A picture containing application

Description automatically generated**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

Icon

Description automatically generated with low confidence

**BÁO CÁO**

**HỌC PHẦN: TRÍ TUỆ NHÂN TẠO NÂNG CAO**

**LAB 02**

**Sinh viên thực hiện**

Nguyễn Anh Khoa – 3122410185

Huỳnh Khôi Nguyên - 3122410267

Trần Gia Nuyễn – 3122410274

**TP. HCM THÁNG 9/2025**

**MỤC LỤC**

[**PHÂN CÔNG 3**](#_Toc210599236)

[**1. Lunar lander 4**](#_Toc210599237)

[**2. Solving a Maze Using a Goal-based Agent 7**](#_Toc210599238)

[**3. Robot vaccum 29**](#_Toc210599239)

PHÂN CÔNG

|  |  |  |
| --- | --- | --- |
| **MSSV** | **Họ tên** | **Phân công** |
| 3122410185 | Nguyễn Anh Khoa | Lab2\_01, báo cáo, slide |
| 3122410267 | Huỳnh Khôi Nguyên | Lab2\_02, báo cáo, slide |
| 3122410274 | Trần Gia Nguyễn (nhóm trưởng) | Lab2\_02, tổng hợp báo cáo, slide |

1. Lunar lander

import gymnasium as gym

def run\_episode(agent\_function, max\_steps=1000):

    """Run one episode in the LunarLander-v3 environment using the provided agent."""

    # Initialize the environment

    env = gym.make("LunarLander-v3", render\_mode="human")

    # Reset the environment to generate the first observation (use seed=42 in reset to get reproducible results)

    observation, info = env.reset()

    # run one episode

    for \_ in range(max\_steps):

        # call the agent function to select an action

        action = agent\_function(observation)

        print (f"Obs: {observation} -> Action: {action}")

        # step: execute an action in the environment

        observation, reward, terminated, truncated, info = env.step(action)

        env.render()

        if terminated:

            print(f"Final Reward: {reward}")

            break

    env.close()

    return reward

- Hàm run\_episode mô phỏng 1 lần chạy thử trong môi trường Lunar-Lander-v3 với agent được cung cấp sẵn và có số bước tối đa là max\_steps. Cụ thể như sau, env = gym.make("LunarLander-v3", render\_mode="human" khởi tạo môi trường LunarLander-v3, với chế độ hiển thị hình ảnh là “human”. observation, info = env.reset() đặt lại môi trường cho 1 lần chạy thử mới, và sinh ra observation (trạng thái ban đầu) và info (metadata). Sau đó lặp lại từng bước, action = agent\_function(observation) ,tại mỗi bước agent sẽ chọn 1 hành động dựa trên trạng thái hiện tại và sinh ra trạng thái mới (observation, reward, terminated, truncated, info = env.step(action)). Trong đó, observation là trạng thái mới, reward là phần thưởng, terminated là cờ lệnh kiểm tra đã kết thúc chưa (tàu đã hạ cánh thành công hay thất bại), truncated là cờ lệnh kiểm tra đã hết bước giới hạn, info là thông tin phụ. Tiếp theo, env.render() hiển thị hình ảnh mô phỏng để theo dõi. Khi kết thúc episode (tàu đã hạ cánh thành công hay nổ), đóng môi trường và trả về phần thưởng của episode đó.

from enum import Enum

class Act(Enum):

    LEFT = 1

    RIGHT = 3

    MAIN = 2

    NO\_OP = 0

class Obs(Enum):

    X = 0

    Y = 1

    VX = 2

    VY = 3

    ANGLE = 4

    ANGULAR\_VELOCITY = 5

    LEFT\_LEG\_CONTACT = 6

    RIGHT\_LEG\_CONTACT = 7

- Lớp Act bao gồm 4 hành động để hạ điều chỉnh tàu trong lúc hạ cánh. Với LEFT là bật động cơ trái, RIGHT bật động cơ phải, MAIN bật động cơ chính và NO\_OP không hành động gì cả. Lớp Obs là các giá trị của 1 observation, bao gồm 8 giá trị, X là vị trí ngang của tàu so với bệ đáp (miền giá trị từ [-1.5, 1,5]), Y là vị trí dọc của tàu so với bệ đáp (giá trị từ [0, 1.5]), VX là vận tốc ngang (miền giá trị [-2, 2]), VY là vận tốc dọc (giá trị từ [-2, 2]), ANGLE ứng với góc nghiêng của tàu (khoảng [-3.14, 3.14] radian), ANGULAR\_VELOCITY là tốc độ quay của tàu (giá trị từ [-5, 5] rad/s), LEFT\_LEG\_CONTACT là trạng thái chân tàu trái (có giá trị {0,1} với 0 là chưa chạm đất, 1 là chạm đất), RIGHT\_LEG\_CONTACT là trạng thái chân tàu phải (có giá trị tương tự với chân trái).

# Code goes here

def test\_agent\_function(observation):

    """A simple agent function."""

    # bật động cơ chính nếu tàu đang rơi quá nhanh

    if observation[Obs.VY.value] < -.3:

        return Act.MAIN.value

    if observation[Obs.ANGLE.value] <-.1: # bật động cơ trái, nếu tàu có góc nghiêng về bên trái quá -0.1 radian

        return Act.LEFT.value

    elif observation[Obs.ANGLE.value] > .1: # bật động cơ phải, nếu tàu có góc nghiêng về bên phải quá 0.1 radian

        return Act.RIGHT.value

    if observation[Obs.X.value] < -.5: # bật động cơ trái, nếu tàu bị lệch trái so với vị trí đáp

        return Act.LEFT.value

    elif observation[Obs.X.value] > .5: # bật động cơ phải, nếu tàu bị lệch phải so với vị trí đáp

        return Act.RIGHT.value

    return Act.NO\_OP.value  # Không hành động gì

- Hàm test\_agent\_function sẽ trả về các giá trị tương ứng với các hành động của class Act. Nếu tàu đang rơi thẳng quá nhanh thì bật động cơ chính để giảm tốc. Tương tự với các điều kiện còn lại. Cuối cùng, khi tàu đang ở điều kiện hoàn hảo trong lúc đáp đất trả về giá trị 0 ứng với việc không hành động gì hết (không bật động cơ chính hay động cơ trái hay động cơ phải).

def run\_episodes(agent\_function, n=1000):

    """Run multiple episodes with the given agent and return the rewards for each episode."""

    return [run\_episode\_test(agent\_function) for \_ in range(n)]

rewards = run\_episodes(test\_agent\_function, n = 1000)

print(f"Success Rate: {np.sum(np.array(rewards) == 100)}/{len(rewards)}")

- Hàm run\_episodes chạy nhiều phép thử và trả về danh sách phần thưởng ứng với mỗi lần chạy thử. Sau đó, hàm được sử dụng để chạy nhiều lần thử (episode) với agent là hàm test\_agent\_function (đã được đề cập ở trên) và số lần thử là 1000 (n = 1000). Cuối cùng, np.sum(np.array(rewards) == 100)}/{len(rewards) được dùng để đếm xem có bao nhiêu lần thử có kết quả bằng 100 và in ra màn hình.

