Intelligent Agents: Reflex-Based Agents for the Vacuum-cleaner World

**1 — Giới thiệu & PEAS:**

**Mục tiêu:** xây dựng môi trường mô phỏng robot hút bụi trên lưới n×n, cài đặt 3 dạng agent:

* simple randomized agent (ngẫu nhiên),
* simple reflex agent (phản xạ đơn giản),
* model-based reflex agent (giữ trạng thái, lập kế hoạch quét).

**PEAS:**

* **Performance measure:** mỗi hành động tiêu tốn 1 energy unit; performance = tổng energy dùng để làm sạch toàn bộ phòng (tức số hành động khi phòng sạch). Nếu agent chạy hết năng lượng (max\_steps) mà vẫn còn bẩn thì đó là trường hợp thất bại — trong thí nghiệm cần thống nhất cách tính (xem phần thí nghiệm).
* **Environment:** phòng n×nn (n = 5, 10, 100 trong thí nghiệm), ô ban đầu bẩn với xác suất p (ví dụ p=0.2). Agent biết kích thước n. Bắt đầu ở ô ngẫu nhiên.
* **Actuators:** suck, north, east, south, west.
* **Sensors:** bumper tại 4 hướng (north/east/south/west) (True nếu có tường/không đi tiếp được), và dirt sensor báo ô hiện tại bẩn (True/False).

**2 — Môi trường mô phỏng**

**Ý tưởng:** lưu trạng thái phòng bằng ma trận Boolean room kích thước n x n, True = dirty, False = clean. Lưu vị trí robot robot\_pos = (row, col). Vòng lặp: mỗi bước lấy percepts (bumpers, dirty), gọi agent\_function(bumpers, dirty), thực hiện action (thay đổi vị trí hoặc làm sạch ô), tăng bộ đếm bước. Dừng khi room.any() == False (tức không còn ô dirty) hoặc khi steps == max\_steps.

**Chi tiết:**

* Mỗi action (move hoặc suck) tăng 1 đơn vị energy / steps.
* Nếu agent muốn di chuyển vào tường, môi trường phải *không* cho di chuyển (hoặc coi là hành động không làm thay đổi vị trí nhưng vẫn tiêu tốn 1 energy). Tránh cho agent đi ra ngoài mảng.
* Khi suck: chỉ đặt room[row,col] = False.
* Khi in trạng thái (verbose), hiển thị ma trận để dễ debug: R cho robot, D cho dirty, . cho clean.
* Cách tính performance khi agent không hoàn thành: nếu agent không dọn sạch hết trong max\_steps, ghi nhận performance = max\_steps **và** báo remaining\_dirty = number of dirty squares left. Trong phân tích, sử dụng cả mean(steps) cho những run hoàn thành và proportion\_success (tỷ lệ run hoàn thành), hoặc dùng mean(steps\_with\_penalty) (ví dụ + penalty cho squares chưa dọn).

**3 — Agent programs**

**3.1 Simple randomized agent**

* **Ý tưởng:** chọn action hoàn toàn ngẫu nhiên trong set ["north","east","south","west","suck"].
* **Không rational:** agent có thể cố bump vào tường nhiều lần, có thể không hút bụi khi ô bẩn.

actions = ["north", "east", "south", "west", "suck"]

def simple\_randomized\_agent(bumpers, dirty):

return np.random.choice(actions)

**3.2 Simple reflex agent**

* **Ý tưởng:** nếu ô hiện tại bẩn -> suck. Nếu không thì chọn ngẫu nhiên một hướng **không bị chắn** bởi bumper (lọc các hướng từ bumpers = False).
* Đây là agent phản xạ: không nhớ gì, chỉ dựa trên percept hiện tại.

def simple\_reflex\_agent(bumpers, dirty):

if dirty: return "suck"

possible = [d for d, blocked in bumpers.items() if not blocked]

if possible: return np.random.choice(possible)

else: return "suck"

