TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

Logo

Description automatically generated

**TIỂU LUẬN CUỐI KỲ**

**MÔN HỌC: TRÍ TUỆ NHÂN TẠO**

**TÊN TIỂU LUẬN: SỬ DỤNG CÁC THUẬT TOÁN AI VÀO GAME**

**RẮN SĂN MỒI**

**Giảng viên:** GV Phan Thị Huyền Trang

**Sinh viên thực hiện:**

|  |  |  |
| --- | --- | --- |
| **Mã số SV** | **Họ và tên** | **Mức độ**  **đóng góp (%)** |
| 22110184 | Lê Quốc Nam | 50% |
| 22110187 | Lê Chí Nghĩa | 50% |

*TP. Hồ Chí Minh, tháng 11 năm 2024*

**Quy định chung:**

**- Bản báo cáo trình bày trên giấy A4**, font size 13, lề trái 3cm; lề phải-trên-dưới 2cm; dãn dòng 1.3 lines.

- Mỗi phần được bắt đầu trang mới.

**\* Gợi ý cấu trúc báo cáo:**

**Nội dung báo cáo tiểu luận đánh giá điểm cuối kỳ gồm các nội dung chính sau:**

- Trang bìa (Tên trường, khoa; tên môn học; Tên đề tài, tên giảng viên, Mã số và họ tên sinh viên)

- Mục lục

- Danh mục các cụm từ/từ viết tắt

- Danh sách hình vẽ

- Danh sách bảng biểu

**Phần 1.** Mở đầu (mô tả về project)

- Phát biểu bài toán

- Mục đích, yêu cầu cần thực hiện

- Phạm vi và đối tượng

**Phần 2.** Cơ sở lý thuyết dùng để thực hiện project

- Trình bày về công cụ và môi trường để lập trình, thư viện hỗ trợ lập trình, ngôn ngữ lập trình, các phương pháp, kỹ thuật được sử dụng ...

**Phần 3.** Phân tích, thiết kế giải pháp

- Sơ đồ khối/ ý tưởng thuật toán

- Chi tiết về các thuật toán chính

**Phần 4.** Thực nghiệm, đánh giá, phân tích kết quả

- Mô tả quá trình đánh giá thử nghiệm, mô tả về dữ liệu... cấu hình phần cứng, các tham số thiết lập...

- Trình bày các kết quả thử nghiệm,

- Giới thiệu kết quả giao diện phần mềm, giải thích/hướng dẫn thực thi phần mềm....

**Phần 5.** Phần kết luận

- Đánh giá những kết quả đã thực hiện được

- Định hướng phát triển

**Tài liệu tham khảo**

Danh mục các tài liệu, trang web tham khảo (chú ý chỉ rõ vị trí tham khảo trong báo cáo)

**\* Phần trình bày báo chuẩn bị bằng PPT slides.**

**Phần 1: Mô tả dự án Snake Game với AI**

**1.1. Phát biểu bài toán**

Trời thời đại mà công nghệ phát triển mạnh mẽ, trong 2 năm trở lại đây các công cụ AI trở nên ngày càng phổ biến và có một sự ảnh hưởng rất lớn đến thị trường ngành Công nghệ thông tin. Với tính chất quan trọng như vậy, việc học các công thức cũng như các thuật toán là điều cần thiết đối với mọi người. Để bắt đầu việc học các công nghệ đó, có thể sẽ cần các dự án để áp dụng và một dự án game Rắn săn mồi là một game cơ bản và cổ điển mà có thể ai cũng biết. Dự án này hướng đến việc phát triển một phiên bản Snake Game hiện đại, trong đó tích hợp các thuật toán trí tuệ nhân tạo để tự động hoá việc điều khiển con Rắn. Thông qua việc ứng dụng các thuật toán vào game, mọi người có thể phần nào hình dung được cách thuật toán chạy và tính thực tiễn của các thuật toán đó.

**1.2. Mục đích và yêu cầu thực hiện**

**Mục đích**

Giáo dục: Tạo ra một ví dụ thực tế về việc ứng dụng các thuật toán AI vào game, giúp người học dễ dàng hơn trong việc tiếp cận về các khái niệm và trí tuệ nhân tạo.

Nghiên cứu: Khám phá sự thần kỳ và khả năng của các thuật toán trong việc tối ưu chiến lược chơi game.

