TRƯỜNG ĐH 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: Lập trình game Pacman sử dụng các thuật toán tìm kiếm**

Giảng viên: TS. Phan Thị Huyền Trang

**Danh sách sinh viên thực hiện**

|  |  |  |
| --- | --- | --- |
| **Mã số SV** | **Họ và tên** | **Mức độ**  **đóng góp (%)** |
| 22110138 | Nguyễn Trung Hiếu | 100% |
| 22110200 | Lê Hoàng Bảo Phúc | 100% |

*TP. Hồ Chí Minh, tháng 11 năm 2024*

# **I) Mở đầu**

## **1. Phát biểu bài toán**

Pacman là trò chơi điện tử kinh điển mà trong đó người chơi sẽ điều khiển nhân vật chính(Pacman) di chuyển trong một mê cung được tạo bởi ma trận(grid) để ăn các hạt thức ăn trong khi tránh các con ma(ghosts). Bài toán sử dụng các thuật toán tìm kiếm trong trí tuệ nhân tạo cho các con ma sao cho tối ưu hóa đường đi của chúng.

**1.1 Các thành phần chính của bài toán**

Môi trường: Bao gồm mê cung được biểu diễn dưới dạng lưới ô vuông(grid), trong đó:

* Các ô trống(‘ ‘) là các ô có thể di chuyển đồng thời là các mục tiêu mà pacman cần ăn.
* Các bức tường(‘1’) là các khu vực không thể đi qua.
* P là Pacman, nhân vật chúng ta điều khiển.
* R,S,O,P là các con ma với các màu khác nhau: Red, Sky Blue, Orange, Pink. Các đối tượng thù địch với mục tiêu truy đuổi Pacman.

Điều kiện kết thúc:

Khi tất cả các chấm trên bản đồ được ăn hết thì sẽ hoàn thành 1 cấp độ.

Hoặc khi Pacman bị ma bắt 3 lần thì sẽ game over.

Sử dụng các thuật toán tìm kiếm trong game Pacman không chỉ dừng lại ở việc các thực

## **2. Mục đích, yêu cầu thực hiện**

### **2.1 Mục đích**

* Phát triển trò chơi Pacman làm cho các con ma có thể tìm đường đi để truy đuổi Pacman.
* Sử dụng các thuật toán tìm kiểm khác nhau để so sánh hiệu quả tìm đường của các con ma.
* Áp dụng các thuật toán tìm kiếm mù và tìm kiếm có thông tin vào trò chơi.
* Đảm bảo các thực thể ma đưa ra quyết định di chuyển dựa vào vị trí của chúng và vị trí của pacman.
* Sử dụng thư viện pygame để lập trình game, thiết kế thuật toán và tối ưu hóa.

### **2.2 Yêu cầu cơ bản**

Xây dựng môi trường trò chơi:

* Mê cung được thiết kế với các ô chứa tường, chấm, pacman và các con ma.
* Xử lý logic trò chơi,
* Xử lý va chạm giữa Pacman, tường, các con ma.
* Sử dụng các thuật toán tìm kiếm mù và tìm kiếm có thông tin cho các con ma
* Khi Pacman ăn Berry, Pacman được cường hóa trong khoảng thời gian ngắn và có thể ăn ma.
* Cập nhật vị trí của Pacman cùng ma một cách liên tục.
* Cung cấp giao diện chuẩn trò chơi cho các thực thể.
* Đánh giá hiệu quả thuật toán.
* Áp dụng 1 trong các nhóm thuật toán: Tìm kiếm mù, tìm kiếm có thông tin, tìm kiếm cục bộ, quay lui và Reinforcement Learning chương trình.

## **3. Phạm vi và đối tượng**

### 3.1 Phạm vi ứng dụng

* Tạo trò chơi Pacman trong môi trường mê cung 2D.
* Triển khai các thuật toán tìm kiếm để điều hướng Ghost đến Pacman, cân nhắc các thực thể di chuyện động trong môi trường
* Áp dụng các thuật toán tìm kiếm mù, tìm kiếm có thông tin, tìm kiếm cục bộ, quay lui và Reinforcement Learning chương trình.
* Mê cung được mô phỏng dưới dạng ô vuông(grid), có các loại phần tử: tường, thức ăn, berry, ghost, ma.

