Hàm KhoiTao\_NiemTin\_Non tạo ra ma trận niềm tin ban đầu cho agent

Ban đầu, các ô ( i, j) sẽ được gán giá trị từ 1 tới 10 ( max\_loai), tượng trung cho 10 loại con Pokemon. Tác nhân agent không biết ô nào chứa giá trị nào nên phải giả sử tất cả trường hợp đều có thể xảy ra

Nếu ô thực tế ma\_tran[i][j]== 0, tức là ô trống, niềm tin là set() rỗng

Nếu ô không trống, agent không biết loại nào nên gán set({1,2,3,4,5,6,7,8,9,10}), tất cả các loại có thể có

def KhoiTao\_NiemTin\_Non(ma\_tran, max\_loai=10):

    niemtin = {}

    tat\_ca\_loai = set(range(1, max\_loai + 1))

    for i in range(1, 11):

        for j in range(1, 11):

            if ma\_tran[i][j] == 0:

                niemtin[(i, j)] = set()

            else:

                niemtin[(i, j)] = set(tat\_ca\_loai)

    return niemtin

Tiếp theo, hàm Quan\_Sat cho agent quan sát xung quang ô (x,y) trong phạm vi , ta sẽ có một danh sách chứa các ô ( x\_new, y\_new) mà agent thay đổi trong agent map

Duyệt tất cả cặp ( dx, dy) trong khoảng [-2;2], ma trận vuông kích thước 5 x 5. Nếu giá trị đang có trong agent map khác giá trị thật trong ma\_tran thì cập nhật agent\_map bằng giá trị thật đó và thêm vào danh sách

def Quan\_Sat(self,agent\_map,  x, y, pham\_vi=2):

        thay\_doi = []

        for dx in range(-pham\_vi, pham\_vi + 1):

            for dy in range(-pham\_vi, pham\_vi + 1):

                x\_new, y\_new = x + dx, y + dy

                if 0 <= x\_new < len(self.ma\_tran) and 0 <= y\_new < len(self.ma\_tran[0]):

                    if agent\_map[x\_new][y\_new] != self.ma\_tran[x\_new][y\_new]:

                        agent\_map[x\_new][y\_new] = self.ma\_tran[x\_new][y\_new]

                        thay\_doi.append((x\_new, y\_new))

        return thay\_doi

Sau đó, ta sẽ cập nhật lại niềm tin của agent sau khi agent quan sát được

Agent\_map chứa các giá trị mà agent biết được sau khi quan sát

Danh\_sach\_quan\_sat là danh sách các ô vừa được cập nhật trong agent map

Hàm Cap\_Nhat\_Niem\_Tin\_Sau\_Quan\_Sat sẽ duyệt từng vị trí ô mà vừa thay đổi trong agent\_map, lấy giá trị agent biết được tại vị trí đó ( do hàm Quan\_Sat gán ở trên), ô rỗng (0), set niemtin thành set(), không rỗng thì gán niềm tin thành tập chứa đúng giá trị này, khi này agent biết chính xác giá trị nào ( Pokemon nào)

Dựa vào niềm tin của agent, hàm Doan\_Cap\_Tu\_Niem\_Tin sẽ tìm một giá trị v mà agent có ít nhất 2 ô có khả năng chứa v, tức là tạo thành cặp để ăn nhau

Cac\_gia\_tri là dict sẽ map giá trị v tới các vịt rí pos có thể xuất hiện theo niềm tin, duyệt từng (pos, tap) tring niemtin, với mỗi v trong tap, ( bỏ qua v==0, không phải pokemon), thêm pos vào cac\_gia\_tri [v]. Sau khi duyệt xong, các\_gia\_tri chứa toàn bộ vị trí mà agent đang tin có thể là v, lọc ra các v khả thi có nhiều hơn 2 vị trí, nếu danh sách này rỗng thì trả None, None, None. Nếu có, vhojn ngẫu nhiên một v trong cac\_v\_kha\_thi, từ cac\_gia\_tri[v] chọn 2 vị trí ngẫu nhiên p1,p2 khác nhau

Ví dụ:

niemtin = {

(1,1): {1, 2}, # ô (1,1) có thể là 1 hoặc 2

(1,2): {2, 3}, # ô (1,2) có thể là 2 hoặc 3

(2,1): {1}, # ô (2,1) chắc là 1

(2,2): set(), # ô (2,2) trống, set thành set()