Giải trí: Tạo ra một phiên bản Snake Game thú vị, nơi mà mọi người có thể tương tác và quan sát cách AI học cách chơi.

**Yêu cầu thực hiện**

1. Về gameplay:

Phát triển giao diện trực quan, dễ dàng quan sát và sử dụng.

Tích hợp các lựa chọn và chế độ chơi.

Hiển thị điểm số và các thông số quan trọng trong quá trình chơi.

1. Về AI  
   Xây dựng các thuật toán dựa trên các kiến thức đã học.

Đạt được số điểm tối ưu và tránh các chướng ngại vật hiệu quả.

Có khả năng thích nghi với các sự thay đổi của môi trường game.

1. Về kỹ thuật

Tối ưu hoá hiệu suất để game chạy mượt mà.

Giao diện dễ quan sát và đánh giá.

Hệ thống lưu log dưới terminal.

**1.3. Phạm vi và đối tượng**

**2. Cơ sở lý thuyết thực hiện project**

**2.1. Công cụ và môi trường lập trình**

Ngôn ngữ lập trình Python: Sử dụng ngôn ngữ lập trình python 3.x.x và làm việc với Pycharm.

Các thư viện hỗ trợ:

PyGame; Thư viện giúp phát triển game dễ dàng hơn, có các công cụ đễ xử lý đồ hoạ, âm thanh, quản lý các sự kiện, vễ các đồ thị.

Numpy: Thư viện tính toán số học một cách nhanh và tiện lợi hơn

Random: Hỗ trợ việc random các con số.

Heapq: Một kiểu cấu trúc dữ liệu giúp bạn quản lý danh sách các phần tử danh sách các phần tử sao cho các phép truy cập, thêm, hoặc xóa phần tử

nhỏ nhất diễn ra nhanh chóng.

Deque: Một cấu trúc dữ liệu **hàng đợi hai đầu (double-ended queue)**,

cho phép thêm (**append**) và xóa (**pop**) phần tử ở cả hai đầu với hiệu

năng cao.

**2.2. Các thuật toán tìm đường đi**

a. Breadth-First Search (BFS)

b. A\*

c. Backtracking

d. Simulated Annealing

**3. Phân tích, thiết kế giải pháp**

3.1 BFS

a. Sơ đồ khối

A diagram of a process

Description automatically generated

b. Chi tiết

def bfs(*start\_pos*, *goal\_pos*, *grid*, *obstacles*):

    open\_list = deque() # danh sách các nút đang xét

    visited = set() # danh sách các nút đã xét rồi

    start\_node = Node(*start\_pos*) # Nút khởi đầu là đầu con rắn

    goal\_node = Node(*goal\_pos*) # vị trí thức ăn

    open\_list.append(start\_node)

    visited.add(start\_node)

    while open\_list:

        current\_node = open\_list.popleft() # lấy nút đầu trong danh sách

        # Nếu nó tới được đồ ăn thì trả về đường đi

        if current\_node == goal\_node:

            path = []

            while current\_node != start\_node:

                path.append(current\_node.position)

                current\_node = current\_node.parent

            path.append(start\_node.position)

            return path[::-1]

        # Lấy các nút mà con rắn có thể di chuyển

        for child in current\_node.get\_neighbors(*grid*, *obstacles*):

            if child not in visited:

                visited.add(child)

                open\_list.append(child)

    return None

Thuật toán tìm kiếm theo chiều rộng (Breadth-First Search - BFS) là một phương pháp duyệt đồ thị hoặc cây, bắt đầu từ một nút gốc và khám phá ra dần các nút lân cận. Thuật toán sẽ ưu tiên duyệt qua các nút có cùng độ sâu trước khi chuyên qua các mức sâu hơn, đảm bảo với tên gọi là duyệt theo chiều rộng.

Đầu tiên khai báo những thứ cần thiết. Dùng một deque và deque có thể lấy dữ liệu từ 2 đầu, việc này có ích trong việc lấy phần tử để duyệt. Một set là visited để lưu có nút đã từng duyệt qua đảm bảo việc mỗi nút chỉ được duyệt một lần. Lặp cho đến deque rỗng (open\_list trong code), lấy nút đầu trong danh sách và kiểm tra xem nó đã đến được đích hay chưa, nếu đã đến tạo 1 danh sách truy vết, còn nếu không thì đi duyệt các nút cận, kiểm tra nút đó đã đi chưa và thêm vào deque. Cứ lặp cho đến khi tìm được đường đi. Trả về None nếu không tìm thấy đường đi.