# **II) Cơ sở lý thuyết dùng để thực hiện project**

## **1. Công cụ và môi trường để lập trình**

- Ngôn ngữ lập trình: Python.

- IDE lập trình: Pycharm.

## **2. Các thư viện hỗ trợ lập trình**

### **2.1 Thư viện pygame**

Pygame là thư viện dùng để phát triển trò chơi 2-D chứa các module để phát triển trò chơi 2D. Một số các chức năng có thể kể đến như:

* Xử lý đồ họa 2D: Có thể sử dụng pygame để vẽ các hình cơ bản(đường thẳng, hình vuông, hình tròn,….), hỗ trợ tải, thao tác trên hình ảnh.
* Xử lý các sự kiện từ bàn phím, chuột,….(game controller).
* Có thể xử lý âm thanh cho trò chơi.
* Quản lý FPS dễ dàng: Dễ dàng điều chỉnh tốc độ khung hình để đảm bảo trò chơi chạy mượt mà.
* Hỗ trợ xử lý các hiệu ứng, va chạm, tính toán phức tạp thường gặp trong phát triển trò chơi.

### **2.2 Thư viện os**

Là thư viện được tích hợp sẵn cung cấp các chức năng tương tác với hệ điều hành, thiên hướng về làm việc với tệp và thư mục, môi trường hệ điều hành.

Các tính năng chính:

* Xem chỉnh sửa thư mục, duyệt thư mục
* Quản lý đường dẫn
* Quản lý quy trình và môi trường

### **2.3 Thư viện heapq**

Thư viện heaq cung cấp công cụ để làm việc với heap – cấu trúc dữ liệu xử lý các phần tử theo thứ tự ưu tiên

Heap trong heapq được triển khai dưới dạng min-heap, trong đó phần tử nhỏ nhất luôn ở đầu, hỗ trợ tìm phần tử nhỏ nhất hoặc lớn nhất, thêm/xóa phần tử với độ phức tạp O(logn).

Các tính năng chính heapq:

* Xây dựng heap từ danh sách
* Chèn phần tử vào heap
* Lấy và loại bỏ phần tử nhỏ nhất
* Chuyển đổi danh sách thông thường thành heap
* Tìm k phần tử nhỏ nhất/ lớn nhất trong danh sách

# **III) Phân tích, thiết kế giải pháp**

## **1. Ý tưởng thuật toán**

Ý tưởng thuật toán để thiết kế trò chơi là thuật toán tìm kiếm mù và tìm kiếm có thông tin:

+ Tìm kiếm mù là các phương pháp tìm kiếm trong đó không có thông tin về khoảng cách hay độ ưu tiên đến trạng thái đích. Thuật toán chỉ biết trạng thái ban đầu và các hành động có thể thực hiện từ mỗi trạng thái

+ Tìm kiếm có thông tin là chiến lược tìm kiếm để sử dụng thông tin bổ sung để xác định ưu tiên các trạng thái hơặc đường đi nào gần hơn với trạng thái đích. Giúp tăng hiệu quả tìm kiếm và giảm số lượng trạng thái phải duyệt

Ưu và nhược điểm của 2 loại tìm kiếm:

|  |  |  |
| --- | --- | --- |
| Đặc điểm | Tìm kiếm mù | Tìm kiếm có thông tin |
| Thông tin | Không cần thông tin thêm | Cần có thông tin heuristic |
| Hiệu quả | Thấp do mở rộng toàn bộ không gian | Cao do giảm số trạng thái cần mở rộng |
| Độ tối ưu | Không tối ưu tùy thuật toán | Thường tìm ra giải pháp tối ưu |
| Độ phức tap triển khai | Dễ triển khai nhưng không hiệu quản cho không gian tìm kiếm lớn | Có thông tin phức tạp hơn trong cài đặt nhưng mang lại hiệu quả cao với hàm heuristic |

