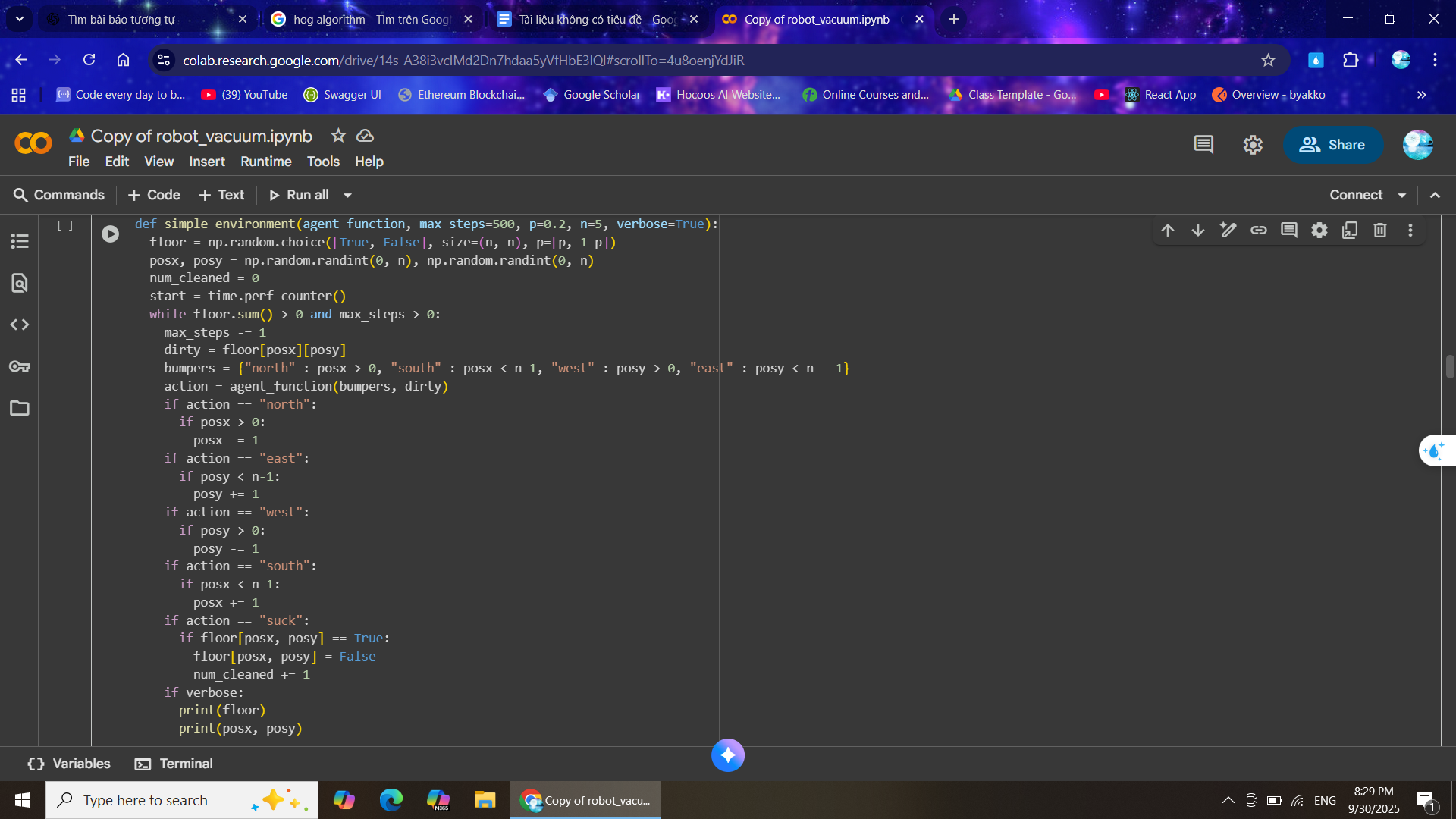