3.2 A\*

a. Sơ đồ khối

A diagram of a flowchart

Description automatically generated

b. Chi tiết

# Hàm tính khoảng cách bằng Manhattan

def heuristic(*node*, *goal\_node*):

    return abs(*node*.position[0] - *goal\_node*.position[0]) + \

        abs(*node*.position[1] - *goal\_node*.position[1])

def a\_star(*start\_pos*, *goal\_pos*, *grid*, *obstacles*):

    open\_list = [] # danh sách nút có thể đi

    visited = [] # danh sách nút đã đi

    start\_node = Node(*start\_pos*)

    goal\_node = Node(*goal\_pos*)

    # Dùng heapq để có thể lấy các đường có cost nhỏ nhất

    heapq.heappush(open\_list, (0 + heuristic(start\_node, goal\_node), start\_node))

    g\_costs = {start\_node: 0}

    h\_costs = {}

    traces = {}

    while open\_list:

        \_, current\_node = heapq.heappop(open\_list)

        visited.append(current\_node)

        if current\_node == goal\_node:

            path = []

            while current\_node != start\_node:

                path.append(current\_node.position)

                current\_node = traces[current\_node]

            path.append(start\_node.position)

            return path[::-1]

        for child in current\_node.get\_neighbors(*grid*, *obstacles*):

            if child in visited:

                continue

            g\_cost = g\_costs[current\_node] + 1

            h\_cost = heuristic(child, goal\_node)

            f\_cost = g\_cost + h\_cost

            if child not in g\_costs or g\_cost < g\_costs[child]:

                g\_costs[child] = g\_cost

                h\_costs[child] = h\_cost

                traces[child] = current\_node

                heapq.heappush(open\_list, (f\_cost, child))

    return None

A\* là thuật toán tìm kiếm trong đồ thị theo loại Informed Search. Thuật toán sẽ tìm một đường đi từ một nút gốc tới một nút đích cho trước (hoặc tới một nút thỏa mãn một điều kiện đích). Thuật toán sử dụng một hàm "heuristic" để xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó. Thuật toán này duyệt các nút theo thứ tự của đánh giá heuristic này. Do đó, thuật toán A\* là một ví dụ của tìm kiếm theo lựa chọn tốt nhất (best-first search).

Với A\*, ta cần một hàm heuristic để ước lượng khoảng cách từ nút hiện tại tới nút cần đến. Hàm heuristic được tính theo 3 công thức, ở đây dùng theo Manhattan, hàm này tính bằng cách lấy toạ độ x, y của 2 nút. Dùng một heapq để open\_list có thể lưu được một tuple và có thể dùng để lấy giá trị nhỏ nhất của f\_cost. Duyệt cho đến khi open\_list rỗng hoặc không tồn tại đường đi. Tương tự BFS, duyệt các nút cận để tiếp tục đi, nhưng ở đây không dùng for để đi lần lượt các nút cận mà dùng các đánh giá trị khoảng cách của từng nút trong danh sách cận đó đến đích, nút nào có giá trị g\_cost nhỏ nhất sẽ được ưu tiên vào hàng đợi. Lặp lại việc này cho điến khi tìm được đường đi có giá trị khoảng cách nhỏ nhất hoặc không có đường đi nào.

3.3 Backtracking

a. Sơ đồ khối

A diagram of a process

Description automatically generated

b. Chi tiết

ef backtracking(*start\_pos*, *goal\_pos*, *grid*, *obstacles*):

    start\_node = Node(*start\_pos*)

    goal\_node = Node(*goal\_pos*)

    visited = set()

    path = []

    def backtrack(*current\_node*, *goal\_node*):

        if *current\_node* == *goal\_node*:

            return True

        visited.add(*current\_node*.position)

        for neighbor in *current\_node*.get\_neighbors(*grid*, *obstacles*):

            if neighbor.position not in visited:

                path.append(neighbor.position)

                if backtrack(neighbor, *goal\_node*):

                    return True

                path.pop()

        return False

    path.append(start\_node.position)

    if backtrack(start\_node, goal\_node):

        return path

    return None

Backtracking (quay lui) là một thuật toán tìm kiếm và tối ưu mạnh mẽ, thường được sử dụng để giải quyết các bài toán tổ hợp như là phân chia tập hợp, tô màu đồ thị và ìm đường trong không gian trạng thái. Thuật toán này hoạt động dựa trên đệ quy, thông qua việc kiểm tra các khả năng khả thi và xây dựng dần dần lời giải, sau đó quay lui khi phát hiện lựa chọn hiện tại không dẫn tới kết quả.

