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 Văn Anh Tú – 3122410445

Trương Hữu Nghĩa - 3122410263

Đỗ Khôi Nguyên - 3122410266

Lai Tấn Tài - 3122410366

**TP. HCM THÁNG 09/2025**

**MỤC LỤC**

[1. Explore Heuristic 3](#_Toc210156400)

[2. Lunar lander 7](#_Toc210156401)

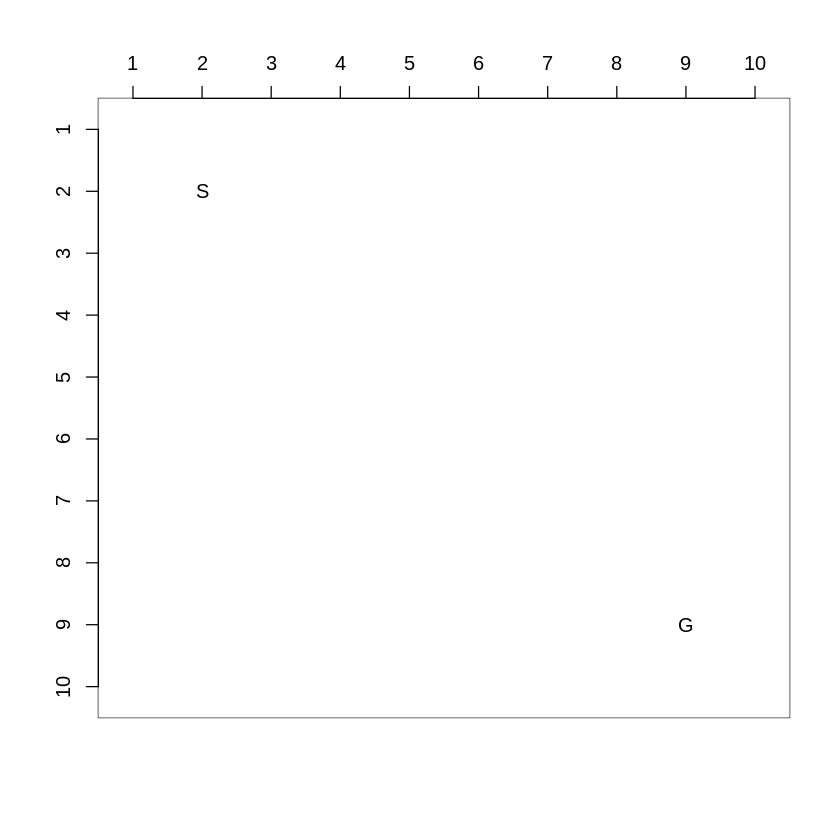
[3. Solving a Maze Using a Goal-based Agent 10](#_Toc210156402)

[4. Robot vaccum 32](#_Toc210156403)

1. Explore Heuristic

**Yêu cầu của readme:** ở bài toán Mê cung tìm kiếm sự khác biệt của các phương pháp heuristic

**Ta có:** Mê cung 10x10 với điểm bắt đầu (S) và kết thúc (G) sau:



**Greedy best first search**

Trong bài toán này sẽ dùng độ đo Manhattan để tính toán h(n)

Công thức của độ đo manhattan:

Ở bài toán này ta bắt đầu ở vị trí (2,2)

1. **Bắt đầu** tại (2,2).

* h(2,2)=∣2−9∣+∣2−9∣=14

1. **Lựa chọn nút tiếp theo**:

* GBFS xem các ô láng giềng (2,3), (3,2).
* (2,3): h=∣2−9∣+∣3−9∣=13.
* (3,2): h=13
* Chọn một trong hai (ví dụ (2,3)).

1. **Tiếp tục**:

* Từ (2,3), mở rộng láng giềng → luôn chọn ô nào có hhh nhỏ hơn, tức là ô đi gần đến (9,9) nhất.

1. **Đường đi kết quả**:

* GBFS sẽ liên tục tiến dần về phía (9,9), gần như đi theo đường chéo từ (2,2) đến (9,9).
* Ví dụ đường đi: (2,2) → (3,3) → (4,4) → … → (9,9).

**A\* with manhattan heuristic**

Bài toán A\* là bài toán chọn nút có tổng

Với

* : chi phí từ Start đến n (tức là ví dụ bước 0 thì chi phí này =0 và sẽ tăng lên mỗi bước cho đến khi tới được Goal)
* : ước lượng chi phí từ n đến Goal

Áp dụng vào bài mê cung 10x10

**Bước 0 (Start)**

* S = (2,2).
* g(S) = 0.
* h(S) = 14
* f(S) = 0+14 =14.