## **2. Phân tích các thuật toán chính**

### **2.1 Thuật toán A\***

Thuộc nhóm tìm kiếm có thông tin, tìm đường đi từ một đỉnh hiện tại đến đỉnh đích có sử dụng hàm ước lượng khoảng cách: Hàm Heuristic. Hàm heurisic dùng để đánh giá đường đi tốt nhất có thể đi. Thứ tự ưu tiên đường đi được quyết định bởi hàm Heuristic được đánh giá f(x) = g(x) + h(x). Trong đó g(x) là chi phí đường đi từ điểm xuất phát đến thời điểm hiện tại còn h(x) là hàm ước lượng chi phí từ đỉnh hiện tại đến đích f(x)( Thường là giá trị càng thấp thì độ ưu tiên càng cao)

Các bước chạy thuật toán:

Khởi tạo:

* Open: Tập các node chưa xét
* Close: Tập các trạng thái đã được duyệt
* Cost(p,q): khoảng cách từ p đến q
* g(p): khoảng cách từ trạng đầu đến trạng thái hiện tại p
* h(p): Giá trị ước lượng từ trạng thái hiện tại đến trạng thái đích
* f(p) = g(p) + h(p)

Các bước chạy thuật toán:

Bước 1:

Gán Open = {s}; Close = {}

Bước 2: while(Open !={}) #Sao cho duyệt tất cả các node

Chọn node tốt nhất trong open để lấy nó ra, nếu p là trạng thái đích thì kết thúc và in ra đường đi.

Chuyển p và Close và tạo các trạng thái q kế tiếp sau p

Nếu q đã có trong Open:

* Nếu g(q) > g(p) + Cost(p,q):
  + g(q) = g(p) + Cost(p,q)
  + f(q) = g(q) + h(q)
  + prev(q) = p # Đỉnh cha của q là p

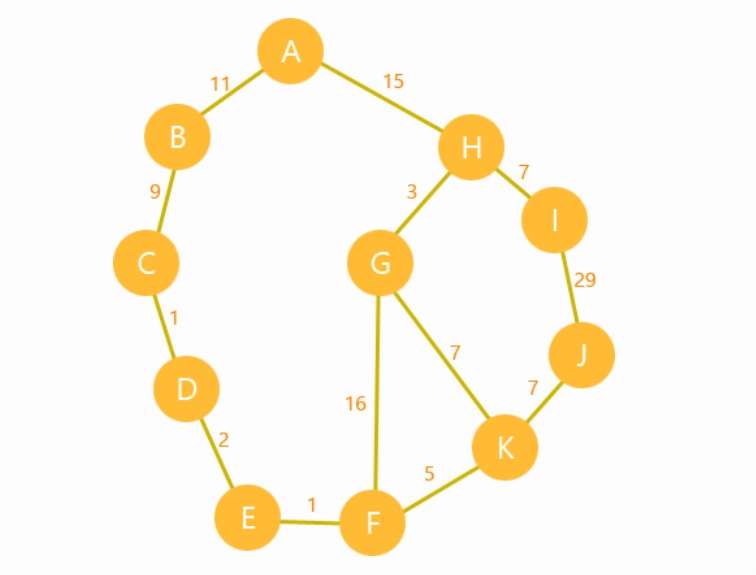
Nếu q chưa có trong Open:

* g(q) = g(p) + cost(p,q)
* f(q) = g(q) + h(q)
* prev(q) = p
* Thêm q vào Open

Nếu q đã có trong Close:

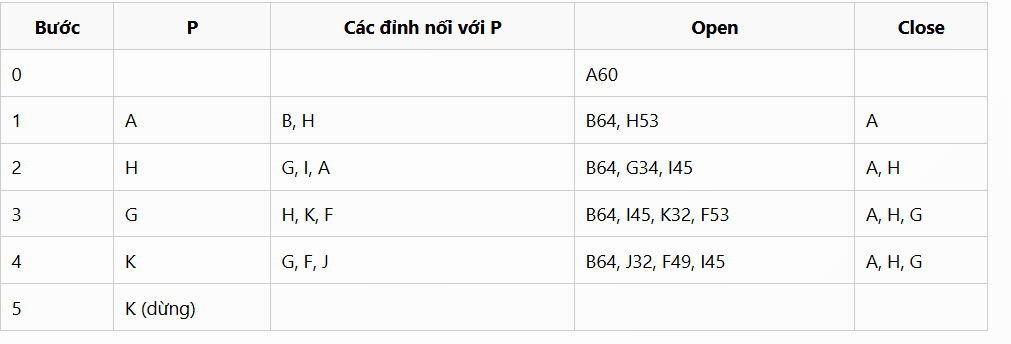
* Nếu g(q) > g(p) + Cost(p, q):
  + Bỏ q khỏi Close
  + Thêm q vào Open

Bước 3: Không tìm được.



h(A) = 60 / h(B) = 53 / h(C) = 36 / h(D) = 35 / h(E) = 35 / h(F) = 19 / h(G) = 16 / h(H) = 38 / h(I) = 23 / h(J) = 0 / h(K) = 7.

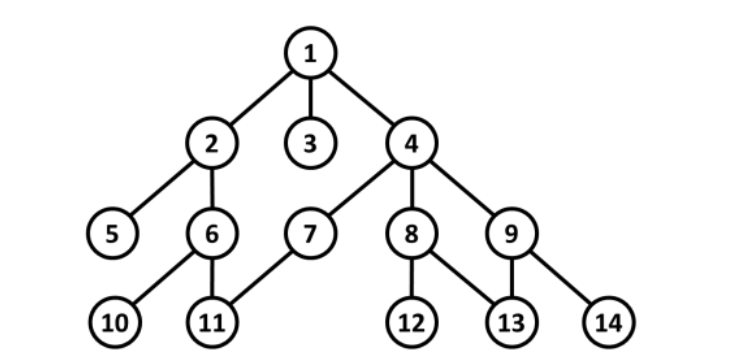
Đỉnh bắt đầu là A và kết thúc là K.



|  |
| --- |
| import heapq  def heuristic(a, b):  # Khoảng cách Manhattan  return abs(a[0] - b[0]) + abs(a[1] - b[1])  def a\_star\_search(grid, start, destination):  rows, cols = len(grid), len(grid[0])  open\_list = []  heapq.heappush(open\_list, (0, start)) # (f\_score, node)  g\_scores = {start: 0} # Chi phí từ start đến mỗi node  f\_scores = {start: heuristic(start, destination)} # f\_score = g\_score + h\_score  came\_from = {}  # Các hướng di chuyển: lên, trái, xuống, phải  directions = [(-1, 0), (0, -1), (1, 0), (0, 1)]  while open\_list:  \_, current = heapq.heappop(open\_list)  if current == destination:  path = []  while current in came\_from:  path.append(current)  current = came\_from[current]  path.append(start)  return path[::-1] # Đảo ngược lại để có đường đi từ start đến destination  for direction in directions:  next\_row, next\_col = current[0] + direction[0], current[1] + direction[1]  if 0 <= next\_row < rows and 0 <= next\_col < cols and grid[next\_row][next\_col] != '1':  next\_position = (next\_row, next\_col)  tentative\_g\_score = g\_scores[current] + 1  if next\_position not in g\_scores or tentative\_g\_score < g\_scores[next\_position]:  came\_from[next\_position] = current  g\_scores[next\_position] = tentative\_g\_score  f\_scores[next\_position] = tentative\_g\_score + heuristic(next\_position, destination)  heapq.heappush(open\_list, (f\_scores[next\_position], next\_position))  return [] |

### **2.2 Thuật toán BFS**

Thuộc nhóm Uninformed search algorithms(Tìm kiếm mù), bắt đầu ở một nút và đánh dấu tất cả các nút chính theo chính xác theo chiều rộng. Chọn một nút duy nhất và truy cập đến tất cả các nút liền kề với nút đã chọn. Mỗi khi duyệt qua thì các nút đó sẽ được đánh dấu. Thuật toán sẽ hoàn thành khi tất cả các nút của đồ thị đã được truy cập.