Trước khi bước đầu triển khai thuật toán, khai báo hai mảng hỗ trợ:

+ Một mảng path để lưu lại đường đi đến đích.

+ Một mảng kiểm tra trạng thái các nút, đảm bảo không lặp lại nút đã duyệt qua.

Thuật toán bắt đầu với một hàm kiểm tra trạng thái của nút hiện tại, kiểm tra xem nút hiện tại có đang bằng nút cần đến hay chưa, nếu rồi thì return True cho chương trình, còn nếu không, duyệt qua các nút cận có thể của nút hiện tại, đệ quy sẽ xuất hiện tại đây, nó sẽ đi đi tìm bằng cách lấy nút cận và lặp lại các bước trên, nếu nút cuối cũng vẫn không đến được nút đích thì pop nó ra khỏi path và thuật toán quay lui để tiếp tục thử các nhánh khác.

3.4 Simulated Annealing

a. Sơ đồ khối

A diagram of a process

Description automatically generated

b. Chi tiết

def heuristic(*node*, *goal\_node*):

    return abs(*node*.position[0] - *goal\_node*.position[0]) + abs(

*node*.position[1] - *goal\_node*.position[1]

    )

def simulated\_annealing(*start\_pos*, *goal\_pos*, *grid*, *obstacles*):

    init\_temporature = 1000

    final\_temp = 1  # nhiệt độ khi thuật toán dừng

    alpha = 0.95  # hệ số làm mát

    current = Node(*start\_pos*)

    goal = Node(*goal\_pos*)

    path\_curr = [current]

    path\_best = path\_curr

    best\_cost = float("inf")

    temp = init\_temporature

    while temp > final\_temp:

        curr\_node = path\_curr[-1]

        neighbors = curr\_node.get\_neighbors(*grid*, *obstacles*)

        if not neighbors:

            break

        next\_node = random.choice(neighbors)

        # Tạo đường đi mới bằng cách thêm node tiếp theo vào đường đi hiện tại

        # Cộng hai danh sách để được đường đi

        new\_path = path\_curr + [next\_node]

        current\_cost = len(path\_curr) + heuristic(path\_curr[-1], goal)

        new\_cost = len(new\_path) + heuristic(next\_node, goal)

        delta\_cost = new\_cost - current\_cost

        if delta\_cost < 0 or random.random() < math.exp(-delta\_cost / temp):

            path\_curr = new\_path

            if new\_cost < best\_cost:

                path\_best = new\_path

                best\_cost = new\_cost

        if next\_node == goal:

            return [node.position for node in path\_best]

        temp \*= alpha

    if path\_best:

        return [node.position for node in path\_best]

    return None

Simulated annealing (mô phỏng ủ nhiệt) là một thuật toán tối ưu ngẫu nhiên, lấy cảm hứng từ nguyên lý làm mát chầm chậm kim loại trong quá trình ủ nhiệt. Nó được sử dụng để tìm kiếm lời giải gần cho các bài toán tối ưu hoá phức tạp bằng cách chấp nhận những lời giải tạm thời kém hơn để tránh bị kẹt trong các trường hợp cục bộ.

Thuật toán bắt đầu với một nhiệt độ cao (1000 độ) và dần dần giảm nhiệt độ theo hệ số làm mát alpha (0.95) cho tới khi nhiệt độ giảm còn nhiệt dộ cuối cùng (final\_temp). Tại mỗi bước nó chọn một nút cận và tính toán chi phí của đường đi mới. Nếu chi phí mới thấp hơn hoặc thoả mãn một điều kiện ngẫu nhiên ( dựa trên nhiệt độ hiện tại) thì đường đi hiện tại sẽ được cập nhật. Quá trình này giúp thuật tránh các trường hợp mắc kẹt trong không gian tìm kiếm. Khi nhiệt độ giảm xuống dưới nhiệt độ cho phép thuật toán sẽ dừng lại và trả về đường đi tốt nhất tìm được.