1. Solving a Maze Using a Goal-based Agent

**Yêu cầu**

Xây dựng các bài toán tìm kiếm bằng cách sử dụng các thành phần chính như trạng thái ban đầu, tập hành động và trạng thái đích trong một môi trường xác định, quan sát đầy đủ.

Cài đặt và so sánh các thuật toán tìm kiếm bao gồm BFS, DFS, GBFS, A\* và IDS để tìm đường trong mê cung.

Phân tích hiệu suất của thuật toán bằng cách đo lường chi phí đường đi, số lượng nút được mở rộng, độ sâu và mức sử dụng bộ nhớ trên nhiều loại mê cung khác nhau.

Sử dụng các công cụ trực quan hóa để biểu diễn đường đi trong mê cung, hỗ trợ việc gỡ lỗi và phân tích.

**Task 1:Định nghĩa các thành phần của bài toán tìm kiếm và xác định kích thước vấn đề**

**Định nghĩa các thành phần bài toán**

Trạng thái khởi đầu(Initial state) : Là trạng thái khởi đầu của bài toán

Ví dụ: đối với bài toán mê cung thì là ô bắt đầu có ký hiệu là ‘S’

Hành động(Action): Hành động của agent sau khi qua bước tiếp nhận thông tin từ môi trường

Ví dụ: Trong bài toán mê cung thì hành động di chuyển trái, phải , lên , xuống

Hàm chuyển trạng thái (Transition model): Là làm giúp agent di chuyển sau khi quyết định được hành động.

Ví dụ: Với bài toán mê cung , thì hàm nhận thông tin đầu vào là trạng thái hiện tại và action để chuyển sang trạng thái khác , với action là “lên”, “xuống”,”trái”,”phải”.

Trạng thái đích(Goal state) : Trạng thái đích của bài toán

Chi phí đường đi ( path cost ): Chi phí của bài toán khi thực hiện thuật toán

Ví dụ : Với bài toán mê cung thì đây là số ô di chuyển từ trạng thái đầu tới trạng thái đích

**Kích thước vấn đề**

n: kích thước của không gian trạng thái

Ví dụ: Kích thước của bài toán mê cung là số cột x số dòng

d: Độ sâu của cây đã được tối ưu đại diện cho đường đi từ trạng thái đầu đến trạng thái đích

Ví dụ : số ô di chuyển tối ưu từ trạng thái đầu đến trạng thái đích

m: độ sâu tối đa của cây , đường đi từ trạng thái ban đầu đến trạng thái đích chưa được tối ưu

Ví dụ: khoảng cách tối đa của bài tóan mê cung : bằng số cột x số dòng - 1

b: số node con tối đa của 1 nhánh

Ví dụ là 4 ( trái , phải , lên , xuống)

**Task 2:Tìm hiểu và thực hiện breadth-first and depth first search**

**Code:**

**Định nghĩa node**

import maze\_helper as mh

import matplotlib.pyplot as plt

import  numpy as np

class Node:

    def \_\_init\_\_(self, pos, parent, action, cost):

        self.pos = tuple(pos)    # the state; positions are (row,col)

        self.parent = parent     # Node cha .

        self.action = action     # Hành động để robot di chuyển lên xuống trái phải

        self.cost = cost         # chi phí đường đi

    def \_\_str\_\_(self):

        return f"pos = ({self.pos[0]} ,{self.pos[1]}); parent = {self.parent}"

    def get\_path\_from\_root(self):

        """returns nodes on the path from the root to the current node."""

        node = self

        path = [node]

        while not node.parent is None:

            node = node.parent

            path.append(node)

        path.reverse()

        return(path)

**1.BFS**:

Ý tưởng của thuật toán là với mỗi 1 node ta sẽ sinh ngẫu nhiên những node liền kề chưa được thăm vào hàng đợi , cứ như thế ta sẽ thực hiện thao tác với mỗi node đầu của hàng đợi cho đến khi tìm được goal hoặc hàng đợi rỗng

Code:

def BFS(maze):

    new\_maze = maze.copy().astype(str)

    start = mh.find\_pos(maze, what="S")

    end = mh.find\_pos(maze, what="G")

    reached = set()

    frontier = []

    action\_effects = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    start\_node = Node(start, None, action\_effects, 0)

    frontier.append(start\_node)

    while len(frontier) > 0:

        node = frontier.pop(0)

        if node.pos == end:

            path = node.get\_path\_from\_root()

            for n in path:

                if new\_maze[n.pos] not in ["S", "G"]:

                    new\_maze[n.pos] = "P"

            return new\_maze

        if node.pos in reached:

            continue

        reached.add(node.pos)

        for action in action\_effects:

            new\_pos = (node.pos[0] + action[0], node.pos[1] + action[1])

            if mh.look(maze, new\_pos) != "X" and new\_pos not in reached:

                if new\_maze[new\_pos] not in ["S", "G"]:

                    new\_maze[new\_pos] = "."  # đánh dấu đã explore

                new\_node = Node(new\_pos, node, action, node.cost + 1)

                frontier.append(new\_node)

    return new\_maze

Đầu vào: là mê cung có kích thước m x n , với m là số dòng và n là số cột , các giá trị trong ma trận là chướng ngại vật và ‘S’ , ’G’

Đầu ra : là mê cung với các node đã được đánh dấu đã duyệt cùng với đường đi từ start đến goal

2.DFS:

Ý tưởng của thuật toán là đi càng sâu càng tốt theo một nhánh trước, nếu gặp ngõ cụt hoặc trạng thái đích không đạt, thì quay lui và thử nhánh khác.

def DFS(maze):

    new\_maze = maze.copy().astype(str)

    start = mh.find\_pos(maze, what="S")

    end = mh.find\_pos(maze, what="G")

    reached = set()

    frontier = []

    action\_effects = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    start\_node = Node(start, None, action\_effects, 0)

    frontier.append(start\_node)

    while len(frontier) > 0:

        node = frontier.pop()

        if node.pos == end:

            path = node.get\_path\_from\_root()

            for n in path:

                if new\_maze[n.pos] not in ["S", "G"]:

                    new\_maze[n.pos] = "P"

            return new\_maze

        if node.pos in reached:

            continue

        reached.add(node.pos)

        for action in action\_effects:

            new\_pos = (node.pos[0] + action[0], node.pos[1] + action[1])

            if mh.look(maze, new\_pos) != "X" and new\_pos not in reached:

                if new\_maze[new\_pos] not in ["S", "G"]:

                    new\_maze[new\_pos] = "."  # đánh dấu đã explore

                new\_node = Node(new\_pos, node, action, node.cost + 1)

                frontier.append(new\_node)

    return new\_maze

Đầu vào: là mê cung có kích thước m x n , với m là số dòng và n là số cột , các giá trị trong ma trận là chướng ngại vật và ‘S’ , ’G’

Đầu ra : là mê cung với các node đã được đánh dấu đã duyệt cùng với đường đi từ start đến goal

Câu hỏi : BFS và DFS đã giải quyết vấn đề khi vòng lặp như thế nào khi không sử dụng **REACHED** ?