(3,3): {4, 2} # ô (3,3) có thể là 4 hoặc 2

}

Ban đầu, cac\_gia\_tri = {}

Duyệt với pos = (1,1), tap = {1,2}

v = 1 , cac\_gia\_tri.setdefault(1,[]).append((1,1))  
ta có: cac\_gia\_tri = {1: [(1,1)]}

v = 2 , cac\_gia\_tri.setdefault(2,[]).append((1,1))

ta có: cac\_gia\_tri = {1: [(1,1)], 2: [(1,1)]}

Duyệt với pos = (1,2), tap = {2,3}

v = 2, append (1,2) , cac\_gia\_tri[2] = [(1,1), (1,2)]

v = 3 , append (1,2) , cac\_gia\_tri[3] = [(1,2)]

Duyệt với pos = (2,1), tap = {1}

v = 1, append (2,1) , cac\_gia\_tri[1] = [(1,1), (2,1)]

Duyệt với pos = (2,2), tap = set() , không lặp gì cả

Duyệt với pos = (3,3), tap = {4,2}

v = 4 , cac\_gia\_tri[4] = [(3,3)]

v = 2 , append (3,3) → cac\_gia\_tri[2] = [(1,1),(1,2),(3,3)]

Kết thúc for:

cac\_gia\_tri == {

1: [(1,1), (2,1)],

2: [(1,1), (1,2), (3,3)],

3: [(1,2)],

4: [(3,3)] }

Cac\_v\_kha\_thi = [1,2] ( vì 1 có 2 vị trí, 2 có 3 vị trí), giả sử chọn 2, pos\_list = cac\_gia\_tri[2] = [(1,1),(1,2),(3,3)], p1, p2 random, giả sử chọn (1,1) và (3,3), agent đoán 2 ô đó đều loại 2

Hàm Co\_The\_An\_BFS tìm một đường đi từ ô p1 đến ô p2 sao cho:agent chỉ đi qua những ô đã biết (không phải -1) và hoặc là ô trống (0) hoặc là chính ô đích (p2), số lần rẽ không vượt quá 2. Nếu tìm được đường thỏa điều kiện trả về danh sách các ô theo thứ tự (path). Nếu không tìm được trả None

p1 = (x1,y1) là ô bắt đầu, p2 = (x2,y2) là ô đích

dir = [(-1,0),(1,0),(0,-1),(0,1)] là danh sách 4 hướng idx: 0 lên, 1 xuống, 2 trái , 3 phải

idx là chỉ số hướng (0,1,2,3) dùng để so sánh với huong hiện tại để biết có rẽ không

queue (deque): hàng đợi cho BFS, lưu các trạng thái (x, y, huong, re):

x,y vị trí hiện tại,

huong = hướng tới ô hiện tại (là idx của lần bước vừa đi). Giá trị -1 ở trạng thái khởi tạo nghĩa là chưa có hướng trước đó (bắt đầu).

re = số lần rẽ đã thực hiện từ đầu đường đến trạng thái này

visited (set): chứa các trạng thái đã thăm, mình lưu cả (x,y,huong,re) để tránh lặp lại cùng vị trí với cùng hướng và cùng số rẽ

parent (dict): dùng để truy đường khi tìm được đích

Khởi tạo initial\_state = (x1, y1, -1, 0) (bắt đầu chưa có hướng, 0 lần rẽ), đánh dấu đã thăm

Trong vòng while: lấy trạng thái đầu (popleft), đặt tên (x,y,huong,re), nếu (x,y) là đích (x2,y2) và re <= 2 thì tìm thấy đường hợp lệ, truy lại đường từ đích về nguồn, sau đó reverse để theo thứ tự p1 tới p2, trả về path

Nếu chưa phải đích thig thử 4 hướng kế tiếp:

Tính toạ độ x\_new,y\_new, thỏa điều kiện không phải -1 ( đã biết ), là ô rỗng (0) hoặc ô đích

Tính new\_re , số lần rẽ nếu đi sang idx

nếu huong == -1 (chưa có hướng trước) thì không được tính là rẽ.

nếu huong == idx thì không rẽ, khác thì new\_re = re + 1.

Nếu new\_re <= 2 thì tạo new\_state = (x\_new, y\_new, idx, new\_re); nếu state này chưa thăm thì:

thêm new\_state vào visited và queue,

đặt parent[(x\_new,y\_new)] = (x,y) (để có thể dựng đường).