3.5 Q Learning

a. Sơ đồ khối

A diagram of a flowchart

Description automatically generated

b. Chi tiết

class QLearning:

    def \_\_init\_\_(*self*, *size\_of\_state*, *size\_of\_action*):

*self*.size\_of\_state = *size\_of\_state*

*self*.size\_of\_action = *size\_of\_action*

*self*.learning\_rate = 0.1  # tỷ lệ học

*self*.discount\_factor = 0.95  # hệ số giảm giá

*self*.epsilon = 0.1  # tỷ lệ khám phá ngẫu nhiên

        # Tạo thư mục models nếu chưa tồn tại

*self*.model\_dir = os.path.join(os.path.dirname(\_\_file\_\_), "models")

        os.makedirs(*self*.model\_dir, *exist\_ok*=True)

        # Tìm file q\_table có điểm số cao nhất

        files = glob.glob(os.path.join(*self*.model\_dir, "q\_table\_\*.npy"))

        best\_score = -1

        best\_table = None

        for file in files:

            score = int(re.findall(r"q\_table\_*(\d*+*).*npy", os.path.basename(file))[0])

            if score > best\_score:

                best\_score = score

                best\_table = file

        if best\_table:

*self*.q\_table = np.load(best\_table)

            print(f"Đã tải Q-table từ file {os.path.basename(best\_table)}")

        else:

            print("Tạo Q-table mới")

*self*.q\_table = np.zeros((*size\_of\_state*, *size\_of\_action*))

    def get\_action(*self*, *state*):

        if np.random.rand() < *self*.epsilon:

            return np.random.randint(*self*.size\_of\_action)

        else:

            return np.argmax(*self*.q\_table[*state*])

    # cập nhật Q-table với số điểm thưởng

    def update(*self*, *state*, *action*, *reward*, *next\_state*, *done*):

        if *done*:

            point = *reward*

        else:

            point = *reward* + *self*.discount\_factor \* np.amax(*self*.q\_table[*next\_state*])

*self*.q\_table[*state*][*action*] = (1 - *self*.learning\_rate) \* *self*.q\_table[*state*][*action*]

+ *self*.learning\_rate \* point

        # giảm tỷ lệ khám phá ngẫu nhiên sau mỗi lần

        if *self*.epsilon > 0.01:

*self*.epsilon \*= 0.995

    def get\_state(*self*, *game*):

        head\_x, head\_y = *game*.snake.get\_head\_position()

        food\_x, food\_y = *game*.food.position

        head\_x = int(head\_x / GRIDSIZE)

        head\_y = int(head\_y / GRIDSIZE)

        food\_x = int(food\_x / GRIDSIZE)

        food\_y = int(food\_y / GRIDSIZE)

        # Tính toán hướng tương đối của thức ăn

        food\_dir\_x = 0

        food\_dir\_y = 0

        if food\_x < head\_x:

            food\_dir\_x = -1

        elif food\_x > head\_x:

            food\_dir\_x = 1

        if food\_y < head\_y:

            food\_dir\_y = -1

        elif food\_y > head\_y:

            food\_dir\_y = 1

        # Kiểm tra nguy hiểm ở 4 hướng

        array\_consider = [0, 0, 0, 0]  # [UP, DOWN, LEFT, RIGHT]

        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        for i, (dx, dy) in enumerate(directions):

            next\_x = head\_x + dx

            next\_y = head\_y + dy

            # Kiểm tra va chạm với tường

            if (

                next\_x < 0

                or next\_x >= GRID\_WIDTH

                or next\_y < 0

                or next\_y >= GRID\_HEIGHT

            ):

                array\_consider[i] = 1

            elif (next\_x \* GRIDSIZE, next\_y \* GRIDSIZE) in *game*.obstacles.positions:

                array\_consider[i] = 1

            elif (next\_x \* GRIDSIZE, next\_y \* GRIDSIZE) in *game*.snake.positions[1:]:

                array\_consider[i] = 1

        state = (

            array\_consider[0] \* 1

            + array\_consider[1] \* 2

            + array\_consider[2] \* 4

            + array\_consider[3] \* 8

            + (food\_dir\_x + 1) \* 16

            + (food\_dir\_y + 1) \* 48

        )

        return state

Q-Learning là một thuật toán thuộc nhóm Reinforcement Learning ( học tăng cường) mạnh mẽ, được sử dụng để dạy cho các agent cách tự đưa quyết định tối ưu trong môi trường. Thuật toán hoạt động dựa trên nguyên tắc thử và sai, nơi mà agent đưa ra các tương tác với môi trường, thực hiện và nhận phần thưởng hoặc hình phạt từ hành động đó. Sau đó cập nhật bảng Q-table để được trạng thái tốt hơn cho các lần sau.