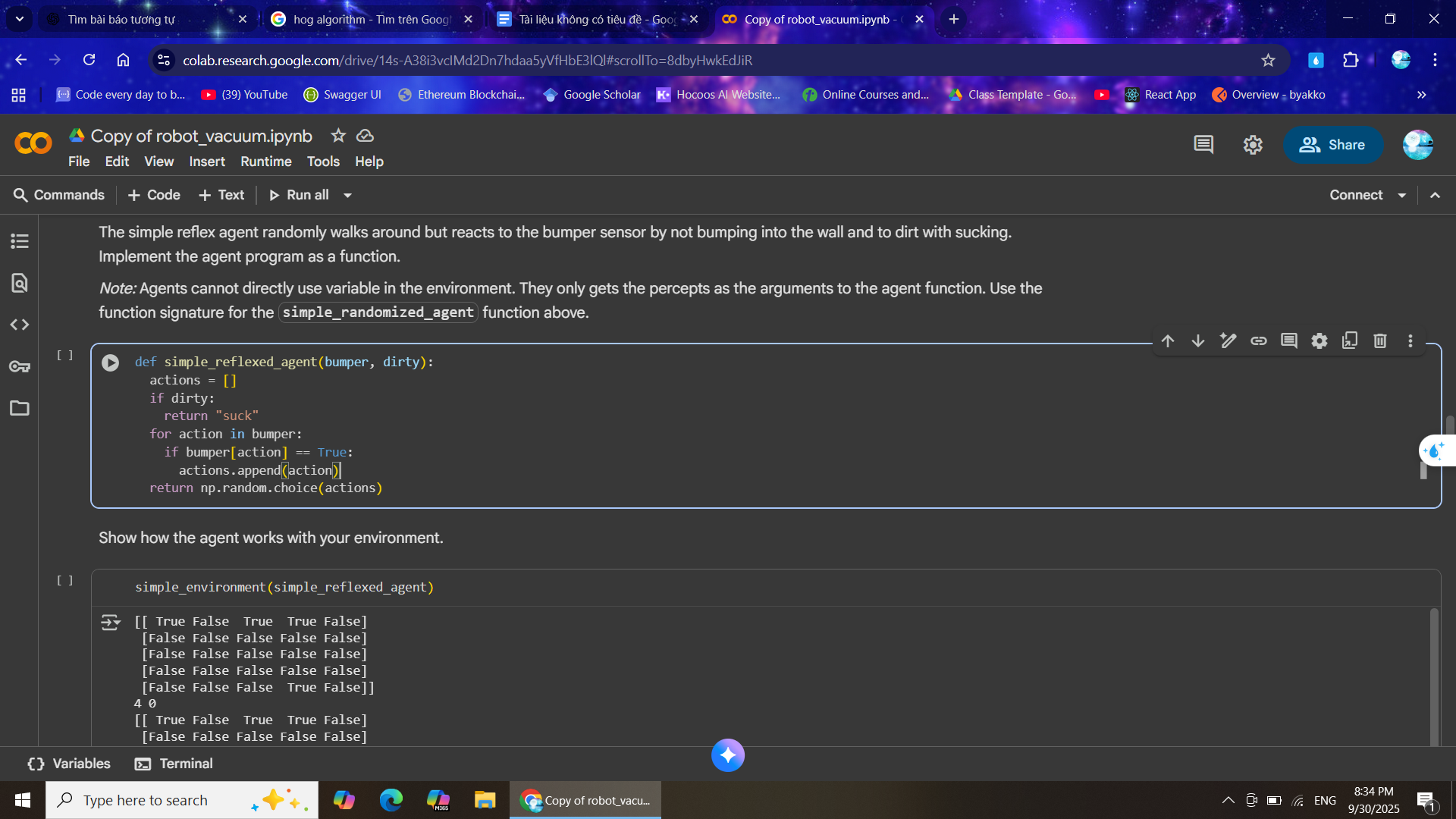
Họ và Tên: LAI TẤN TÀI