Các bước chạy thuật toán:

Khởi tạo: 14 node vẫn trong trạng thái chưa được đánh dấu, chọn node số 1 làm gốc, đánh dấu trước nó. Đưa node số 1(gốc) vào hàng đợi.

Duyệt:

Đưa node kề của node đầu tiên trong hàng đợi và đưa nó vào hàng đợi, kiểm tra xem nó có phải đích không. Nếu không phải thì đưa các node kề của nó vào trong hàng đợi và đánh dấu đã duyệt.

Thuật toán sẽ dừng lại khi tất cả các node đã được duyệt hoặc tìm được đường đi với kết quả trả về là đường đi đến đích hoặc là một đường đi rỗng(không có kết quả).

|  |
| --- |
| from collections import deque  def bfs\_search(grid, start, destination):  rows, cols = len(grid), len(grid[0])  queue = deque([(start, [start])])  visited = set()  visited.add(tuple(start))  directions = [(0, -1), (-1, 0), (0, 1), (1, 0)]  while queue:  (current, path) = queue.popleft()  # Kiểm tra nếu đã đến đích  if current == destination:  return path  # Thử di chuyển đến các hướng kế tiếp  for direction in directions:  next\_row, next\_col = current[0] + direction[0], current[1] + direction[1]  # Kiểm tra nếu tọa độ tiếp theo nằm trong phạm vi và không phải là tường  if 0 <= next\_row < rows and 0 <= next\_col < cols and grid[next\_row][next\_col] != '1':  next\_position = (next\_row, next\_col)  if next\_position not in visited:  visited.add(next\_position) # Đánh dấu là đã thăm  queue.append((next\_position, path + [next\_position])) # Thêm vào hàng đợi với đường đi mới  return [] # Nếu không tìm thấy đường đi, trả về danh sách rỗng |

**2.3 Thuật toán Simulated Annealing**

Là thuật toán tối ưu hóa ngẫu nhiên được lấy cảm hứng từ quá trình ủ nhiệt trong luyện kim. Thuật toán sử dụng để tìm kiếm giải pháp tốt nhất(Hoặc gần tốt nhất) cho các bài toán tối ưu hóa. Đặc biệt là trong không gian tìm kiếm lớn với nhiều cực trị cục bộ.

Nguyên lý hoạt động của Simulate Annealing:

- Khám giá không gian nghiệm: Trong tập hợp tất cả các giải pháp, tìm được nghiệm tốt nhất với giá trị tối ưu của hàm mục tiêu: Bắt đầu với một nghiệm ngẫu nhiên xi trong không gian nghiệm, tạo nghiệm lân cận xj từ nghiệm xi bằng các thay đổi( ví dụ như cộng một số ngẫu nhiên nhỏ vào các biến)

- Đánh giá nghiệm mới:

+ Tính hàm mục tiêu f(x) của nghiệm hiện tại xcurrent và nghiệm mới xnew

+ Tính mức độ cải thiện của 2 nghiệm:



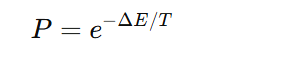
Nếu ΔE<0 : Nghiệm mới tốt hơn.

Nếu ΔE>0: Nghiệm mới tệ hơn.

- Quyết định chấp nhận nghiệm mới: Simulated Annealing sử dụng cơ chế ngẫu nhiên để thoát khỏi cực trị cục bộ:

+ Nếu nghiệm mới tốt hơn, nó sẽ luôn được chấp nhận

+ Nếu nghiệm mới tệ hơn, nó được chấp nhận với xác suất:



T: Nhiệt độ hiện tại( Dùng để kiểm soát mức độ chấp nhận nghiệm xấu)

e−ΔE/T Khi T lớn, xác xuất gần 1 – nghiệm xấu dễ được chấp nhận, khi T giảm, xác suất sẽ giảm và nghiệm xấu khó được chấp nhận hơn.

+ Ý nghĩa: cho phép thuật toán nhảy ra khỏi cực trị cục bộ và tìm kiếm thêm trong không gian nghiệm.