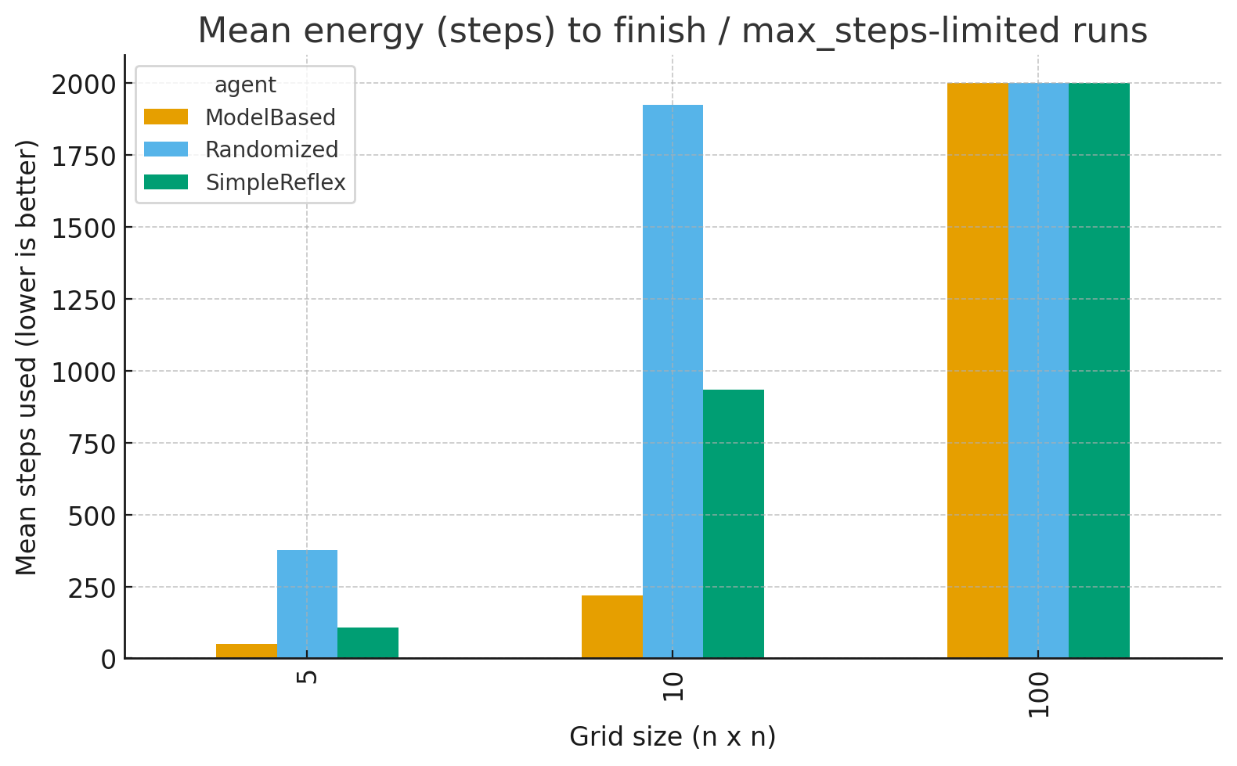
**3.3 Model-based reflex agent**

* **Mục tiêu:** giữ trạng thái (belief/state) để lập kế hoạch quét toàn bộ phòng hiệu quả (ít di chuyển thừa).
* **State cần có (tối thiểu):**
  + n (kích thước), position = (row,col) (ước đoán vị trí hiện tại),
  + visited[r,c] (đã từng đến ô này chưa),
  + known\_dirty[r,c] (quan sát thấy ô này bẩn/sạch trước đó),
  + located (đã định vị toạ độ đầy đủ so với góc nào chưa).
* **Chiến lược đơn giản & hiệu quả:**
  + Định vị góc (ví dụ: di chuyển tới bức tường phía bắc, sau đó sang tường phía tây) để xác định góc trên-trái (NW). Khi gặp bumpers['north']==True và bumpers['west']==True thì ta biết vị trí = (0,0).
  + Tạo một **đường đi zig-zag cố định** (scanline): hàng 0 từ trái sang phải, hàng 1 từ phải sang trái, ... — đảm bảo thăm từng ô một lần (nếu không có obstacles).
  + Khi tới ô bẩn thì suck. Khi sạch thì di chuyển theo đường kế hoạch tới ô chưa thăm tiếp theo.
  + Cập nhật visited và known\_dirty sau mỗi percept.
* **Các lưu ý thực tế:**
  + Nếu agent bắt đầu ở ô ngẫu nhiên, bước 1 (định vị) phải di chuyển về bắc/tây cho đến khi gặp bumper. Nên chỉ thử 2 hướng để định vị góc NW (hoặc tương tự dùng bắc và tây).
  + Nếu có obstacles (không phải tường), chiến lược định vị và zig-zag cần mở rộng (thăm tất cả ô có thể là một bài toán tìm đường — DFS/BFS).

**3.4 Kết quả chạy**

| **Size** | **Agent** | **Mean steps** | **Success rate** |
| --- | --- | --- | --- |
| 5x5 | Randomized | 375.82 | 0.64 |
| 5x5 | SimpleReflex | 108.56 | 0.98 |
| 5x5 | ModelBased | 49.28 | 1.00 |
| 10x10 | Randomized | 1925.86 | 0.18 |
| 10x10 | SimpleReflex | 934.60 | 0.98 |
| 10x10 | ModelBased | 219.06 | 1.00 |
| 100x100 | Randomized | 2000.00\* | 0.00 |
| 100x100 | SimpleReflex | 2000.00\* | 0.00 |
| 100x100 | ModelBased | 2000.00\* | 0.00 |