Nếu queue rỗng mà không tìm được đích , trả về None

Ví dụ:

Giả sử cho trạng thái ban đầu là: (4,(1,1),-1,0) và trạng thái đích (4,(1,2),None,None)

|  |  |  |
| --- | --- | --- |
| State | Queue | Visited |
|  | A1(4,(1,1),-1,0) | A1 |
| A1 | A2(0,(0,1), lên, 0 ), A3 (4,(1,2),phải,0), A4(0, (1,0),  trái, 0) | A1,A2,A3,A4 |
| A2 | A3, A4,A5(0,(0,0),trái,1),A6(0,(0,2),phải,1) | A1,A2,A3,A4,A5,A6 |
| A3  is GOAL |  |  |

Hàm cap\_Kha\_Nang\_An kiểm tra trong niềm tin có ít nhất tồn tại một cặp ô có khả năng cùng là một loại, ví dụ 2 ô đều là loại 3 thì trả về True, ngược lại False

counts là một dictionary tạm để đếm số ô trong niemtin mà có thể chứa giá trị v

Vòng lặp đầu:

Duyệt từng (pos, tap) trong niemtin.

Duyệt từng v trong tap

Bỏ qua v == 0 (0 được hiểu là ô trống).

counts[v] += 1 (tăng bộ đếm cho giá trị v).

Sau khi có counts, vòng lặp thứ hai kiểm tra: nếu có bất cứ counts[v] >= 2 thì trả True (vừa đủ 2 ô cùng loại v). Nếu không, trả False.

Ví dụ:

niemtin = {

(1,1): {1, 2}, # ô A có thể là 1 hoặc 2

(1,2): {2, 3}, # ô B có thể là 2 hoặc 3

(1,3): {4}, # ô C chắc chắn là 4

(1,4): set(), # ô D trống, đã biết là 0

(2,1): {1, 5}, # ô E có thể là 1 hoặc 5

Khởi tạo counts = {}.

Xét pos (1,1), tap = {1,2}:

v=1: counts[1] = counts.get(1,0)+1 → 1, counts = {1:1}

v=2: counts[2] = 1 nên counts = {1:1, 2:1}

1 xuất hiện 1 lần, 2 xuất hiện 1 lần

Xét pos (1,2), tap = {2,3}:

v=2: counts[2] = 2 nên counts = {1:1, 2:2}

v=3: counts[3] = 1 nên counts = {1:1, 2:2, 3:1}

1 xuất hiện 1 lần, 2 xuất hiện 2 lần. 3 xuất hiện 1 lần

Xét pos (1,3), tap = {4}:

v=4: counts[4] = 1 nên counts = {1:1,2:2,3:1,4:1}

1 xuất hiện 1 lần, 2 xuất hiện 2 lần. 3 xuất hiện 1 lần, 4 xuất hiện 1 lần

Xét pos (1,4), tap = set():

vòng for v in tap không chạy (không thay đổi counts)

Xét pos (2,1), tap = {1,5}:

v=1: counts[1] = 2 nên counts = {1:2,2:2,3:1,4:1}

v=5: counts[5] = 1 nên counts = {1:2,2:2,3:1,4:1,5:1}

Bây giờ counts có các giá trị >=2: counts[1]=2 và counts[2]=2.

Hàm sẽ thấy c=2 (>=2) cho v=1, trả về True, nghĩa là có khả năng tồn tại một cặp ô cùng loại (ở ví dụ trên: ô (1,1) & (2,1) có thể đều là 1; hoặc (1,1) & (1,2) có thể đều là 2).

Hàm KhoiTao\_NiemTin\_1\_P khởi tạo ma trận cho agent trong môi trường nhìn thấy một phần (agent chỉ thấy ô trong một ma trận vuông bán kính pham\_vi quanh vị trí bắt đầu).

Tạo agent\_map kích thước 12×12, khởi tất cả giá trị bằng -1 (agent chưa quan sát ô đó).

Khởi tạo niemtin rỗng

start là vị trí bắt đầu của agent, gán vào x,y

if abs(i - x) <= pham\_vi and abs(j - y) <= pham\_vi:

Điều kiện này kiểm tra ô (i,j) có nằm trong ma trận vuông kích thước pham\_vi quanh (x,y) không, cả tọa độ hàng và cột khác không quá pham\_vi. Kết quả vùng quan sát là ma trận vuông có kích thước 5 x5

Nếu ô nằm trong vùng quan sát

agent\_map[i][j] = ma\_tran[i][j]

if ma\_tran[i][j] == 0:

niemtin[(i, j)] = set()

else:

niemtin[(i, j)] = {ma\_tran[i][j]}

Ghi giá trị thật từ ma\_tran vào agent\_map (agent thấy ô đó).