Trả lời:

* **BFS with reached set:** BFS luôn cần reached (hay visited) để tránh thăm lại cùng một state. Nếu bỏ reached, BFS có thể lặp vô hạn trong graph có chu trình.
* **DFS without reached**: DFS thường triển khai mà không lưu toàn bộ reached để tiết kiệm bộ nhớ. Tuy nhiên, nếu không kiểm soát thì DFS cũng dễ rơi vào vòng lặp vô hạn. Do đó, DFS **ít nhất phải có cycle checking** trong nhánh hiện tại (path-based checking):
  + Khi mở rộng node mới, không cho phép quay lại một state đã xuất hiện trong path hiện tại.
  + Điều này ngăn DFS đi vòng tròn vô hạn trong một nhánh, nhưng không ngăn việc khám phá lại cùng state theo nhánh khác.

**Kết luận:**

* BFS cần reached để đảm bảo đầy đủ và tránh lặp.
* DFS không có reached toàn cục → phải có **cycle checking trong path hiện tại**, nếu không sẽ bị kẹt trong vòng lặp

Câu hỏi : Các cài đặt của bạn có đầy đủ và tối ưu không? Giải thích lý do. Độ phức tạp về thời gian và không gian của từng cài đặt là gì? Đặc biệt thảo luận sự khác biệt về độ phức tạp không gian giữa BFS và DFS.

Trả lời:

* Với BFS :
  + Đầy đủ: Có, vì BFS sẽ luôn tìm ra lời giải nếu tồn tại đường đi từ start đến goal.
  + Tối ưu: Có, vì BFS luôn tìm được đường đi ngắn nhất (ít bước nhất) trong môi trường với chi phí đồng nhất.
* Với DFS
  + Đầy đủ: Không đảm bảo, vì DFS có thể đi vào vòng lặp vô hạn nếu không kiểm tra chu trình, hoặc bỏ lỡ lời giải nếu không duyệt hết các nhánh.
  + Tối ưu: Không, vì DFS có thể tìm ra lời giải không phải ngắn nhất.

**Độ phức tạp về thời gian và bộ nhớ:**

* **Với BFS:**
  + *Time complexity*: O(b^d), với b là branching factor (số nhánh tối đa), d là độ sâu lời giải tối ưu.
  + *Space complexity*: O(b^d), vì BFS lưu trữ tất cả các node ở mỗi mức trước khi chuyển sang mức tiếp theo (frontier có thể rất lớn).
* **Với DFS:**
  + *Time complexity*: O(b^m), với m là độ sâu tối đa của cây tìm kiếm.
  + *Space complexity*: O(bm), vì DFS chỉ lưu trữ các node trên đường đi hiện tại (stack), nên sử dụng ít bộ nhớ hơn nhiều so với BFS.

**So sánh không gian giữa BFS và DFS:**

Tính đầy đủ và tối ưu của các giải thuật

* **BFS (Breadth-First Search)**
  + *Đầy đủ (Completeness)*: Có, BFS luôn tìm được lời giải nếu tồn tại (với hệ số phân nhánh hữu hạn).
  + *Tối ưu (Optimality)*: Có, BFS tìm lời giải ngắn nhất nếu chi phí mỗi bước bằng nhau.
  + *Độ phức tạp*:
    - Thời gian: O(b^d) với b là hệ số phân nhánh, d là độ sâu của nghiệm gần nhất.
    - Không gian: O(b^d) (do phải lưu toàn bộ frontier và reached).
* **DFS (Depth-First Search, không dùng reached toàn cục)**
  + *Đầy đủ*: Không, DFS có thể rơi vào vòng lặp vô hạn hoặc bỏ lỡ nghiệm nông nếu không có giới hạn độ sâu.
  + *Tối ưu*: Không, vì DFS có thể trả về nghiệm dài hơn trong khi vẫn còn nghiệm ngắn hơn.
  + *Độ phức tạp*:
    - Thời gian: O(b^m) với m là độ sâu tối đa (có thể vô hạn).
    - Không gian: O(bm) (chỉ cần lưu một nhánh từ gốc đến lá).
* **GBFS (Greedy Best-First Search)**
  + *Đầy đủ*: Không đảm bảo, vì có thể bị kẹt nếu heuristic dẫn vào ngõ cụt.
  + *Tối ưu*: Không, GBFS không đảm bảo tìm được đường đi ngắn nhất.
  + *Độ phức tạp*:
    - Thời gian: phụ thuộc mạnh vào heuristic, trường hợp xấu là O(b^d).
    - Không gian: cùng bậc với BFS, do lưu trong hàng đợi ưu tiên.
* **A\***
  + *Đầy đủ*: Có, nếu heuristic chấp nhận được (admissible) và hệ số phân nhánh hữu hạn.
  + *Tối ưu*: Có, nếu heuristic vừa admissible vừa consistent.
  + *Độ phức tạp*:
    - Thời gian: vẫn mang tính chất lũy thừa, nhưng thường ít hơn BFS khi heuristic tốt.
    - Không gian: cũng lũy thừa, do phải lưu tất cả node đã sinh.

So sánh BFS và DFS về không gian

* **BFS**: cần bộ nhớ cho tất cả node ở độ sâu d, tăng theo hàm mũ O(b^d).
* **DFS**: chỉ cần bộ nhớ cho một đường đi và các nhánh chưa mở rộng, tỷ lệ thuận với độ sâu O(bm).
* **Kết luận**: BFS tối ưu nhưng tốn bộ nhớ, DFS tiết kiệm bộ nhớ nhưng không tối ưu và không đầy đủ trong đồ thị vô hạn.

**Task 3:Tìm hiểu và thực hiện Greedy best-first search và A\* search**

**1.Greedy best-first search:**

Thuật toán luôn chọn mở rộng node có giá trị heuristic (ước lượng khoảng cách đến đích, ví dụ: Manhattan distance) nhỏ nhất trước. GBFS chỉ quan tâm đến việc đến goal nhanh nhất theo ước lượng, không xét chi phí đã đi.

**Code:**

def Greedy\_best\_first\_search(maze):

    new\_maze = maze.copy().astype(str)

    start = mh.find\_pos(maze, what = "S")

    end =  mh.find\_pos(maze, what = "G")

    reached = set()

    frontier = []

    action\_effects = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    start\_node = Node(start, None,action\_effects, 0)

    frontier.append(start\_node)

    while len(frontier) > 0:

        # Sort the frontier based on the heuristic (Manhattan distance)

        frontier.sort(key=lambda node: abs(node.pos[0] - end[0]) + abs(node.pos[1] - end[1]))

        node = frontier.pop(0)  # Pop the node with the lowest heuristic value

        if node.pos == end:

            path = node.get\_path\_from\_root()

            for n in path:

                if new\_maze[n.pos] not in ["S", "G"]:

                    new\_maze[n.pos] = "P"

            return new\_maze

        if node.pos in reached:

            continue

        reached.add(node.pos)

        for action in action\_effects:

            new\_pos = (node.pos[0] + action[0], node.pos[1] + action[1])

            if mh.look(maze, new\_pos) != "X":

                if new\_maze[new\_pos] not in ["S", "G"]:

                    new\_maze[new\_pos] = "."

                new\_node = Node(new\_pos, node, action, node.cost + 1)

                frontier.append(new\_node)

    return new\_maze

Đầu vào: là mê cung có kích thước m x n , với m là số dòng và n là số cột , các giá trị trong ma trận là chướng ngại vật và ‘S’ , ’G’

Đầu ra : là mê cung với các node đã được đánh dấu đã duyệt cùng với đường đi từ start đến goal

2.A\* search:

A\* mở rộng node có tổng chi phí nhỏ nhất, với tổng chi phí là: **f(n) = g(n) + h(n)**

**Code:**

def A\_star\_search(maze):

    new\_maze = maze.copy().astype(str)

    start = mh.find\_pos(maze, what = "S")

    end =  mh.find\_pos(maze, what = "G")

    reached = set()

    frontier = []

    action\_effects = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    start\_node = Node(start, None,action\_effects, 0)

    frontier.append(start\_node)

    while len(frontier) > 0:

        # Sort the frontier based on the f(n) = g(n) + h(n)

        frontier.sort(key=lambda node: node.cost + abs(node.pos[0] - end[0]) + abs(node.pos[1] - end[1]))

        node = frontier.pop(0)  # Pop the node with the lowest f(n) value

        if node.pos == end:

            path = node.get\_path\_from\_root()

            for n in path:

                if new\_maze[n.pos] not in ["S", "G"]:

                    new\_maze[n.pos] = "P"

            return new\_maze

        if node.pos in reached:

            continue

        reached.add(node.pos)

        for action in action\_effects:

            new\_pos = (node.pos[0] + action[0], node.pos[1] + action[1])

            if mh.look(maze, new\_pos) != "X":

                if new\_maze[new\_pos] not in ["S", "G"]:

                    new\_maze[new\_pos] = "."

                new\_node = Node(new\_pos, node, action, node.cost + 1)

                frontier.append(new\_node)

    return new\_maze

**Câu hỏi: Bạn đã thực hiện thành công và tối ưu chưa ? Độ phức tạp về thời gian và bộ nhớ?**

Đánh giá về độ đầy đủ, tối ưu và độ phức tạp

* **BFS (Breadth-First Search)**
  + *Đầy đủ*: Có (sẽ tìm thấy nghiệm nếu tồn tại).
  + *Tối ưu*: Có, nếu mọi bước đi có chi phí bằng nhau.
  + *Độ phức tạp*:
    - Thời gian: O(b^d) với b = hệ số phân nhánh, d = độ sâu nghiệm nông nhất.
    - Không gian: O(b^d) (rất tốn bộ nhớ).
* **DFS (Depth-First Search, không có reached toàn cục, chỉ cycle check)**
  + *Đầy đủ*: Không (có thể lặp vô hạn hoặc bỏ lỡ nghiệm).
  + *Tối ưu*: Không (có thể trả về đường đi dài hơn).
  + *Độ phức tạp*:
    - Thời gian: O(b^m) với m = độ sâu tối đa.
    - Không gian: O(bm) (ít tốn bộ nhớ, tỷ lệ thuận với độ sâu).
* **GBFS (Greedy Best-First Search)**
  + *Đầy đủ*: Không đảm bảo (có thể bị kẹt ở heuristic kém).
  + *Tối ưu*: Không.
  + *Độ phức tạp*:
    - Thời gian: phụ thuộc vào heuristic, xấu nhất O(b^d).
    - Không gian: O(b^d) (ưu tiên queue cũng lớn như BFS).
* **A\***
  + *Đầy đủ*: Có (nếu heuristic admissible và nhất quán).
  + *Tối ưu*: Có, với heuristic admissible.
  + *Độ phức tạp*:
    - Thời gian: có thể lũy thừa nhưng thường ít hơn BFS nhờ heuristic.
    - Không gian: vẫn lũy thừa, phải lưu tất cả node đã sinh.

So sánh BFS và DFS về không gian

* BFS cần lưu toàn bộ frontier ở độ sâu d ⇒ O(b^d) (rất lớn).
* DFS chỉ cần lưu một nhánh + node anh em ⇒ O(bm) (rất nhỏ so với BFS).
* Đây là điểm khác biệt quan trọng: BFS tối ưu nhưng tốn bộ nhớ, DFS tiết kiệm bộ nhớ nhưng không tối ưu và không đầy đủ.

**Task 4: So sánh và thảo luận**

So sánh kết quả thuật toán với từng mê cung:

+ smaill\_maze.txt

A table with numbers and text

AI-generated content may be incorrect.

**+Medium\_maze.txt**

**A table with numbers and symbols

AI-generated content may be incorrect.**

+ Large\_maze.txt

A table with numbers and a number

AI-generated content may be incorrect.

+ loops\_maze.txt

A table with numbers and symbols

AI-generated content may be incorrect.

+L\_maze.txt

A table with numbers and symbols

AI-generated content may be incorrect.

+Empty\_maze:

A table with numbers and symbols

AI-generated content may be incorrect.

+Empty\_maze\_2.txt

A table with numbers and text

AI-generated content may be incorrect.

+Open\_maze.txt

A table with numbers and symbols

AI-generated content may be incorrect.

**Thảo luận:**

**Hiểu rõ bản chất từng thuật toán:**

Mỗi thuật toán có ưu, nhược điểm riêng. BFS luôn tìm được đường đi ngắn nhất nhưng tốn nhiều bộ nhớ. DFS tiết kiệm bộ nhớ hơn nhưng dễ bị lặp vô hạn và không đảm bảo tối ưu.

**Tầm quan trọng của heuristic:**

Các thuật toán có heuristic như Greedy Best-First Search và A\* cho kết quả nhanh và hiệu quả hơn nhiều nếu chọn được hàm heuristic phù hợp (ví dụ: khoảng cách Manhattan). A\* thường tối ưu cả về chi phí và thời gian.

**Kiểm soát vòng lặp và trạng thái đã thăm:**

Việc kiểm tra trạng thái đã thăm (cycle checking) là rất quan trọng, đặc biệt với DFS, để tránh lặp vô hạn trong mê cung có vòng lặp.

**Quản lý bộ nhớ:**

BFS tiêu tốn nhiều bộ nhớ khi không gian trạng thái lớn, trong khi DFS và IDS tiết kiệm hơn nhưng có thể mất nhiều thời gian hơn.

**Không có thuật toán nào là tốt nhất cho mọi trường hợp:**

Việc lựa chọn thuật toán phù hợp phụ thuộc vào đặc điểm của bài toán (kích thước mê cung, có nhiều vòng lặp hay không, yêu cầu tối ưu hay chỉ cần tìm ra lời giải).

**Advanced task : IDS và Multiple Goal**

**IDS:** là sự kết hợp giữa DFS và BFS, nhằm tận dụng ưu điểm của cả hai:

* IDS thực hiện nhiều lần tìm kiếm theo chiều sâu (DFS), mỗi lần với một giới hạn độ sâu (depth limit) tăng dần từ 0 lên đến khi tìm được lời giải.
* Ở mỗi lần lặp, IDS chạy DFS nhưng chỉ cho phép đi sâu tối đa đến depth limit hiện tại.
* Nếu chưa tìm thấy lời giải ở độ sâu đó, tăng depth limit lên 1 và lặp lại quá trình.
* IDS đảm bảo tính đầy đủ (complete) và tối ưu (optimal) như BFS, nhưng sử dụng bộ nhớ ít như DFS.

**Code:**

def DLS(node, depth, new\_maze, end, action\_effects):

        if node.pos == end:

            path = node.get\_path\_from\_root()

            for n in path:

                if new\_maze[n.pos] not in ["S", "G"]:

                    new\_maze[n.pos] = "P"

            return True

        if depth <= 0:

            return False

        for action in action\_effects:

            new\_pos = (node.pos[0] + action[0], node.pos[1] + action[1])

            if mh.look(new\_maze, new\_pos) != "X":

                if new\_maze[new\_pos] not in ["S", "G"]:

                    new\_maze[new\_pos] = "."

                new\_node = Node(new\_pos, node, action, node.cost + 1)

                if DLS(new\_node, depth - 1, new\_maze, end, action\_effects):

                    return True

        return False

def IDS(maze):

    new\_maze = maze.copy().astype(str)

    start = mh.find\_pos(maze, what = "S")

    end =  mh.find\_pos(maze, what = "G")

    action\_effects = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    max\_depth = 1

    while True:

        start\_node = Node(start, None, None, 0)

        if DLS(start\_node, max\_depth, new\_maze, end, action\_effects):

            return new\_maze

        max\_depth += 1

    return -1

**Thử nghiệm với small\_maze và medium\_maze:**

**Với smaill\_maze**

**A maze with a red path in the middle

AI-generated content may be incorrect.**

Thời gian chạy của mê cung nhỏ có thời gian chạy dưới 1 phút.

Nhưng với mê cung trung bình thì thuật toán không thể giải được vì thời gian quá lâu , vấn đề nằm ở việc chọn chiều sau cho mỗi lần chạy , mỗi lần chạy với độ sâu mới thì ta vẫn phải chạy lại toàn bộ những độ sâu trước đó , độ phức tạp là b^d với d là độ sâu , và b là số trạng thái con được sinh ra

**Multiple Goal:**

**Câu hỏi :** Tạo một vài mê cung với nhiều mục tiêu bằng cách thêm một hoặc hai mục tiêu nữa vào mê cung cỡ vừa. Tác nhân hoàn thành khi nó tìm thấy một trong các mục tiêu. Giải mê cung bằng các cài đặt của bạn cho DFS, BFS và IDS. Thực hiện các thí nghiệm để cho thấy các cài đặt nào tìm ra lời giải tối ưu và cài đặt nào không. Thảo luận lý do tại sao lại như vậy.

A screenshot of a game

AI-generated content may be incorrect.

A screenshot of a game

AI-generated content may be incorrect.

Thảo luận

1. **Kết quả thực nghiệm:**
   * **BFS (Breadth-First Search):**
     + BFS luôn tìm ra đường đi ngắn nhất đến một trong các mục tiêu, vì nó mở rộng các trạng thái theo mức (level).
     + BFS đảm bảo tìm được lời giải tối ưu (ít bước nhất).
   * **DFS (Depth-First Search):**
     + DFS có thể tìm thấy một mục tiêu bất kỳ, nhưng không đảm bảo đó là mục tiêu gần nhất.
     + Đường đi tìm được thường không tối ưu (có thể dài hơn cần thiết).
     + Kết quả phụ thuộc vào thứ tự mở rộng các nhánh.
   * **IDS (Iterative Deepening Search):**
     + IDS cũng đảm bảo tìm được đường đi ngắn nhất đến một trong các mục tiêu, giống như BFS, nhưng sử dụng ít bộ nhớ hơn.
     + Tuy nhiên, IDS có thể tốn nhiều thời gian hơn do phải lặp lại nhiều lần với các giới hạn độ sâu khác nhau.
2. **Lý do:**
   * **BFS và IDS** đều mở rộng các trạng thái theo mức, nên khi gặp một trong các mục tiêu đầu tiên, đó chắc chắn là mục tiêu gần nhất (tối ưu).
   * **DFS** đi sâu theo một nhánh trước, nên có thể gặp mục tiêu xa hơn trước mục tiêu gần, dẫn đến lời giải không tối ưu.

**Kết luận:**

* **BFS và IDS** luôn tìm ra đường đi ngắn nhất đến một trong các mục tiêu (tối ưu).
* **DFS** không đảm bảo tối ưu, chỉ đảm bảo tìm ra một lời giải nếu tồn tại.

**Weighted A\* search:**

**Công thức f(n) = g(n) + W.h(n)**

Giả sử W = 10

Code:

def A\_star\_search\_add\_weight(maze):

    new\_maze = maze.copy().astype(str)

    start = mh.find\_pos(maze, what = "S")

    end =  mh.find\_pos(maze, what = "G")

    reached = set()

    frontier = []

    action\_effects = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    start\_node = Node(start, None,action\_effects, 0)

    frontier.append(start\_node)

    while len(frontier) > 0:

        # Weight = 2

        # Sort the frontier based on the f(n) = g(n) + 10\*h(n)

        frontier.sort(key=lambda node: node.cost + 10\*(abs(node.pos[0] - end[0]) + abs(node.pos[1] - end[1])))

        node = frontier.pop(0)  # Pop the node with the lowest f(n) value

        if node.pos == end:

            path = node.get\_path\_from\_root()

            for n in path:

                if new\_maze[n.pos] not in ["S", "G"]:

                    new\_maze[n.pos] = "P"

            return new\_maze

        if node.pos in reached:

            continue

        reached.add(node.pos)

        for action in action\_effects:

            new\_pos = (node.pos[0] + action[0], node.pos[1] + action[1])

            if mh.look(maze, new\_pos) != "X":

                if new\_maze[new\_pos] not in ["S", "G"]:

                    new\_maze[new\_pos] = "."

                new\_node = Node(new\_pos, node, action, node.cost + 1)

                frontier.append(new\_node)

    return new\_maze

with open("large\_maze.txt", "r") as f:

    maze\_str\_medium = f.read()

medium\_maze = mh.parse\_maze(maze\_str\_medium)

new = A\_star\_search\_add\_weight(medium\_maze)

mh.show\_maze(new)

new = A\_star\_search(medium\_maze)

mh.show\_maze(new)

Kết quả :

So sánh giữa 2 trường hợp không có trọng số và có trọng số

A screenshot of a maze

AI-generated content may be incorrect.

Kết quả :

Trường hợp đầu W = 10 ta tập frontier ít được thêm các ô vào hơn dẫn đến tiết kiệm bộ nhớ hơn

**Câu hỏi**: Điều gì sẽ xảy ra nếu agent không biết trước bản đồ mê cung? Điều này có nghĩa là tác nhân phải đối mặt với một môi trường chưa biết, nơi nó không biết hàm chuyển tiếp. Vậy môi trường sẽ trông như thế nào (mô tả PEAS)? Bạn sẽ triển khai một agent hợp lý để giải quyết mê cung như thế nào? Nếu agent vẫn có một thiết bị GPS để thông báo khoảng cách đến mục tiêu thì sao?

Nếu tác nhân không biết trước bản đồ mê cung, môi trường trở thành **không xác định** và **không quan sát đầy đủ**. Khi đó:

**Mô tả môi trường theo PEAS:**

* **Performance measure (P):**  
  Đến được mục tiêu càng nhanh càng tốt, số bước di chuyển ít nhất, tránh va vào tường.
* **Environment (E):**  
  Mê cung không biết trước, chỉ biết trạng thái hiện tại và các ô lân cận (có thể chỉ biết khi di chuyển đến).
* **Actuators (A):**  
  Di chuyển lên/xuống/trái/phải.
* **Sensors (S):**  
  Cảm biến phát hiện tường, vị trí hiện tại, có thể biết khoảng cách đến mục tiêu nếu có GPS.

**Cách xây dựng tác nhân hợp lý:**

* **Khám phá và xây dựng bản đồ:**  
  Tác nhân cần vừa di chuyển vừa khám phá, ghi nhớ các ô đã đi qua và các bức tường phát hiện được.
* **Chiến lược tìm kiếm:**
  + Có thể dùng thuật toán như DFS/BFS/A\* trên bản đồ tạm thời đã khám phá.
  + Nếu gặp ngõ cụt, quay lại và thử hướng khác (giống giải mê cung thực tế).
  + Nếu có GPS, dùng khoảng cách đến mục tiêu làm heuristic cho A\* để ưu tiên hướng đi gần mục tiêu.
* **Cập nhật tri thức:**  
  Sau mỗi bước, cập nhật bản đồ nội bộ và điều chỉnh kế hoạch nếu phát hiện đường đi bị chặn.

**Nếu có GPS:**

* **Lợi ích:**  
  GPS cung cấp khoảng cách đến mục tiêu, giúp tác nhân định hướng tốt hơn (ưu tiên các hướng giảm khoảng cách).
* **Áp dụng:**  
  Kết hợp khám phá với A\* heuristic dựa trên GPS để tìm đường hiệu quả hơn, dù chưa biết toàn bộ mê cung.

1. Robot vaccum

**Task 1: Triển khai môi trường mô phỏng**

**- Mô tả cài đặt:**

* Sử dụng mảng 2 chiều để lưu trạng thái sạch/bẩn của từng ô giúp dễ kiểm tra và cập nhật.
* Agent bắt đầu ở vị trí ngẫu nhiên để mô phỏng thực tế.
* Cảm biến va chạm (bumpers) giúp agent không đi ra ngoài phòng.
* Vòng lặp kiểm tra điều kiện dừng khi phòng đã sạch hoặc hết bước để đảm bảo hiệu quả.

**- Code:**

Ảnh có chứa văn bản, đồ điện tử, màn hình, ảnh chụp màn hình

Nội dung do AI tạo ra có thể không chính xác.

**- Giải thích:**

* **room = np.random.rand(n, n) < p:** Tạo một ma trận kích thước n x n, mỗi ô có xác suất p là bẩn (True), còn lại là sạch (False).
* **x, y = random.randint(0, n-1), random.randint(0, n-1)**: Đặt robot vào vị trí ngẫu nhiên trong phòng.
* **actions\_taken = 0:** Biến đếm số hành động đã thực hiện.
* **Vòng lặp for step in range(max\_steps):** Mỗi bước, robot sẽ thực hiện một hành động cho đến khi phòng sạch hoặc hết số bước cho phép.
* **bumpers:** Tạo cảm biến va chạm cho 4 hướng. Nếu robot ở sát mép phòng thì cảm biến trả về True (không đi tiếp được).
* **dirty = room[y, x]:** Kiểm tra ô hiện tại có bẩn không.
* action = agent\_function(bumpers, dirty): Gọi hàm agent để quyết định hành động dựa trên cảm biến.
* **actions\_taken += 1:** Tăng số lượng hành động.
* **Nếu action == "suck":** Robot làm sạch ô hiện tại.
* **Nếu robot chọn di chuyển (north/south/west/east) và không bị cản**, cập nhật vị trí tương ứng.
* **if not room.any():** Nếu tất cả các ô đều sạch, kết thúc vòng lặp.
* **return actions\_taken:** Trả về tổng số hành động đã thực hiện để đánh giá hiệu năng agent.

**- Tổng kết :**

Agent này giúp mô phỏng quá trình robot hút bụi hoạt động trong phòng, cung cấp cảm biến, xử lý hành động và đo lường hiệu quả làm sạch của agent, đảm bảo tính khách quan khi đánh giá các chiến lược khác nhau.

**- Minh chứng môi trường hoạt động với Agent ngẫu nhiên đơn giản:**

Ảnh có chứa văn bản, ảnh chụp màn hình, danh thiếp, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

**- Kết quả:**

Ảnh có chứa văn bản, thực đơn, sách, ảnh chụp màn hình

Nội dung do AI tạo ra có thể không chính xác.

**Task 2: Agent phản xạ đơn giản**

**- Mô tả cài đặt:**

* Agent ưu tiên làm sạch nếu ô hiện tại bẩn, vì mục tiêu là làm sạch phòng nhanh nhất.
* Nếu không bẩn, agent chọn hướng bất kỳ không bị cản để di chuyển, tránh lặp lại hành động vô ích.

**- Code:**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

**- Giải thích:**

* **if dirty: return "suck":** Nếu cảm biến báo ô hiện tại đang bẩn (dirty == True), agent sẽ ưu tiên thực hiện hành động hút bụi ("suck") để làm sạch ngay lập tức. Đây là hành vi hợp lý vì mục tiêu chính là làm sạch phòng càng nhanh càng tốt.
* **for direction in ["north", "east", "south", "west"]: ...** :Nếu ô hiện tại đã sạch, agent sẽ kiểm tra lần lượt các hướng di chuyển: bắc, đông, nam, tây.
* **if not bumpers[direction]: return direction**: Với mỗi hướng, agent kiểm tra cảm biến va chạm (bumpers). Nếu hướng đó không bị cản (giá trị là False), agent sẽ chọn di chuyển theo hướng đó. Điều này giúp agent tránh va vào tường và luôn ưu tiên di chuyển sang ô mới.
* **return "suck" # fallback**: Nếu tất cả các hướng đều bị cản (agent bị kẹt ở góc), agent sẽ thực hiện lại hành động hút bụi. Đây là phương án dự phòng để đảm bảo agent luôn thực hiện một hành động hợp lệ.

**- Tổng kết:**

Hàm này mô phỏng một agent phản xạ đơn giản:

Luôn làm sạch nếu ô hiện tại bẩn.

Nếu ô sạch, di chuyển sang hướng bất kỳ không bị cản.

Nếu bị kẹt, tiếp tục hút bụi (dù ô đã sạch).

Nhược điểm duy nhất là: chưa tối ưu vì không ghi nhớ trạng thái phòng hay vị trí đã đi qua, chỉ hoạt động với môi trường đơn giản

**- Minh chứng agent phản xạ đơn giản hoạt động với môi trường:**

Ảnh có chứa văn bản, ảnh chụp màn hình, danh thiếp, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

**- Kết quả:**

Ảnh có chứa văn bản, thực đơn, ảnh chụp màn hình, sách

Nội dung do AI tạo ra có thể không chính xác.

**Task 3: Agent phản xạ dựa trên mô hình**

**- Mô tả cài đặt:**

Agent dựa trên mô hình sẽ lưu lại vị trí hiện tại của mình và trạng thái sạch/bẩn của từng ô trong phòng. Agent sẽ di chuyển theo một quy tắc nhất định (ví dụ: đi theo từng hàng, zig-zag) để đảm bảo quét sạch toàn bộ phòng. Nhờ việc ghi nhớ các ô đã đi qua và đã làm sạch, agent này có thể tối ưu số bước di chuyển và tránh lặp lại các hành động không cần thiết.

**- Code:**

Ảnh có chứa văn bản, đồ điện tử, màn hình, ảnh chụp màn hình

Nội dung do AI tạo ra có thể không chính xác.

**- Giải thích:**

* **\_\_init\_\_(self, n):** Khởi tạo agent với kích thước phòng n x n.
  + **self.x, self.y:** Vị trí hiện tại của agent (bắt đầu ở góc trên trái).
  + **self.direction:** Hướng di chuyển hiện tại, khởi đầu là "east" (đi sang phải).
  + **self.visited:** Tập hợp các ô đã đi qua, giúp agent ghi nhớ lịch sử di chuyển.
* **agent\_function(self, bumpers, dirty):** Hàm quyết định hành động dựa trên cảm biến và trạng thái agent.
  + **pos = (self.x, self.y):** Lưu vị trí hiện tại vào tập visited.
  + **Nếu ô hiện tại bẩn (dirty == True)**, agent sẽ hút bụi ("suck").
  + **Nếu không bẩn,** agent di chuyển theo chiến lược zig-zag:
    - **Nếu đang đi sang phải (direction == "east"):**
      * **Nếu chưa bị cản phải (not bumpers["east"])**, di chuyển sang phải.
      * **Nếu bị cản phải nhưng chưa bị cản dưới (not bumpers["south"])**, di chuyển xuống dưới và đổi hướng sang trái.
    - **Nếu đang đi sang trái (direction == "west")**:
      * **Nếu chưa bị cản trái (not bumpers["west"])**, di chuyển sang trái.
      * **Nếu bị cản trái nhưng chưa bị cản dưới (not bumpers["south"])**, di chuyển xuống dưới và đổi hướng sang phải.
  + **Nếu bị kẹt (mọi hướng đều bị cản),** agent thử di chuyển theo bất kỳ hướng nào không bị cản (ưu tiên theo thứ tự north, east, south, west).
  + **Nếu vẫn không di chuyển được**, agent thực hiện hút bụi ("suck") như phương án dự phòng.

**- Ý nghĩa chiến lược zig-zag:**

Agent sẽ đi hết một hàng, xuống hàng tiếp theo, rồi đi ngược lại, lặp lại cho đến khi hết phòng.

Nhờ ghi nhớ vị trí và hướng, agent tránh lặp lại các ô đã đi qua, tối ưu hóa số bước di chuyển.

**- Tổng kết:**

Agent này thông minh hơn agent phản xạ đơn giản vì có khả năng ghi nhớ và lập kế hoạch di chuyển.

Phù hợp với phòng lớn, hình dạng phức tạp, hoặc có vật cản.

Đảm bảo mọi ô đều được kiểm tra và làm sạch, giảm thiểu số bước thừa.

**- Minh chứng Agent dựa theo mô hình hoạt động với môi trường:**

Ảnh có chứa văn bản, danh thiếp, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

**- Kết quả:**

Ảnh có chứa văn bản, thực đơn, ảnh chụp màn hình, sách

Nội dung do AI tạo ra có thể không chính xác.

**Task 4: So sánh hiệu năng các agent**

**- Mô tả:**

* Sử dụng pandas và matplotlib để trực quan hóa kết quả, giúp dễ dàng nhận xét và so sánh.
* Chạy nhiều lần (100 lần) để lấy kết quả trung bình, giảm ảnh hưởng của ngẫu nhiên.

**- Code:**

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, Hệ điều hành

Nội dung do AI tạo ra có thể không chính xác.

**- Giải thích:**

**1. Hàm evaluate\_agent**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, danh thiếp

Nội dung do AI tạo ra có thể không chính xác.

* **Mục đích:**Chạy thử nghiệm agent nhiều lần (mặc định 100 lần) trên phòng kích thước n x n, mỗi lần khởi tạo phòng và vị trí agent ngẫu nhiên.
* **Kết quả:**Trả về giá trị trung bình số hành động cần thiết để làm sạch phòng (hiệu năng agent).

**2. Khởi tạo các biến**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

**sizes:**Danh sách các kích thước phòng cần đánh giá (5x5, 10x10, 100x100).

results: Tạo dictionary để lưu kết quả cho từng loại agent.

**3. Vòng lặp đánh giá từng agent trên từng kích thước phòng**

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, đa phương tiện

Nội dung do AI tạo ra có thể không chính xác.

* **Randomized agent:** Đánh giá agent chọn hành động ngẫu nhiên.
* **Simple reflex agent:** Đánh giá agent phản xạ đơn giản (ưu tiên hút bụi, nếu sạch thì di chuyển).
* **Model-based agent:** Đánh giá agent dựa trên mô hình (ghi nhớ vị trí, di chuyển zig-zag).