\* = bằng max\_steps do hầu hết run không hoàn tất với giới hạn 2000 bước trên lưới 100×100.



**4 — Nhận xét và kết luận**

* **Model-based** tốt nhất trên lưới nhỏ/ vừa (định vị + quét zig-zag đảm bảo thăm mọi ô): mean steps nhỏ và success\_rate = 1.0 cho n=5,10.
* **Simple reflex** tốt hơn randomized: phản xạ suck khi gặp bẩn giúp tiết kiệm nhiều so với agent hoàn toàn ngẫu nhiên.
* Với **100×100** và max\_steps=2000, tất cả agent không hoàn tất (success\_rate = 0). Nếu bạn muốn so sánh cho 100×100, cần cho max\_steps lớn hơn (ví dụ ≥ n\*n) hoặc dùng metric khác (số ô đã dọn trong năng lượng giới hạn).

MAZE

**1 — Giới thiệu:**

* **Bài toán:** tìm đường từ ô bắt đầu S đến ô đích G trên lưới (maze). Mỗi bước di chuyển (UP/DOWN/LEFT/RIGHT) có chi phí 1.
* **State:** vị trí (row, col).
* **Hành động:** di chuyển 4 hướng (không đi vào tường).
* **Mục tiêu:** tìm dãy hành động đưa từ S tới G.
* **Metric báo cáo:** path cost (độ dài đường đi), số nút mở rộng (nodes expanded), độ sâu lớn nhất (max depth), kích thước lớn nhất của frontier (max frontier).

**2 — Breadth-First Search (BFS):**

* **Mô tả:** BFS thăm các đỉnh theo từng lớp sâu: mở tất cả các nút ở depth d trước khi mở nút ở depth d+1. Do chi phí mỗi bước bằng nhau, BFS tìm được đường ngắn nhất (theo số bước).
* **Pseudo code:**

BFS(start):

frontier = queue chứa node start

reached = { start }

while frontier không rỗng:

node = frontier.dequeue()

if node.state == goal: return path(node)

for each action dẫn tới child\_state:

if child\_state not in reached:

reached.add(child\_state)

frontier.enqueue(child\_node)

return failure

* **Tính chất:**
  + Time complexity: O(b^d) — b: branching factor, d: depth của lời giải tối ưu
  + Space complexity: O(b^d) — frontier + reached (rất tốn bộ nhớ)
* **Ưu / Nhược:**
  + Ưu: luôn tìm đường ngắn nhất, đơn giản.
  + Nhược: tiêu thụ bộ nhớ lớn khi d lớn / môi trường mở.