Thuật toán sử dụng một bảng Q-table để lưu trữ giá trị của từng hành động tại mỗi trạng thái cụ thể. Từng hành động khác nhau và môi trường khác nhau thì Q-Learning sẽ lưu trạng thái đó vào Q-table. Mỗi khi agent thực hiện một hành động bất kì thì hệ thống sẽ tính toán số điểm thường và một vài chỉ số khác dựa vào các tham số đầu vào. Sau đó cập nhật Q-table với giá trị các trạng thái tốt hơn, giúp các lần train sau dễ dàng đạt được số điểm thưởng cao hơn. Điều này giúp agent dần dần cải thiện được chiến lược và hiệu xuất. Q-table được lưu trữ và tải từ file, cho phép tiếp tục train mà không cần khởi tạo lại từ đầu, giúp việc học hiệu quả hơn theo thời gian.

3.6 So sánh các thuật toán

|  |  |  |  |
| --- | --- | --- | --- |
| **Thuật toán** | **Ưu điểm** | **Nhược điểm** | **Thích hợp cho** |
| BFS | * Luôn tìm được đường đi nếu chi phí bằng nhau * Đầy đủ: Nếu có đường đi, BFS sẽ tìm được * Dễ triển khai, dễ hiểu | * Tốn bộ nhớ: phải lưu tất cả trạng thái trong quá trình dò tìm * Chậm trong môi trường rộng lớn | * Tìm đường đi ngắn nhất với đồ thị nhỏ |
| A\* | * Hiệu quả: có kết hợp tính chi phí giúp cải thiện tốc độ * Linh hoạt: có thể tuỳ chỉnh hàm heuritic để phù hợp với bài toán | * Phụ thuộc: hiệu quả sẽ phụ thuộc khá lớn vào hàm heuritic * Khó triển khai: cần nắm rõ các hàm heuritic | * Đường đi ngắn nhất, bài toán tối ưu |
| Backtracking | * Đơn giản: dễ triển khai * Hiệu quả: phù hợp với bài toán nhỏ hoặc cần duyệt tổ hợp | * Chậm chạp: khi môi trường lớn, thuật toán trở nên ngày càng mở rộng khiến mất thời gian * Không tối ưu: không đảm bảo tìm được đường đi tối ưu nhất | * Tổ hợp: sudoku, bài toán có môi trường nhỏ |
| Simulated annealing | * Ngẫu nhiên: không bị kẹt trong cự trị môi trường * Tối ưu bộ nhớ: sử dụng khá ít bộ nhớ | * Không tối ưu: với việc random, thuật toán không đảm bảo tìm tới đường đi nhanh nhất * Phụ thuộc: phụ thuộc khá lớn vào cách chọn các tham số nhiệt dộ avf hệ số * Chậm chạp: so với BFS hay A\* thi SA chậm chạp hơn rất nhiều | * Tối ưu phức tạp, có cực trị |
| Q-Learning | * Không yêu cầu về môi trường: Q-Learning không cần biết trước hoạt động của môi trường, chỉ cần tương tác và nhận phản hồi * Áp dụng: Q-Learning có thể áp dụng cho nhiều bài toán khác nhau | * Không hiệu quả trong môi trường quá lớn * Phụ thuộc vào nhiều tham số * Không thích nghi đủ tốt khi môi trường đột ngột thay đổi | * Training model |

**4. Thực nghiệm, đánh giá và phân tích kết quả**

**4.1. Môi trường thực nghiệm**

**4.1.1. Cấu hình phần cứng và phần mềm**

Về phần cứng:

CPU: core i5 hoặc tương đương

RAM: 8GB trở lên

Về phần mềm:

IDE: yêu cầu có pycharm hoặc visual đã cấu hình python

Python phiên bản 3.8.x

Các thư viện cần thiết: numpy, pygame, matplotlib

**4.1.2. Tham số thiết lập**

Kích thước bảng và grid được định nghĩa trong file constans.py

A screen shot of a computer

Description automatically generated

Có thể tuỳ chỉnh tốc độ khung hình FPS, số lượng vật cản và số lượng game để thực hiện so sánh trong file constants.py:

A blue square with white text

Description automatically generated with medium confidence

Tuỳ chỉnh màu sắc ưu thích của các vật thể trong constants.py:

A black square with white text

Description automatically generated with medium confidence

Khi muốn train model có thể tuỳ chỉnh số lần train, giới hạn số bước di chuyển của con rắn và ẩn hiện UI để việc train nhanh hơn bằng các comment code từ dòng 37 tới 39 trong file src/q\_learning/train.py:

A screen shot of a computer program

Description automatically generated

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Run game \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

+ Run file main\_snake.py để hiện run các thuật toán.

+ Run trong terminal dòng lệnh python -m src.q\_learning.train để bắt đầu train

**4.2. Dữ liệu thử nghiệm**

**4.2.1. Cấu trúc bản đồ**

- Bản đồ được thiết kế dạng các ô lưới 2D

- Các chướng ngại vật được random ngẫu nhiên trên bản đồ

- Con rắn sẽ xuất hiện từ giữa bảng đồ với dộ dài bằng 1

- Thức ăn được spawn ngẫu nhiên

**4.2.2. Các tình huống thử nghiệm**

- Đường đi ngắn (thức ăn gần rắn)

- Đường đi trung bình

- Đường đi phức tạp (nhiều chướng ngại vật)

- Đường đi dài (thức ăn xa rắn)

**4.3. Kết quả thực nghiệm**

**4.3.1. So sánh hiệu suất các thuật toán**

**Các chỉ tiêu đánh giá:**

* Điểm số
* Hiệu quả di chuyển ( số bước di chuyển / số điểm đạt được )
* Hiệu suất thời gian ( thời gian thực thi / số điểm đạt được )

**Kết quả so sánh các chỉ số đối với 4 thuật toán:**

**\*** Các tham số để so sánh là GRID\_SIZE = 60, OBSTACLE = 5, NUM\_COMPARE = 5

Vì thức ăn và chướng ngại vật xuất hiện ngẫu nhiên nên với mỗi lần so sánh sẽ có thay đổi đôi chút, nhưng nhìn chung chung vẫn có thể đánh giá một cách tổng quan.

Ở 5 lần thứ 1: GRID\_SIZE = 60, OBSTACLE = 10, NUM\_COMPARE = 5

A graph of different colored lines

Description automatically generated

Ở 5 lần thứ 2: GRID\_SIZE = 30, OBSTACLE = 10, NUM\_COMPARE = 5

A graph of different colored lines

Description automatically generated

Ở 5 lần thứ 3: GRID\_SIZE = 30, OBSTACLE = 20, NUM\_COMPARE = 5

A graph of different colored lines

Description automatically generated with medium confidence

Sau 3 lần so sánh ta có thể kết luận như sau:

BFS (Breadth-First Search):

+ Luôn tìm được đường đi ngắn nhất

+ Tốc độ xử lý trung bình cao

+ Phù hợp với bản đồ nhỏ và trung bình

A\* (A star):

+ Tìm được đường đi tối ưu

+ Tốc độ xử lý nhanh nhất

+ Hiệu quả trong hầu hết các trường hợp

Backtracking:

+ Đường đi không phải lúc nào cũng tối ưu

+ Tốc độ xử lý trung bình khá

+ Phù hợp với bản đồ đơn giản ít vật cản

Simulated Annealing:

+ Đường đi không hẳn tối ưu

+ Tốc độ xử lý trung bình yếu

+ Cân bằng giữa thời gian và số bước di chuyển

**4.3.2. Giao diện và hướng dẫn sử dụng**

***Hướng dẫn sử dụng: chạy file main\_snake.py để lựa chọn 4 thuật toán tìm đường, chạy lệnh  python -m src.q\_learning.train dưới terminal để train model***

Màn hình chính: Có 5 lựa chọn ứng với 4 thuật toán và 1 phép so sánh, click vào 1 button bất kỳ để chạy thuật toán mong muốn

A screenshot of a computer

Description automatically generated

Màn hình game: Hiện thị điểm số góc trên trái và giao diện các agent, môi trường của game

A screenshot of a video game

Description automatically generated

Màn hình tạm dừng: Nhấn Esc để tạm dừng game. Màn hình sẽ hiển thị 2 lựa chọn là continue để tiếp tục game và restart để lựa chọn lại thuật toán.

A screenshot of a game

Description automatically generated

**Quy định chung:**

**- Bản báo cáo trình bày trên giấy A4**, font size 13, lề trái 3cm; lề phải-trên-dưới 2cm; dãn dòng 1.3 lines.

- Mỗi phần được bắt đầu trang mới.

**\* Gợi ý cấu trúc báo cáo:**

**Nội dung báo cáo tiểu luận đánh giá điểm cuối kỳ gồm các nội dung chính sau:**

- Trang bìa (Tên trường, khoa; tên môn học; Tên đề tài, tên giảng viên, Mã số và họ tên sinh viên)

- Mục lục

- Danh mục các cụm từ/từ viết tắt

- Danh sách hình vẽ

- Danh sách bảng biểu

**Phần 1.** Mở đầu (mô tả về project)

- Phát biểu bài toán

- Mục đích, yêu cầu cần thực hiện

- Phạm vi và đối tượng

**Phần 2.** Cơ sở lý thuyết dùng để thực hiện project

- Trình bày về công cụ và môi trường để lập trình, thư viện hỗ trợ lập trình, ngôn ngữ lập trình, các phương pháp, kỹ thuật được sử dụng ...

**Phần 3.** Phân tích, thiết kế giải pháp

- Sơ đồ khối/ ý tưởng thuật toán

- Chi tiết về các thuật toán chính

**Phần 4.** Thực nghiệm, đánh giá, phân tích kết quả

- Mô tả quá trình đánh giá thử nghiệm, mô tả về dữ liệu... cấu hình phần cứng, các tham số thiết lập...

- Trình bày các kết quả thử nghiệm,

- Giới thiệu kết quả giao diện phần mềm, giải thích/hướng dẫn thực thi phần mềm....

**Phần 5.** Phần kết luận

- Đánh giá những kết quả đã thực hiện được

- Định hướng phát triển

**Tài liệu tham khảo**

Danh mục các tài liệu, trang web tham khảo (chú ý chỉ rõ vị trí tham khảo trong báo cáo)

**\* Phần trình bày báo chuẩn bị bằng PPT slides.**

**5. Kết luận**

5.1. Kết quả đã đạt được

1. Xây dựng được một game rắn săn mồi cơ bản

- Đã phát triển được game rắn săn mồi với đầy đủ các chức năng cơ bản

- Có giao diện dễ nhìn, dễ sử dụng

2. Tích hợp AI thành công

- Đã cài đặt được các thuật toán tìm đường cơ bản

- Tích hợp DeepLearning giúp con rắn tự thân vận động

- AI đã có khả năng tránh được các chướng ngại vật

3. Hiệu năng tốt

- Game vận hành ổn, mượt mà

- Thuật toán AI phản hồi nhanh

5.2. Định hướng phát triển

1. Cải thiện AI

- Phát triển thêm các thuật toán tìm đường hoàn hảo hơn, tối ưu hơn

- Tích hợp machine learning để AI có thể tự học từ các ván trước

- Tối ưu hoá các thuật toán hiện có

2. Mở rộng gameplay

- Thêm tính đối kháng cho game, 2 con rắn tranh thức ăn

- Thêm các chế độ chơi mới (multiplayer, battle mode)

- Tăng độ khó cho game với nhiều loại chướng ngại vật khác nhau

- Thêm các item bổ trợ như đi xuyên vật cản

3. Cải thiện giao diện

- Nâng cấp đồ hoạ game với các các hiệu ứng đẹp hơn

- Tích hợp BXH điểm

5.3. Tổng kết

Đồ án Cachephoarong đã thành công ứng dụng các thuật toán AI tìm đường cơ bản cho game rắn săn mồi. Với giao diện dễ nhìn, bắt mắt và lối chơi cổ điển giúp người chơi dễ dàng hiểu các thuật toán và đánh giá mức độ cải thiện game thông qua từng thuật toán. Mặc dù game vẫn còn nhiều sai sót nhưng về cơ bản đã hoàn thành tốt công việc truyền tải nội dung các thuật toán với người chơi và người học. Cảm ơn bạn đã đọc đến đây, chúc bạn một ngày tốt lành !!!.

LINK THAM KHẢO GUI

<https://stackoverflow.com/questions/69585685/dfs-snake-solving-algorithm-nonetype-object-has-no-len>