Nếu giá trị bằng 0 (ô trống), thì niemtin[(i,j)] = set() (rỗng ).

Nếu ô có loại (1,..,10), set niemtin[(i,j)] = ô đó ( đã biết)

Nếu ô không nằm trong vùng quan sat, thì sat thành set() ( trống)

Ví dụ:

j=1 2 3 4 5 6 7 8 9 10

i=1 1 1 2 0 3 4 5 6 7 8

i=2 2 1 0 3 3 2 5 6 7 1

i=3 4 0 2 1 5 2 2 9 10 3

i=4 3 3 1 0 4 4 5 1 2 2

i=5 6 7 8 1 1 2 3 4 5 1

i=6 2 2 3 3 2 6 7 8 9 1

i=7 1 4 5 2 0 10 10 3 2 2

i=8 3 3 2 2 1 5 4 1 1 6

i=9 2 1 3 4 5 2 1 9 8 7

i=10 4 5 2 1 3 3 2 1 0 0

Giả sử start = (5, 6), agent bắt đầu ở hàng 5, cột 6và pham\_vi = 2

Khởi tạo agent\_map toàn -1.

Với pham\_vi=2, vùng quan sát là các (i,j) thỏa abs(i-5) <=2 và abs(j-6) <=2:

Hàng i ∈ {3,4,5,6,7}

Cột j ∈ {4,5,6,7,8}

Vùng quan sát = ô vuông 5×5: (3..7, 4..8)

Ta cập nhật agent\_map chỉ tại ô (3..7,4..8) với giá trị tương ứng từ ma\_tran. Các vị trí khác vẫn -1.

j=4 j=5 j=6 j=7 j=8

i=3 1 5 2 2 9

i=4 0 4 4 5 1

i=5 1 1 2 3 4

i=6 3 2 6 7 8

i=7 2 0 10 10 3

Tạo niemtin, ví dụ một vài ô

(3,4) , {1}

(3,5) , {5}

(3,6) , {2}

(3,7) , {2}

(3,8) , {9}

Tiếp theo, hàm Quan\_Sat\_1\_P :

Quan sát tất cả ô nằm trong hình vuông bán kính pham\_vi quanh vị trí (x,y)

Cập nhật agent\_map tại những ô quan sát được bằng giá trị thật từ ma\_tran (nếu khác nhau) và trả về danh sách các ô đã thay đổi: danh\_sach\_thay\_doi

Khởi tạo danh sách rỗng danh\_sach\_thay\_doi = [] để thu các ô vừa được cập nhật

Duyệt tất cả cặp ( dx, dy) trong khoảng [-2;2], ma trận vuông kích thước 5 x 5. Nếu giá trị đang có trong agent map khác giá trị thật trong ma\_tran thì cập nhật agent\_map bằng giá trị thật đó và thêm vào danh sách

Ví dụ: kích thước 5x5 dễ minh họa, 12 x12 tương tự

ma\_tran (index chạy từ 0):

[ [0, 1, 2, 0, 0],

[0, 3, 0, 4, 0],

[1, 0, 2, 0, 5],

[0, 0, 0, 3, 0],

[4, 0, 0, 0, 0] ]

agent\_map:

[ [-1, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, -1, -1, -1, -1] ]

Bắt đầu từ x=2, y=2 , pham\_vi = 1. Vùng quan sát là hình vuông 3×3 gồm các ô:

(nx,ny) = (1,1),(1,2),(1,3)

(2,1),(2,2),(2,3)