**3 — Depth-First Search (DFS):**

* **Mô tả:** DFS mở rộng một nhánh tới tận cùng trước khi quay lui. Để tránh vòng lặp trong đồ thị có chu trình, dùng **kiểm tra chu trình trên đường hiện hành** (kiểm tra xem trạng thái child có nằm trong path từ gốc tới node hiện tại hay không).
* **Pseudo code:**

DFS(start):

stack = stack chứa node start

while stack không rỗng:

node = stack.pop()

if node.state == goal: return path(node)

for each action dẫn tới child\_state:

if child\_state NOT trong đường từ root tới node (cycle check):

push child\_node vào stack

return failure

* **Tính chất:**
  + Time complexity: O(b^m) — b: branching factor, m: độ sâu tối đa của cây
  + Space complexity: O(b\*m) — chỉ lưu đường hiện tại, tiết kiệm bộ nhớ
* **Ưu / Nhược:**
  + Ưu: bộ nhớ rất thấp (thích hợp không gian lớn).
  + Nhược: có thể tìm đường rất dài, dễ lặp lại khám phá vùng giống nhau (nếu không dùng reached toàn cục), phụ thuộc thứ tự mở rộng.

**4 — Greedy Best-First Search (GBFS):**

* **Mô tả:** GBFS chọn để mở node mà theo **heuristic** (ước lượng tới goal) là “gần goal nhất”. Chỉ dùng h(n) — không xét chi phí đã đi (g(n)). Thường nhanh nhưng không đảm bảo đường ngắn nhất.
* **Heuristic thường dùng:** Manhattan distance.
* **Pseudo code:**

GBFS(start):

frontier = priority queue ordered theo h(state)

reached = { start }

while frontier không rỗng:

node = pop frontier (min h)

if node.state == goal: return path(node)

for each child:

if child not in reached:

reached.add(child)

push child vào frontier with priority = h(child)

return failure

* **Tính chất:**
  + Time complexity: phụ thuộc heuristic; worst-case O(b^m)
  + Space complexity: O(b^m) (cần lưu frontier)
* **Ưu / Nhược:**
  + Ưu: thường mở ít node hơn BFS nếu heuristic tốt; nhanh trong nhiều trường hợp.
  + Nhược: có thể dẫn vào "bẫy" heuristic (đi tới gần mục tiêu theo h nhưng thực tế đường đi xa), không tối ưu.

**5 — A\* Search:**

* **Mô tả:** A\* kết hợp chi phí đã đi g(n) và heuristic h(n) thành f(n) = g(n) + h(n) và luôn mở node có f nhỏ nhất. Nếu h là **admissible** (không overestimate) và **consistent**, thì A\* là **optimal** và **complete**.
* **Pseudo code:**

A\*(start):

frontier = priority queue ordered theo f = g + h

reached = { start: g(start)=0 }

while frontier không rỗng:

node = pop frontier (min f)

if node.state == goal: return path(node)

for each child:

g\_new = node.g + cost(node->child)

if child not in reached OR g\_new < reached[child]:

reached[child] = g\_new

f = g\_new + h(child)