MSSV: 3122410366

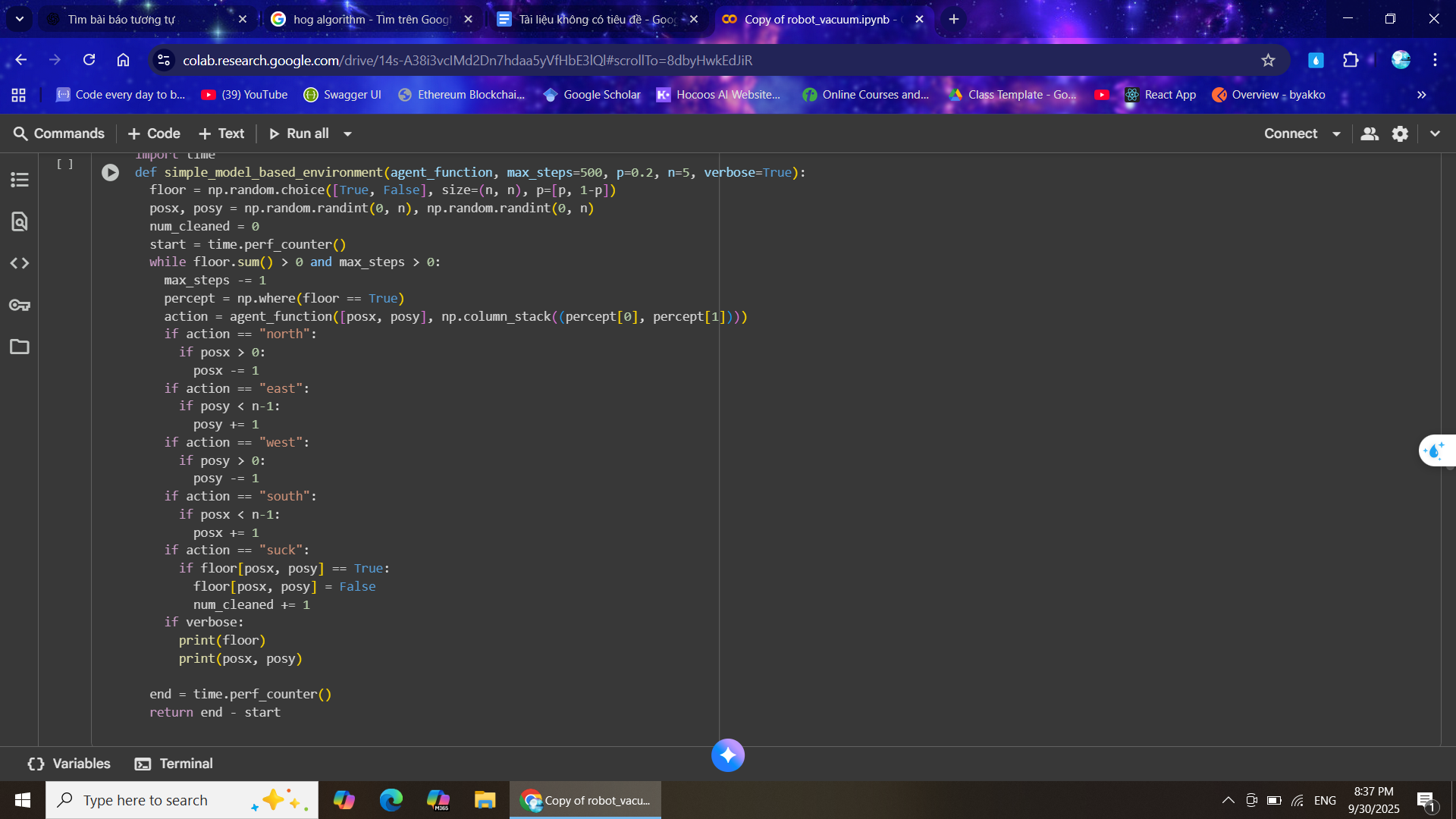
BÀI TẬP:



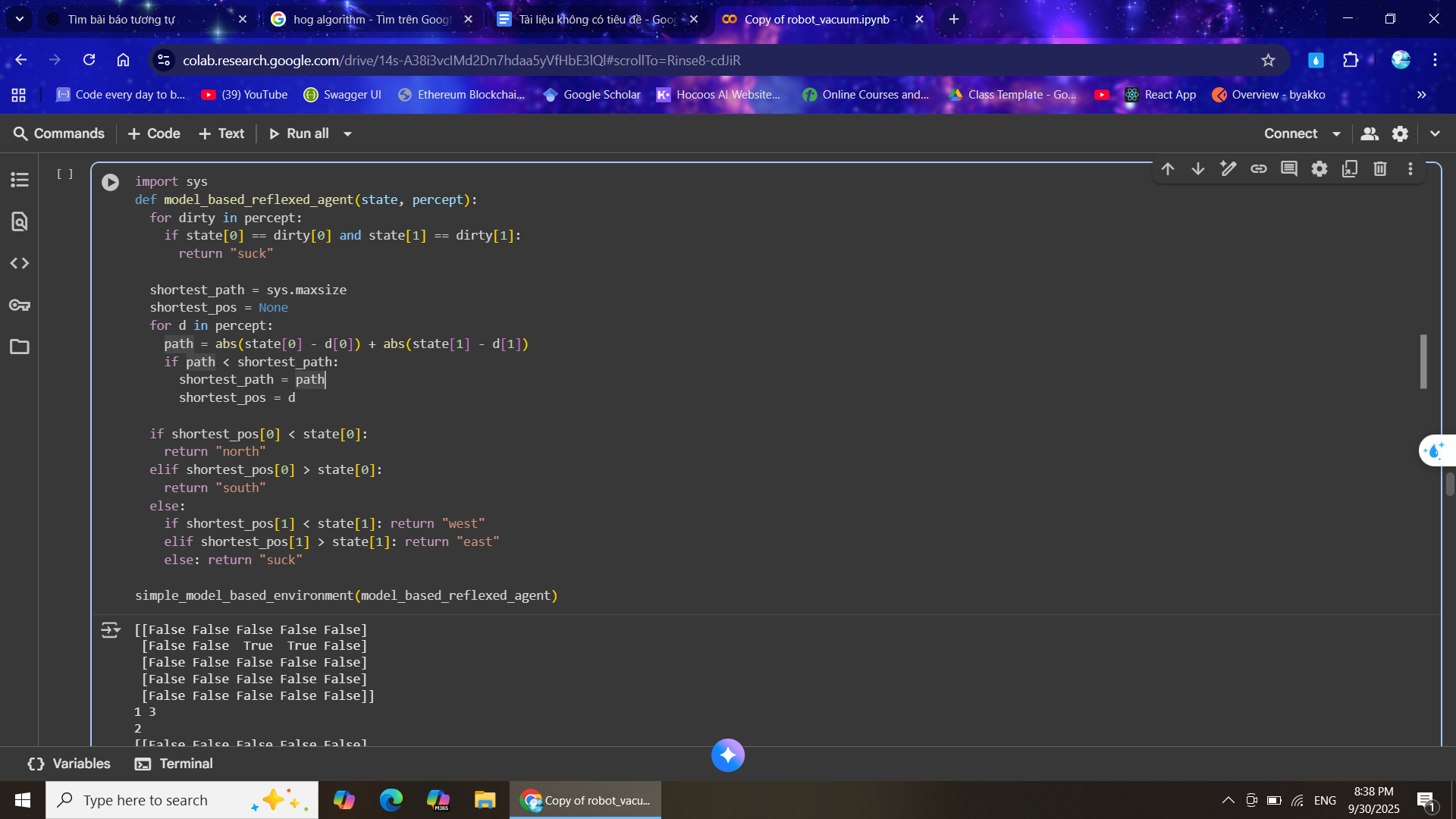
môi trường được xây dựng với đầu vào là hàm tác tử đơn giản và ngẫu nhiên thực hiện các thao tác, được sử dụng để cung cấp môi trường cho các agent như (simple randomized agent, simple reflex agent)



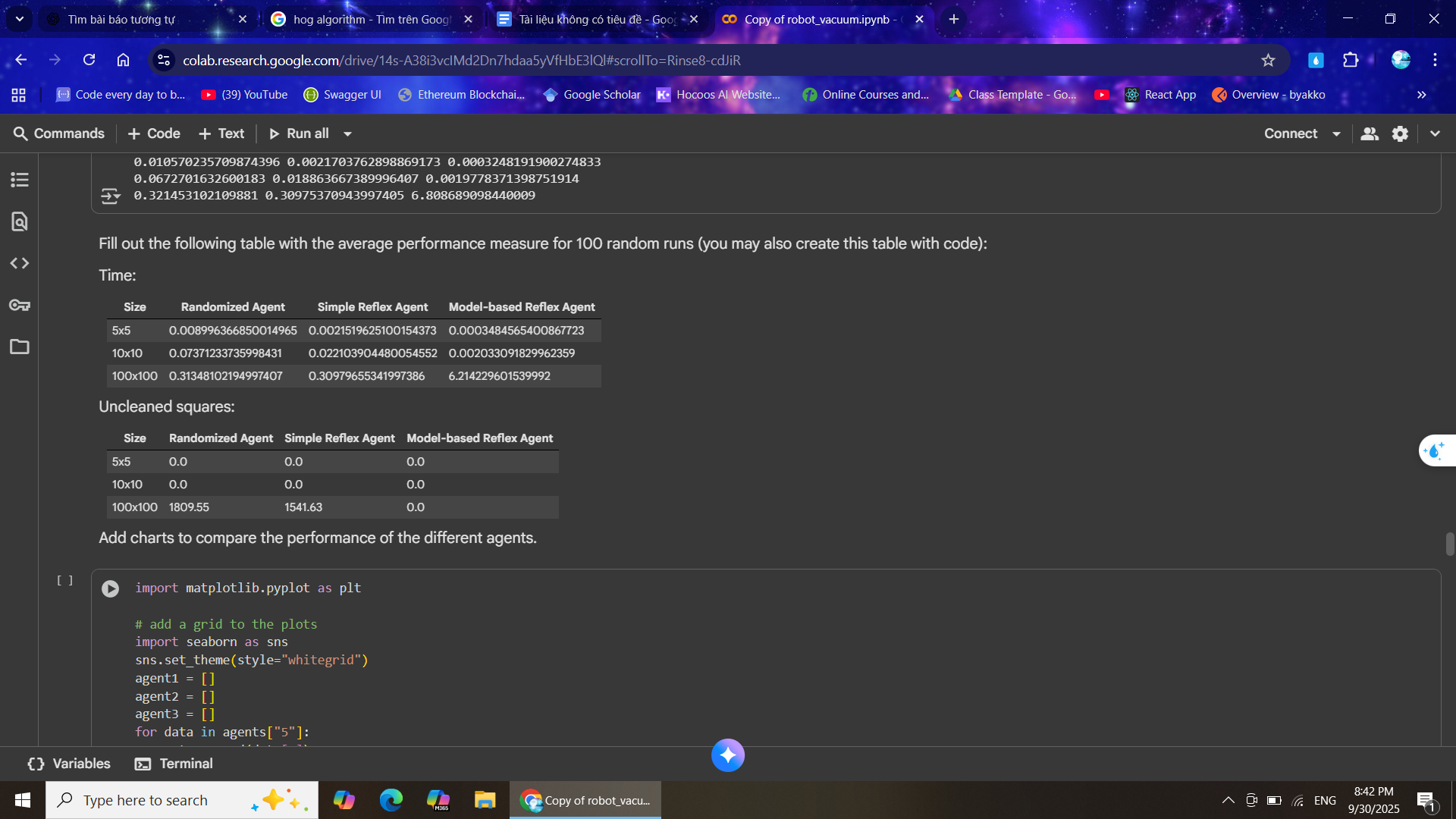
simple reflexed agent có khả năng nhận vào cảm biến cơ bản và đưa ra quyết định từ các cảm biến như hút bụi và không đâm vào tường.