(3,1),(3,2),(3,3)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Bước | dx, dy | nx, ny | agent\_map[nx][ny] trước | ma\_tran[nx][ny] | Khác  không | Cập nhật |
| 1 | |  | | --- | | (-1,-1) | | |  | | --- | | (1,1) | | |  | | --- | | -1 | | |  | | --- | | 3 | | |  | | --- | | khác | | agent\_map[1][1]=3 ; append (1,1) |
| 2 | (-1,0) | (1,2) | -1 | 0 | khác | agent\_map[1][2]=0 ; append (1,2) |
| 3 | -1,1 | 1,3 | -1 | 4 | khác | agent\_map[1][3]=4 ; append (1,3) |
| 4 | 0,-1 | 2,1 | -1 | 0 | khác | agent\_map[2][1]=0 ; append (2,1) |
| 5 | 0,0 | 2,2 | -1 | 2 | khác | agent\_map[2][2]=2 ; append (2,2) |
| 6 | 0,1 | 2,3 | -1 | 0 | khác | agent\_map[2][3]=0 ; append (2,3) |
| 7 | 1,-1 | 3,1 | -1 | 0 | khác | agent\_map[3][1]=0 ; append (3,1) |
| 8 | 1,0 | 3,2 | -1 | 0 | khác | agent\_map[3][2]=0 ; append (3,2) |
| 9 | 1,1 | 3,3 | -1 | 3 | khác | agent\_map[3][3]=3 ; append (3,3) |

Vì tất cả agent\_map ban đầu là -1, tất cả 9 ô đều khác ma\_tran nên tất cả đều cập nhật.

Kết quả trả về: danh\_sach\_thay\_doi cuối cùng =[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]

agent\_map sau khi hàm chạy (phần quan sát được)

[ [-1, -1, -1, -1, -1],

[-1, 3, 0, 4, -1],

[-1, 0, 2, 0, -1],

[-1, 0, 0, 3, -1],

[-1, -1, -1, -1, -1] ]

def Cap\_Kha\_Nang\_An(niemtin):

    counts = {}

    for pos, tap in niemtin.items():

        for v in tap:

            if v != 0:

                counts[v] = counts.get(v, 0) + 1

    for v, c in counts.items():

        if c >= 2:

            return True

    return False

Class UI

Ghi\_Lich\_Su(): Ghi lại thông tin mỗi khi người chơi hoàn thành một level. Bao gồm tên thuật toán sử dụng, cấp độ, thời gian hoàn thành và dung lượng bộ nhớ tiêu thụ. Hàm sẽ lấy thời gian hiện tại trừ start\_time để tính thời gian chơi, sau đó ghi vào file history.csv ở dạng bản

draw\_background(): Tạo hiệu ứng nền động cho menu chính. Dùng các giá trị màu chuyển động nhẹ theo thời gian để nền không bị tĩnh, giúp giao diện sống động hơn.

Tao\_board\_game(): để tạo ma trận 12 x 12, gồm ma trận ngoài cùng là số 0,bên trong là ma trận 10 x10 chứa các icon Pikachu. Hàm sẽ sinh ngẫu nhiên các cặp giá trị 1,2,3,4,5,6,7,8,9,10 sao cho mỗi icon đều có 2 bản để sau này dùng thuật toán ăn

Gan\_icon(): để hiện các icon Pokemon tương ứng với với giá trị đó trên giao diện

Ve\_Duong\_Di(): để vẽ line nối hai con Pokemon khi cặp đó được ăn, dựa vào danh sách trong path để vẽ các đoạn thẳng nối liên tiếp

Ve\_Duong\_Di\_1(): cũng để vẽ line như hàm trên Ve\_Duong\_Di() nhưng hiển thêm chi phí cost, heuristic, dùng các thuật toán có sử dụng chi phí

Ham\_reset(): sắp xếp lại vị trí các con Pokemon mỗi lần không có đường ăn, thuận lợi cho việc tiếp tục tìm cặp

chon\_Thuat\_Toan(): nhận vào là chuỗi tên thuật toán, khởi tạo đối tượng để game gọi

Kiem\_Tra\_Het(): để kiểm tra xem còn icon nào chưa ăn hết không, duyệt qua các hàng, cột, nếu tất cả ô bằng 0 thì gọi Show\_Congrats() để hiện chúc mừng và tăng level

Sap\_Hinh(): sau khi các cặp được ăn, các con Pokemon còn lại sẽ dịch chuyển dồn theo hướng nhất định, trong ham Lay\_Huong\_Sap() tùy theo từng level down , up, left, right, center

Show\_Congrats(): để hiển thị hiệu ứng khi qua màn, hiện hình tương ứng icon của từng level

Tu\_Dong(): các thuật toán BFS, DFS, A\*, Greedy tự động dò tìm các cặp có thể ăn được, duyệt toàn bộ ma trận, tìm cặp có thể nối thỏa nhỏ hơn hoặc bằng 2 lần rẽ, vẽ đường ăn và cập nhật ma trận.