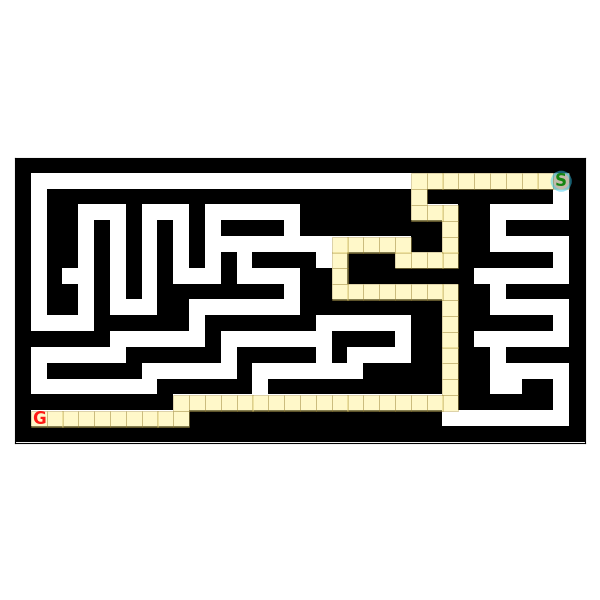
push child vào frontier với priority f

return failure

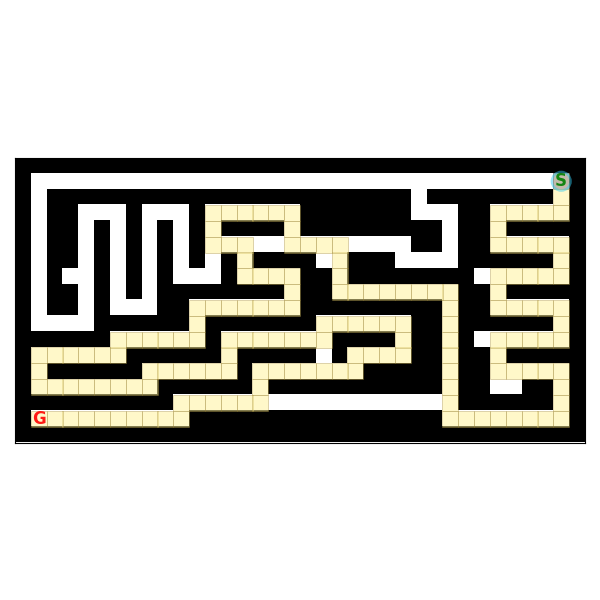
* **Tính chất:**
  + Time complexity: O(b^d) worst-case (nhưng thường ít hơn BFS)
  + Space complexity: O(b^d) (cần lưu frontier và reached)
* **Ưu / Nhược:**
  + Ưu: cân bằng giữa tìm nhanh và tối ưu; thường tốt nhất nếu heuristic tốt.
  + Nhược: vẫn tiêu tốn bộ nhớ lớn; hiệu năng phụ thuộc vào chất lượng heuristic.

**6 — Thực hành:**

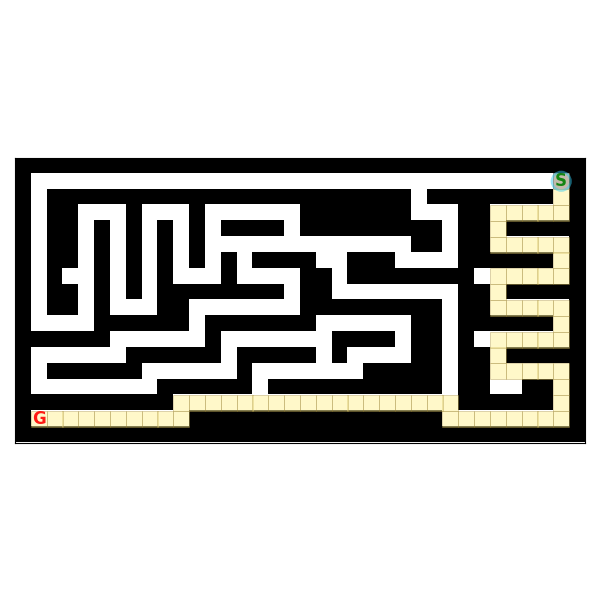
BFS:



DFS:



GBFS:



A\*:

