi quá nhanh → bật động cơ chính để giảm tốc.

Ưu điểm:

- Đơn giản, dễ triển khai.
- Phù hợp với các môi trường ít phức tạp.

Nhược điểm:

- Không có trí nhớ, không học được từ kinh nghiệm.
- Không tối ưu trong các môi trường phức tạp, nhiều yếu tố bất định.

3. Môi trường Lunar Lander

Môi trường Lunar Lander nằm trong bộ thư viện Gymnasium, mô phỏng nhiệm vụ điều khiển tàu vũ trụ hạ cánh an toàn xuống bề mặt Mặt Trăng.

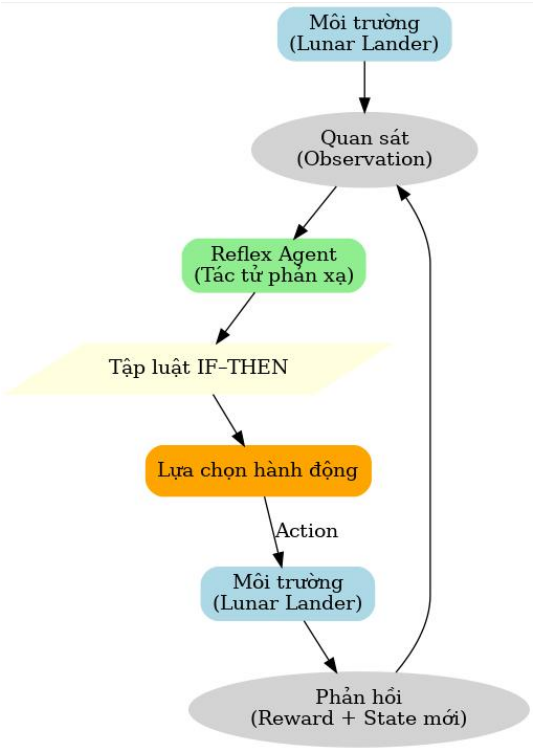
- Trạng thái (State): gồm 8 giá trị quan sát, bao gồm: vị trí ngang, vị trí dọc, vận tốc ngang, vận tốc dọc, góc tàu, tốc độ xoay, và trạng thái tiếp xúc của 2 chân đáp.
- Hành động (Action): có 4 lựa chọn:

  0. Không làm gì.
  1. Bật động cơ chính (đẩy thẳng đứng).
  2. Bật động cơ phụ bên trái.
  3. Bật động cơ phụ bên phải.
- Phần thưởng (Reward):
  - Hạ cánh đúng vị trí an toàn → nhận thưởng cao.
  - Hạ cánh sai vị trí hoặc bị rơi → nhận điểm phạt.
  - Sử dụng nhiên liệu cũng bị trừ điểm, nhằm khuyến khích tối ưu hóa hành động.

4. Nguyên lý hoạt động của Simple Reflex Agent trong Lunar Lander

Trong môi trường này, Simple Reflex Agent sẽ hoạt động theo nguyên tắc:

1. Quan sát trạng thái hiện tại (vị trí, vận tốc, góc nghiêng, trạng thái tiếp xúc).
2. So sánh trạng thái với tập luật IF–THEN.
3. Chọn hành động phù hợp (bật động cơ chính, trái, phải hoặc không làm gì).
4. Nhận phản hồi từ môi trường (reward + state mới).
5. Lặp lại cho đến khi kết thúc episode (tàu hạ cánh hoặc nổ).



IV. Kết luận

Qua quá trình xây dựng và thử nghiệm, nhóm đã triển khai thành công một Simple Reflex-Based Agent trong môi trường Lunar Lander. Agent này được thiết kế dựa trên các luật if–then đơn giản, giúp tàu vũ trụ có thể thực hiện các hành động cơ bản như giảm tốc độ rơi, điều chỉnh góc nghiêng và duy trì trạng thái cân bằng.

Kết quả cho thấy:

- Agent có thể điều khiển tàu hạ cánh trong một số tình huống đơn giản.
- Tuy nhiên, do không có trí nhớ hay khả năng học tập từ kinh nghiệm, hiệu suất của agent còn hạn chế, đặc biệt trong các trường hợp phức tạp hoặc khi môi trường thay đổi nhiều.
- So với các phương pháp học tăng cường (Reinforcement Learning), Simple Reflex Agent thiếu tính tối ưu và khả năng tổng quát hóa.