Tu\_Dong\_1(): tự động dò tìm các cặp có thể ăn được, duyệt toàn bộ ma trận, tìm cặp có thể nối thỏa nhỏ hơn hoặc bằng 2 lần rẽ, vẽ đường ăn và cập nhật ma trận đối với các thuật toán ràng buộc Backtracking, Forward Checking

Tu\_Dong\_BFS\_Non(doi\_tuong): Mô phỏng agent không biết toàn bộ ma trận , từ đó cập nhật niềm tin khi quan sát thêm. Agent chỉ thấy xung quanh nó trong phạm vi 2 ô, sau đó mở rộng vùng đã biết qua Quan\_Sat()

Tu\_Dong\_BFS\_1\_P(): Mô phỏng agent chỉ biết một phần ma trận , từ đó cập nhật niềm tin khi quan sát thêm. Agent chỉ thấy xung quanh nó trong phạm vi 2 ô, sau đó mở rộng vùng đã biết qua Quan\_Sat()

Class UI\_MENU() ( Giao diện menu với các sự lựa chọn)

draw\_background(): tạo hiệu ứng hình nền động animation cho menu bằng cách dịch chuyển nhẹ hình nền theo tọa độ x và y, tạo cảm giác thú vị hơn

draw\_main\_menu(): để hiện màn hình của menu bao gồm: nút play, options, history

draw\_choose\_screen(): để vẽ một giao diện khác hiện danh sách các thuật toán, nút accept và nút cancel

Class BFS

Hàm khởi tạo \_\_init\_\_(): khởi tạo đối tượng BFS, các thuật toán sau cũng khởi tạo tương tự

ma\_tran: ma trận trò chơi Pikachu

self.dir: 4 hướng di chuyển : phải, xuống, lên, trái.

self.simulation: danh sách lưu các bước đi để vẽ mô phỏng đường đi

Hàm La\_Dich(): để kiểm tra trạng thái hiện tại có phải là trạng thái đích hay không, xác định BFS đã tìm thấy đường nối đúng hay chưa

Hàm Sinh\_Trang\_Thai\_Ke\_Tiep\_BFS(): từ trạng thái hiện tại, sinh ra các trạng thái kế tiếp và thêm vào hàng đợi BFS, tạo trạng thái mới Trang\_Thai\_Moi, thêm vào hàng đợi hang\_doi và tập visited, ghi lại parent[Trang\_Thai\_Moi] = Trang\_Thai\_HT để truy vết đường đi, lưu vào self.simulation để vẽ lại đường di chuyển sau này.

Hàm BFS():

Khởi tạo hàng đợi (queue) bằng deque với trạng thái ban đầu.

Lặp đến khi hàng đợi rỗng:

Lấy trạng thái đầu tiên ra (popleft()) xét

Nếu là đích

Dò ngược parent[] để xây dựng đường đi hoàn chỉnh.

Trả về danh sách tọa độ của đường đi.

Nếu chưa phải đích:

Gọi Sinh\_Trang\_Thai\_Ke\_Tiep\_BFS() để mở rộng

Hàm Ve\_Simulation(): mô phỏng lại quá trình duyệt BFS

Class DFS

Hàm Sinh\_Trang\_Thai\_Ke\_Tiep\_DFS(): từ trạng thái hiện tại, sinh ra các trạng thái kế tiếp và thêm vào hàng đợi DFS, tạo trạng thái mới Trang\_Thai\_Moi, thêm vào stack và tập visited, ghi lại parent[Trang\_Thai\_Moi] = Trang\_Thai\_HT để truy vết đường đi, lưu vào self.simulation để vẽ lại đường di chuyển sau này.

Hàm BFS():

Khởi tạo stack là một list rỗng

Lặp đến khi stack rỗng:

Lấy trạng thái cuối cùng ra (pop()) xét

Nếu là đích

Dò ngược parent[] để xây dựng đường đi hoàn chỉnh.

Trả về danh sách tọa độ của đường đi.