Môi trường được xây dựng cho các hàm tác tử phức tạp hơn

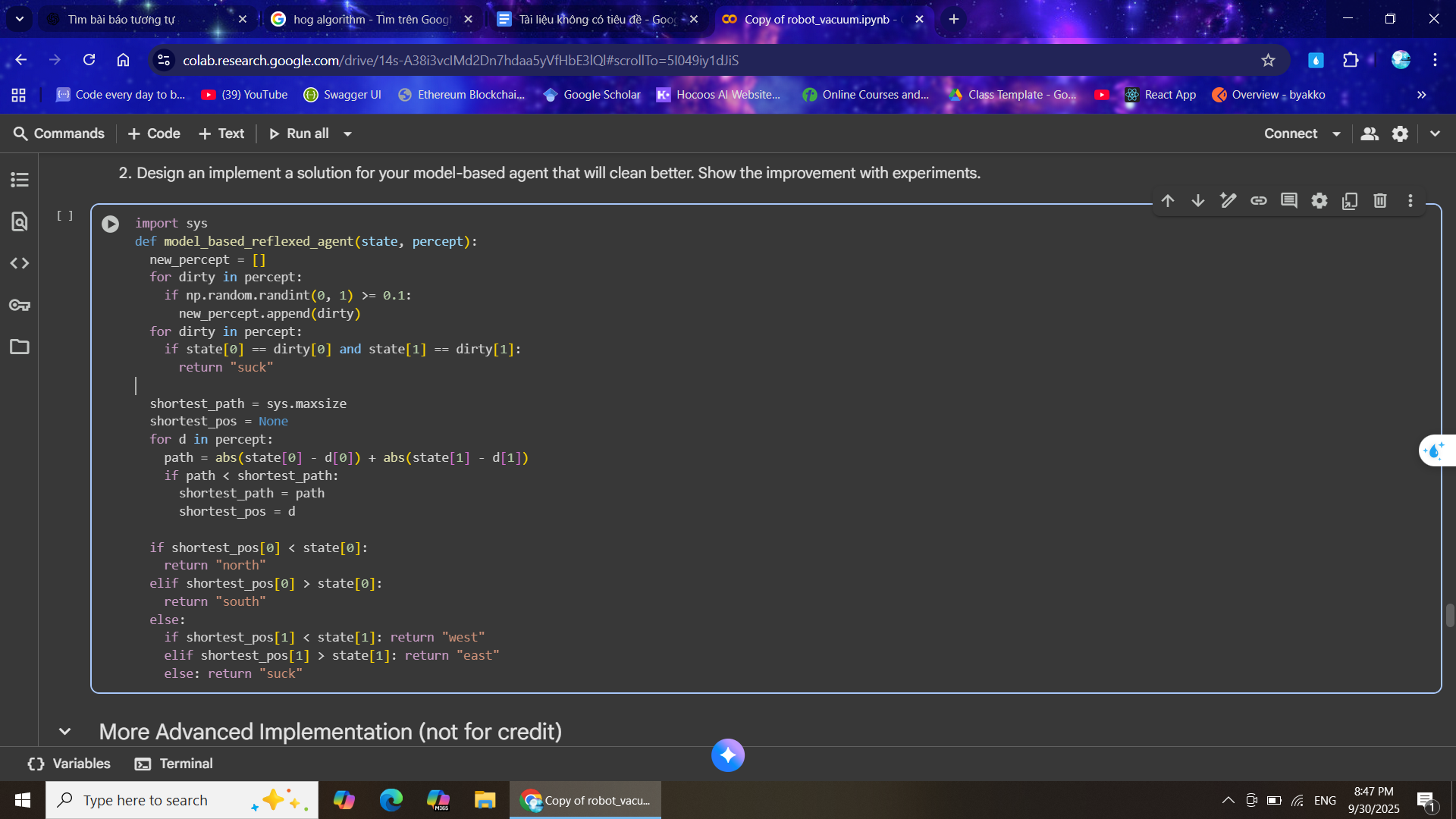


Model-based agent được xây dựng dựa trên việc áp dụng tính toán manhattan để tìm đường đi ngắn nhất đến vị trí bẩn, Mô hình này có khả năng tìm đường đi tối ưu nhất và có khả năng vượt qua sự sai lệch của sensor (do mỗi step thì mô hình đều cập nhật trạng thái và tính toán toàn bộ với vị trí bẩn) nhưng không giải quyết được vật cản cũng như là hình dạng phòng bị khuất.



Ở trên là thống kê của 3 mô hình, trong đó ta thấy được tuy các mô hình simple có khả năng đáp ứng được kích thước nhỏ khá tốt nhưng khi kích thước lớn thì loại mô hình này có vị trí bẩn chưa được dọn rất nhiều. Trong khi model-based giải quyết được hiệu quả các vấn đề này dù cần thời gian lớn hơn nhưng hiệu suất rất đáng kể.

BÀI TẬP NÂNG CAO:



Được thêm 10% nhiễu vào cảm biến trong đó mô hình có thể dễ dàng giải quyết được một cách dễ dàng thông qua tính toán manhattan và biết được trạng thái hiện tại.

import numpy as np

import time

def environment\_with\_obstacles(agent\_function, max\_steps=500, p=0.2, n=5, obstacle\_p=0.1, verbose=True):

floor = np.random.choice([True, False], size=(n, n), p=[p, 1-p])

obstacles = np.random.choice([True, False], size=(n, n), p=[obstacle\_p, 1-obstacle\_p])

posx, posy = np.random.randint(0, n), np.random.randint(0, n)

while obstacles[posx, posy]:

posx, posy = np.random.randint(0, n), np.random.randint(0, n)

num\_cleaned = 0

start = time.perf\_counter()

while floor.sum() > 0 and max\_steps > 0:

max\_steps -= 1

percept = np.where(floor == True)

action = agent\_function([posx, posy], np.column\_stack((percept[0], percept[1])), obstacles)

newx, newy = posx, posy

if action == "north" and posx > 0: newx -= 1

elif action == "south" and posx < n-1: newx += 1

elif action == "west" and posy > 0: newy -= 1

elif action == "east" and posy < n-1: newy += 1

if not obstacles[newx, newy]:

posx, posy = newx, newy

if action == "suck" and floor[posx, posy]:

floor[posx, posy] = False

num\_cleaned += 1

if verbose:

print("Floor:\n", floor.astype(int))

print("Obstacles:\n", obstacles.astype(int))

print("Agent:", posx, posy)

end = time.perf\_counter()

return num\_cleaned, end - start

def agent\_with\_obstacles(state, percept, obstacles):

for dirty in percept:

if state[0] == dirty[0] and state[1] == dirty[1]:

return "suck"

shortest\_path = sys.maxsize

target = None

for d in percept:

dist = abs(state[0] - d[0]) + abs(state[1] - d[1])

if dist < shortest\_path:

shortest\_path = dist

target = d

if not target:

return "suck"

if target[0] < state[0]:

if not obstacles[state[0]-1, state[1]]: return "north"

if target[0] > state[0]:

if not obstacles[state[0]+1, state[1]]: return "south"

if target[1] < state[1]:

if not obstacles[state[0], state[1]-1]: return "west"

if target[1] > state[1]:

if not obstacles[state[0], state[1]+1]: return "east"

return "suck"

def utility\_environment(agent\_function, n=5, max\_steps=100000, verbose=False):

dirty\_probs = np.random.rand(n, n) \* 0.1

floor = np.zeros((n, n), dtype=bool)

posx, posy = np.random.randint(0, n), np.random.randint(0, n)

total\_utility = 0

for t in range(max\_steps):

randoms = np.random.rand(n, n)

floor[randoms < dirty\_probs] = True

percept = np.where(floor == True)

action = agent\_function([posx, posy], np.column\_stack((percept[0], percept[1])), dirty\_probs)

if action == "north" and posx > 0: posx -= 1

elif action == "south" and posx < n-1: posx += 1

elif action == "west" and posy > 0: posy -= 1

elif action == "east" and posy < n-1: posy += 1

elif action == "suck":

floor[posx, posy] = False

total\_utility += (n\*n - floor.sum())

if verbose and t % 10000 == 0:

print(f"Step {t}, Utility so far: {total\_utility}")

return total\_utility / max\_steps

def utility\_based\_agent(state, percept, dirty\_probs):

for d in percept:

if state[0] == d[0] and state[1] == d[1]:

return "suck"

best = None

best\_prob = -1

for x in range(dirty\_probs.shape[0]):

for y in range(dirty\_probs.shape[1]):

if dirty\_probs[x, y] > best\_prob:

best\_prob = dirty\_probs[x, y]

best = (x, y)

if best[0] < state[0]: return "north"

if best[0] > state[0]: return "south"

if best[1] < state[1]: return "west"

if best[1] > state[1]: return "east"

return "suck"

Utility-based agent có khả năng giải quyết được về vật cản trong môi trường vừa có khả năng giải quyết được hình dạng của căn phòng khi kích thước không phải hình vuông.