**Bước 1**

* Láng giềng:
  + (3,2): g=1,h=13,f=14
  + (2,3): g=1,h=13,f=14
* Chọn một trong hai (vì cùng f=14).

**Bước 2**

* Giả sử mở (3,2).
* Láng giềng mới:
  + (4,2): g=2,h=12,f=14
  + (3,3): g=2,h=12,f=14
* Open chứa: (2,3)[f=14], (4,2)[f=14], (3,3)[f=14].

**Quan sát:**

* Tất cả các bước đầu tiên đều có **f=14**, A\* chưa rút ngắn được.

**Tiếp tục**

* Khi đi xa hơn, ggg tăng lên nhưng hhh giảm đi tương ứng → tổng f=g+h = **luôn giữ gần bằng 14** cho các ô trên đường tối ưu.
* Nhưng nếu ta đi vòng xa (không hướng về đích), thì g tăng mà h không giảm đủ → f tăng → bị loại.

**Đường đi mà A\* tìm được**

 Đường đi ngắn nhất (Manhattan distance từ (2,2) đến (9,9)):

(9−2)+(9−2)=14 bước

 Một lời giải tối ưu:

(2,2)→(2,3)→(2,4)→⋯→(2,9)→(3,9)→⋯→(9,9) hoặc đi theo cột trước rồi hàng sau (đều 14 bước).

**Weighted A\***

Công thức của Weight A\*

Trong Weighted A\*, ta điều chỉnh bằng cách nhân heuristic với hệ số

* : chi phí đã đi từ strart đến n
* : heuristic ước lượng còn lại
* : trọng số (weight)

Ý nghĩa của w

Nếu w = 1 => trở về A\* chuẩn

Nếu w > 1 => heuristic được “khuếch đại”, tức là thuật toán thiên về đi nhanh hơn là tối ưu

Kết quả:

* Tìm được lời giải nhanh hơn, vì ít mở rộng nút hơn
* Lời giải có thể không tối ưu, nhưng được đảm bảo là không quá tệ

**Minh họa trên mê cung 10×10 (S=(2,2), G=(9,9))**

* Dùng Manhattan heuristic.
* Nếu w =2 :

**Trường hợp cụ thể:**

* Start (2,2): g= 0,h = 14  ⟹  f =28
* (3,2): g = 1,h = 13  ⟹  f = 27 g = 1
* (2,3): g = 1,h = 13  ⟹  f =27 g = 1

So với A\* (nơi f đều = 14), Weighted A\* nghiêng mạnh về **giảm h** nhanh nhất → dễ ưu tiên hướng “đi chéo” như GBFS, nhưng vẫn giữ g trong công thức.

Kết quả:

* WA\* sẽ đến goal nhanh hơn A\*, vì mở ít nút hơn.
* Nhưng đường đi có thể **dài hơn 14 bước** (không còn tối ưu tuyệt đối).

**GBFS and A\* with Euclidean Distance for h(n)**

Công thức độ do euclidean

Nếu chi phí thực sự đi lại là Manhattan thì độ đó Euclidean không là một lựa chọn phù hợp cho bài toán

Ở A\* khi môi trường chỉ cho di chuyển 4 hướng có thể mở rộng nút sai => không đảm bảo tối ưu

Tuy nhiên nếu di chuyển tự do theo mọi góc thì euclidean lại là heuristic đúng và hoàn hảo

**Maximum Norm**

Đúng khi ta cho phép di chuyển theo cả 8 hướng (lên, xuống, trái, phải, và chéo).

Khi đó chi phí tối thiểu chính là số bước chéo (đi cùng lúc x và y) cộng với phần chênh lệch còn lại.

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: nếu không sử dụng reached set thì sẽ bị lặp vô hạn khi gặp vòng lặp trong mê cung

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:**

**BFS** cần lưu trữ toàn bộ frontier ở mỗi mức, nên tốn rất nhiều bộ nhớ khi d lớn.

**DFS** chỉ cần lưu đường đi hiện tại (stack), nên bộ nhớ tăng tuyến tính theo độ sâu, phù hợp với các bài toán có không gian trạng thái lớn.

**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?**

**Với GBFS:**

Độ phức tạp về thời gian của Greedy Best-First Search là O(b^m) trong trường hợp xấu nhất, trong đó b là số nhánh tối đa và m là độ sâu tối đa của cây tìm kiếm. Tuy nhiên, trong thực tế, Greedy Best-First Search thường nhanh hơn so với các thuật toán tìm kiếm khác vì nó tập trung vào việc khám phá các nút có giá trị heuristic thấp nhất trước.

Độ phức tạp về không gian của Greedy Best-First Search là O(b^m) trong trường hợp xấu nhất, vì nó cần lưu trữ tất cả các nút trong frontier. Tuy nhiên, trong thực tế, không gian sử dụng thường nhỏ hơn so với các thuật toán tìm kiếm khác vì nó không cần lưu trữ tất cả các nút đã thăm.

**Với A\*:**

Độ phức tạp về thời gian của A\* Search là O(b^d) trong trường hợp xấu nhất, trong đó b là số nhánh tối đa và d là độ sâu của cây tìm kiếm. Tuy nhiên, nếu hàm heuristic được chọn tốt (như Manhattan distance trong trường hợp này), A\* Search có thể hoạt động rất hiệu quả và gần như tuyến tính với độ sâu của giải pháp.

Độ phức tạp về không gian của A\* Search cũng là O(b^d) trong trường hợp xấu nhất, vì nó cần lưu trữ tất cả các nút trong frontier và các nút đã thăm. Tuy nhiên, tương tự như thời gian, nếu hàm heuristic được chọn tốt, không gian sử dụng có thể giảm đáng kể.

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.

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.

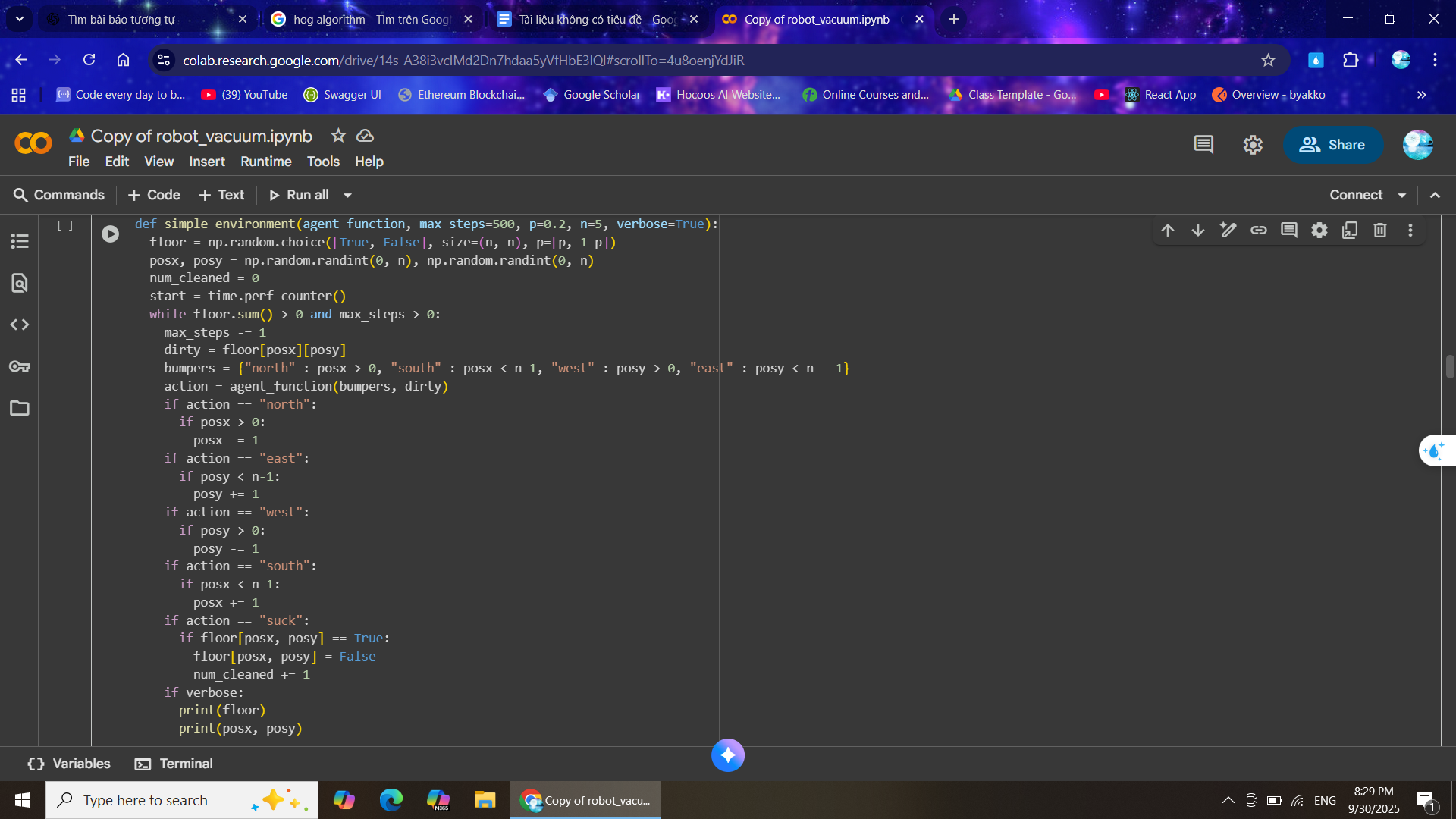
**Cách xây dựng tác nhân hợp lý:**

* **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.

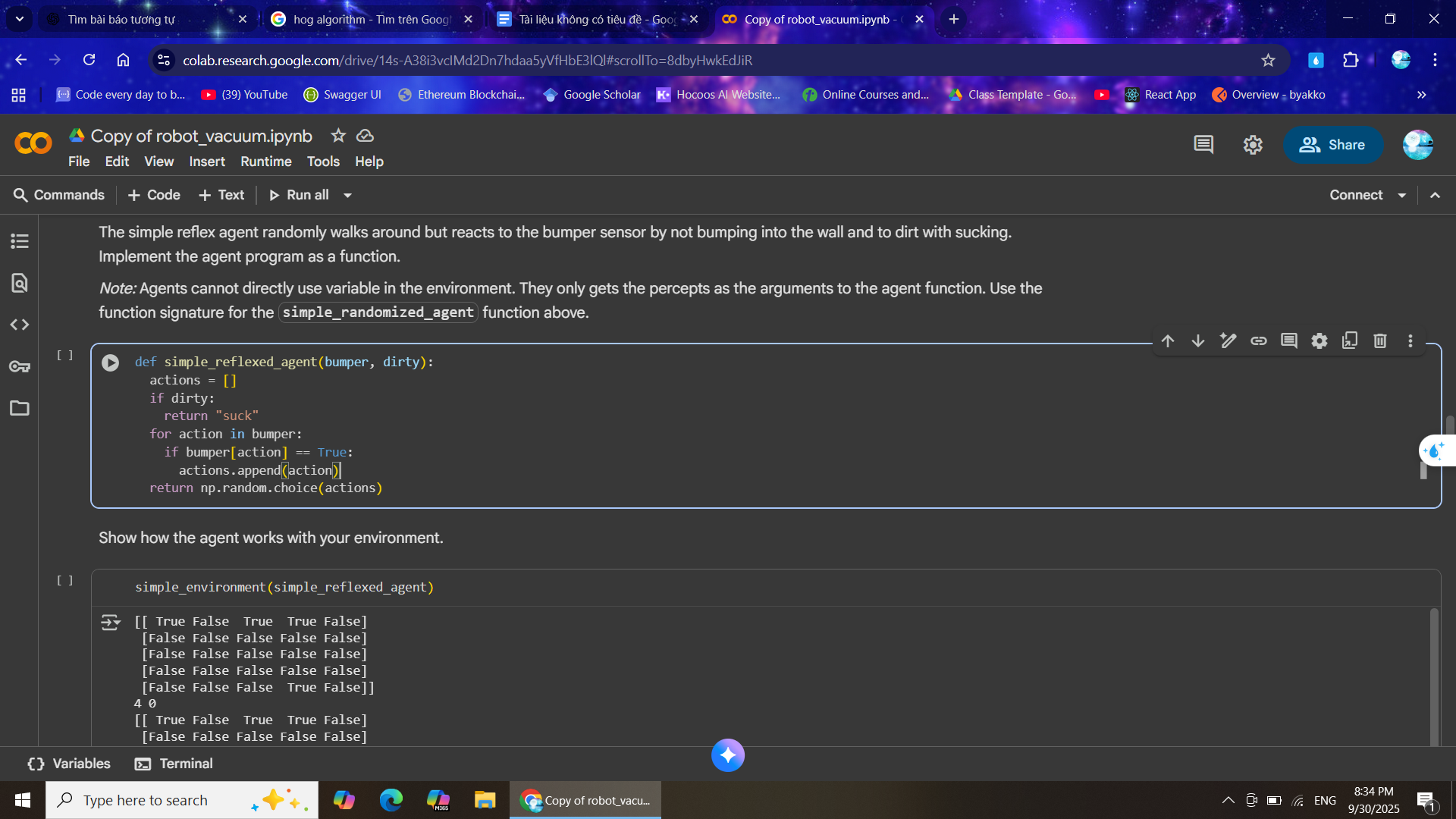
**Nếu có GPS:**

* **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



môi trường được xây dựng với đầu vào là hàm tác tử đơn giản và ngẫu nhiên thực hiện các thao tác, được sử dụng để cung cấp môi trường cho các agent như (simple randomized agent, simple reflex agent)



simple reflexed agent có khả năng nhận vào cảm biến cơ bản và đưa ra quyết định từ các cảm biến như hút bụi và không đâm vào tường.