Nếu chưa phải đích:

Gọi Sinh\_Trang\_Thai\_Ke\_Tiep\_DFS() để mở rộng

Hàm Ve\_Simulation(): mô phỏng lại quá trình duyệt DFS

Class A Sao

Hàm khởi tạo \_\_init\_\_()

lan\_vo: biến đếm để ưu tiên khi chèn vào hàng đợi heapq, tránh lỗi so sánh khi chi phí bằng nhau

Hàm Cost(): mọi bước đi đều có chi phí bằng 1

Hàm Heuristic():tính theo khoảng cách Manhattan, |x2 – x1| + |y2 – y1|

Hàm Heuristic\_1(): tính theo Euclidean, sqrt((x2- x1)\*\*2 + (y2 – y1)\*\*2)

Hàm Sinh\_Trang\_Thai\_Ke\_Tiep\_A\_Sao ():

cost\_all = g + h, trong đó:

g: cost từ điểm đầu đến đang xét

h: khoảng cách heuristic tới đích

Độ ưu tiên trong hàng đợi dựa vào f = g + h

Nếu trạng thái mới chưa được thăm hoặc tìm thấy đường rẻ hơn, thì:

Cập nhật visited

Cập nhật parent để truy vết đường đi

Đưa vào hàng đợi heapq để A\* chọn tiếp bước tối ưu

Hàm Asao()

Sử dụng hàng đợi ưu tiên chứa : chi phí f , thứ tự, trạng thái

Mỗi lần lặp:

Lấy ra trạng thái có f nhỏ nhất

Nếu là đích, truy lại đường đi path đầu tới đích, chi phí, danh sách chi phí tương ứng tại mỗi bước

Ngược lại, sinh trạng thái mới

Class Greedy():

Hàm Sinh\_Trang\_Thai\_Ke\_Tiep\_Greedy()

Di chuyển sang 4 hướng (phải, xuống, lên, trái)

Di chuyển được nếu:

Ô mới là ô trống (0), hoặc là ô đích cùng giá trị.

Không quay ngược lại hướng cũ.

Cho phép rẽ tối đa 2 lần ( 3 đoạn thẳng)

hx là khoảng cách đến đích (Manhattan )

Thêm vào heapq theo thứ tự hx

Ghi lại trạng thái đã đi và đường nối để hiển thị

Hàm Greedy():

Lấy trạng thái có heuristic nhỏ nhất, gần đích nhất

Nếu đến đích, dựng lại đường đi path từ đích tới đầu, sau đó reverse() để đúng thứ tự

Class BEAM\_SEARCH

Hàm tính heuristic sử dụng Manhattan để tính khoảng cách giữa vị trí hiện tại và đích, càng nhỏ càng gần đích

Hàm Sinh\_Trang\_Thai\_Ke\_Tiep\_Beam():

Duyệt 4 hướng khả thi từ (x, y).

Chỉ mở rộng ô mới nếu:

Ô đó trống (0) hoặc là đích hợp lệ (cùng value và tọa độ đích).

Không đi ngược hướng trước (kiểm tra (self.dirs[idx] + self.dirs[huong]) != (0, 0)).

new\_re: số lần rẽ mới (chỉ +1 nếu hướng thay đổi).

Nếu new\_re ≤ 2, thêm vào danh sách trang\_thai\_moi cùng điểm heuristic h

Lưu (x\_new, y\_new, idx, new\_re) vào visited để tránh lặp

Ghi lại đường đi để mô phỏng

Hàm beam\_search():

beam\_width: giới hạn số lượng nhánh được giữ lại mỗi vòng, chỉ giữ 3 con có heuristic tốt nhất.

ds: hàng đợi ưu tiên lưu (f, path, hướng, số lần rẽ).

Vòng lặp:

Lấy ra tối đa beam\_width trạng thái tốt nhất.

Sinh các trạng thái con mới.

Gom tất cả con lại , chọn ra beam\_width con có giá trị heuristic nhỏ nhất để tiếp tục.

Kết thúc khi:

Tìm thấy đường hợp lệ (đến đích, ≤ 2 lần rẽ).

Hoặc không còn đường nào để mở rộng.

Class HILL\_CLIMBING

Hàm tính heuristic sử dụng Manhattan để tính khoảng cách giữa vị trí hiện tại và đích, càng nhỏ càng gần đích

Hàm Sinh\_Trang\_Thai\_Ke\_Tiep\_hill\_clambing()  
So sánh heuristic mới hx với hiện tại hx\_ht.  
Chỉ mở rộng nếu tiến gần đích hơn hoặc bằng (hx ≤ hx\_ht).

Nếu đi xa hơn (hx > hx\_ht) thì bỏ qua (không leo ngược đồi).

Giới hạn ≤ 2 lần rẽ.