**4. Tạo bảng kết quả bằng pandas**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

* **Tạo DataFrame:** Biến kết quả thành bảng, mỗi dòng là một kích thước phòng, mỗi cột là một loại agent.
* **In bảng:** Hiển thị số hành động trung bình của từng agent trên từng kích thước phòng.

**- Kết quả:**

Ảnh có chứa văn bản, Phông chữ, ảnh chụp màn hình, số

Nội dung do AI tạo ra có thể không chính xác.

* Agent dựa trên mô hình (Model-Based) thường hiệu quả nhất, đặc biệt khi kích thước phòng tăng lên.
* Agent ngẫu nhiên (Randomized) kém hiệu quả nhất do không có chiến lược rõ ràng.
* Agent phản xạ đơn giản (Simple Reflex) tốt hơn ngẫu nhiên nhưng vẫn có thể lãng phí hành động.

**Nhận xét:** Khi phòng càng lớn, sự khác biệt về hiệu quả giữa agent dựa trên mô hình và các loại agent khác càng rõ rệt, cho thấy vai trò của việc lập kế hoạch và ghi nhớ trạng thái.

- **Ngoài ra,** còn có đồ thị biểu diễn:

**Code:**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

Ảnh có chứa văn bản, hàng, Sơ đồ, ảnh chụp màn hình

Nội dung do AI tạo ra có thể không chính xác.

**Task 5: Độ bền vững**

**-Giải thích về độ bền vững của các agent**

**1. Nếu phòng có kích thước không xác định:**

* Agent ngẫu nhiên và phản xạ đơn giản vẫn hoạt động nhưng hiệu quả thấp, dễ bỏ sót hoặc lặp lại các bước.
* Agent dựa trên mô hình có thể khám phá phòng, ghi nhớ các vị trí đã đi qua, xây dựng bản đồ dần dần nên thích nghi tốt hơn.

**2. Nếu khu vực làm sạch có hình dạng bất thường:**

* Agent ngẫu nhiên và phản xạ đơn giản dễ bỏ sót các khu vực xa hoặc lãng phí hành động.
* Agent dựa trên mô hình nếu ghi nhớ các ô đã đi qua sẽ thích nghi tốt, đảm bảo làm sạch toàn bộ khu vực.

**3. Nếu phòng có vật cản:**

* Tất cả agent cần xử lý cảm biến va chạm. Agent ngẫu nhiên và phản xạ đơn giản có thể bị kẹt hoặc va chạm lặp lại.
* Agent dựa trên mô hình có thể ghi nhớ vị trí vật cản để tránh lặp lại va chạm, di chuyển tối ưu hơn.

**4. Nếu cảm biến bụi không hoàn hảo (10% sai lệch):**

* Agent có thể bỏ sót ô bẩn hoặc làm sạch ô đã sạch. Agent ngẫu nhiên và phản xạ đơn giản không có cơ chế kiểm tra lại.
* Agent dựa trên mô hình có thể kiểm tra lại các ô nghi ngờ nhiều lần để giảm số ô còn bẩn.

**5. Nếu cảm biến va chạm không hoàn hảo (10% sai lệch):**

* Agent có thể thử di chuyển vào vị trí không hợp lệ, dễ bị kẹt hoặc lãng phí hành động.
* Agent dựa trên mô hình có thể học từ các lần di chuyển thất bại, ghi nhớ vị trí thực tế của tường/vật cản để tránh lặp lại lỗi.

**Advanced: Mô phỏng cảm biến bụi không hoàn hảo**

**Code:**

Ảnh có chứa văn bản, đồ điện tử, ảnh chụp màn hình, máy tính

Nội dung do AI tạo ra có thể không chính xác.

**Giải thích:**

* Mỗi lần agent kiểm tra cảm biến bụi, có 10% khả năng nhận kết quả sai (nếu ô đang bẩn thì báo sạch, nếu sạch thì báo bẩn).
* Agent thực hiện hành động dựa trên cảm biến này, có thể bỏ sót ô bẩn hoặc làm sạch ô đã sạch.
* Hàm trả về số hành động đã thực hiện và số ô còn bẩn sau khi kết thúc (uncleaned squares).

**Ý nghĩa:**

* Kết quả cho thấy khi cảm biến không hoàn hảo, số ô còn bẩn có thể tăng lên nếu agent không có chiến lược kiểm tra lại các ô nghi ngờ.
* Agent dựa trên mô hình có thể cải tiến bằng cách kiểm tra lại các ô đã đi qua nhiều lần để giảm số ô còn bẩn, còn agent đơn giản sẽ dễ bỏ sót hoặc lãng phí hành động.

**More Advanced Implementation**

**Ý tưởng nâng cao**

**1. Vật cản:** Có thể thêm một ma trận boolean để đánh dấu các ô là vật cản. Agent cần kiểm tra cảm biến va chạm và ghi nhớ vị trí các vật cản để tránh lặp lại va chạm.

**2. Agent khám phá phòng không biết kích thước:** Agent sẽ di chuyển theo một quy tắc (ví dụ: luôn đi đến ô chưa được kiểm tra gần nhất, giống như tìm kiếm theo chiều sâu). Agent cần lưu lại các ô đã đi qua và xây dựng bản đồ phòng dần dần.

**3. Agent dựa trên utility:** Mỗi ô có xác suất bị bẩn lại, agent cần học các xác suất này và ưu tiên làm sạch các ô có xác suất cao. Utility của trạng thái là số ô sạch hiện tại. Agent sẽ tối ưu hóa utility trong một số bước giới hạn.

**- Code:**

Ảnh có chứa văn bản, ảnh chụp màn hình, máy tính, phần mềm

Nội dung do AI tạo ra có thể không chính xác.

**Giải thích:**

* Mỗi ô trong phòng có xác suất obstacle\_p trở thành vật cản, agent không thể đi qua các ô này.
* Khi agent kiểm tra cảm biến va chạm (bumper), nếu hướng di chuyển gặp tường hoặc vật cản thì cảm biến báo True, agent không thể di chuyển theo hướng đó.
* Agent thực hiện hành động như bình thường, nhưng sẽ bị hạn chế bởi các vật cản, có thể bị kẹt hoặc phải đi vòng.
* Hàm trả về số hành động đã thực hiện để làm sạch phòng.

**Ý nghĩa:**

* Môi trường có vật cản làm tăng độ khó cho agent, đòi hỏi agent phải có chiến lược ghi nhớ vị trí vật cản để tránh lặp lại va chạm và tối ưu hóa đường đi.
* Agent đơn giản dễ bị kẹt hoặc lãng phí hành động, agent dựa trên mô hình sẽ hiệu quả hơn nếu biết ghi nhớ và tránh các vị trí bị cản.