Lưu parent để có thể truy ngược đường đi.

Mỗi trạng thái hợp lệ được đưa vào hang\_doi

Hàm hill\_clambing():

Bắt đầu từ trạng thái ban đầu (gốc).

Vòng lặp:

Lấy ra trạng thái hiện tại có heuristic tốt nhất.

Nếu đạt đích thì truy ngược đường đi và trả kết quả.

Ngược lại, chỉ sinh các trạng thái con tốt hơn hoặc bằng.

Nếu không còn con nào tốt hơn thì dừng (mắc kẹt).

Class MU\_TOAN\_PHAN

Hàm KhoiTao\_NiemTin\_Non: Khởi tạo niềm tin ban đầu cho từng ô, ô trống có tập rỗng, ô có giá trị thì chứa giá trị từ 1 tới 10

Hàm Quan\_Sat: Cập nhật bản đồ agent theo phạm vi quan sát và trả về danh sách ô có thay đổi

Hàm Cap\_Nhat\_Niem\_Tin\_Sau\_Quan\_Sat: Điều chỉnh niềm tin dựa trên kết quả quan sát , ô trống thì rỗng, ô có giá trị thì xác định chắc chắn

Hàm Doan\_Cap\_Tu\_Niem\_Tin: Dự đoán ngẫu nhiên một cặp ô có cùng giá trị khả thi từ niềm tin

Hàm Co\_The\_An\_BFS: Kiểm tra bằng BFS xem hai ô có thể nối được với nhau (≤2 lần rẽ, đi qua ô trống)

Hàm Cap\_Kha\_Nang\_An: Kiểm tra xem còn ít nhất một giá trị xuất hiện ≥2 lần trong niềm tin

Class MU\_1\_P

Hàm KhoiTao\_NiemTin\_1\_P: Khởi tạo bản đồ niềm tin và bản đồ agent ban đầu cho biết 1phần ma trận

Hàm Quan\_Sat\_1\_P: Cập nhật thông tin bản đồ agent trong phạm vi quan sát và trả về danh sách ô có thay đổi.

Class BACKTRACKING

Hàm \_is\_opposite(a, b) :Kiểm tra xem hai hướng a và b có đối nhau (ngược chiều) hay không. Dùng để tránh đi lùi lại hướng cũ.

Hàm init\_Tap\_Gia\_Tri() → Tạo danh sách 4 ô kề cạnh (trên, dưới, trái, phải) từ vị trí hiện tại (x, y) để xét bước tiếp theo.

Hàm init\_Tap\_Rang\_Buoc()   
Xác định xem ô kế tiếp có hợp lệ không, dựa trên:

Ô đó có nằm trong bản đồ và không bị chặn.

Không đi ngược hướng trước.

Số lần rẽ không vượt quá 2.

Hàm tìm đường đi từ điểm hiện tại đến đích bằng thuật toán quay lui (Backtracking):

Mỗi bước xét các ô lân cận hợp lệ.

Nếu đến đích thì trả về đường đi.

Nếu bế tắc thì quay lui (xóa bước cuối, thử hướng khác).

Giới hạn số lần đệ quy để tránh lặp vô hạn.

Class FORWARD CHECKING

Hàm forward\_checking(): Tạo tập các ô có thể đi tiếp từ vị trí (x, y) (ô trống hoặc trùng giá trị với đích), giúp loại bỏ các ô không hợp lệ trước khi thử.

Hàm init\_Tap\_Rang\_Buoc() : Kiểm tra ràng buộc di chuyển:

Ô hợp lệ (trong bản đồ, không bị chặn).

Không đi ngược hướng cũ.

Số lần rẽ ≤ 2.

Hàm FW\_Backtracking() : Thuật toán Backtracking kết hợp Forward Checking:

Dùng forward\_checking để chỉ xét các ô khả thi.

Duyệt đệ quy từng hướng hợp lệ, ghi nhận số lần rẽ.

Nếu tới đích thì trả về đường đi.

Nếu bế tắc → quay lui, thử hướng khác.

**CHƯƠNG 8: GIAO DIỆN VÀ ĐÁNH GIÁ HIỆU SUẤT**

8.1 Giao diện MenuA screenshot of a video game

AI-generated content may be incorrect.

Giao diện menu game Pikachu có 3 nút: Play, Options, History

8.2 Bảng danh sách các thuật toán

Khi ấn nút OPTIONS