4. Giảm nhiệt độ

Giảm nhiệt độ để làm nguội:

+ Ở nhiệt độ cao, thuật toán dễ chấp nhận cả nghiệm xấu, giúp nó khám phá không gian nghiệm rộng hơn

+ Ở nhiệt độ thấp, thuật toán chỉ chấp nhận nghiệm tốt, tập trung tinh chỉnh nghiệm xung quanh tốt nhất đã tìm được

Công thức giảm nhiệt độ



α: Hệ số làm nguội.

Thuật toán sẽ dừng lại ghi Nhiệt độ quá thấp hoặc đã đạt số bước tối đa cho phép.

|  |
| --- |
| import math  import random  def manhattan\_distance(pos1, pos2):  return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])  def simulated\_annealing(grid, ghost\_position, pacman\_position, max\_steps=500, initial\_temp=200, cooling\_rate=0.9999999):  rows, cols = len(grid), len(grid[0])  directions = [(0, -1), (-1, 0), (0, 1), (1, 0)] # Trái, Lên, Phải, Xuống  path = [] # Lưu đường đi của Ghost  visited = set() # Lưu các vị trí đã thăm  steps = 0  temp = initial\_temp # Bắt đầu với nhiệt độ cao  # Đánh dấu vị trí ban đầu là đã thăm  visited.add(ghost\_position)  while ghost\_position != pacman\_position and steps < max\_steps:  path.append(ghost\_position) # Lưu vị trí hiện tại vào đường đi  best\_move = None  min\_distance = math.inf  possible\_moves = [] # Lưu các bước di chuyển có khoảng cách Manhattan nhỏ nhất  # Xét tất cả các hướng di chuyển  for direction in directions:  next\_row = ghost\_position[0] + direction[0]  next\_col = ghost\_position[1] + direction[1]  # Kiểm tra tính hợp lệ của ô tiếp theo  if (  0 <= next\_row < rows  and 0 <= next\_col < cols  and grid[next\_row][next\_col] != '1' # Không phải chướng ngại vật  and (next\_row, next\_col) not in visited # Chưa thăm  ):  next\_position = (next\_row, next\_col)  distance = manhattan\_distance(next\_position, pacman\_position)  # Thêm các bước di chuyển có khoảng cách Manhattan nhỏ nhất  if distance < min\_distance:  min\_distance = distance  best\_move = next\_position  possible\_moves = [next\_position] # Reset list nếu tìm thấy bước di chuyển tốt nhất  elif distance == min\_distance:  possible\_moves.append(next\_position)  # Nếu không tìm thấy bước đi nào hợp lý, thoát khỏi vòng lặp  if not best\_move:  break  # Chọn bước đi ngẫu nhiên nếu có nhiều lựa chọn  next\_move = random.choice(possible\_moves)  next\_distance = manhattan\_distance(next\_move, pacman\_position)  delta = next\_distance - min\_distance  # Quyết định có chấp nhận bước đi hay không dựa trên nhiệt độ và delta  if delta < 0: # Nếu bước đi mới tốt hơn (khoảng cách giảm)  best\_move = next\_move  else: # Nếu bước đi mới tồi hơn, chấp nhận với xác suất P = exp(-delta / temp)  probability = math.exp(-delta / temp)  if random.random() < probability:  best\_move = next\_move  # Di chuyển Ghost đến bước đi đã chọn  ghost\_position = best\_move  steps += 1  # Đánh dấu vị trí đã đi qua  visited.add(ghost\_position)  # Giảm nhiệt độ (làm lạnh) từ từ nhưng không quá nhanh  temp \*= cooling\_rate  # Nếu nhiệt độ quá thấp, có thể thoát khỏi vòng lặp  if temp < 0.01:  break  # Kiểm tra nếu Ghost đã tìm thấy Pacman  if ghost\_position == pacman\_position:  path.append(pacman\_position)  return path |

**2.4 Thuật toán Backtracking AC3**

**2.4.1 Thuật toán quay lui**

Backtracking(quay lui) là kỹ thuật tìm kiếm giải pháp theo từng bước bằng cách thử các đường đi khác nhau và quay lại nếu chúng dẫn đến ngõ cụt, thuật toán thường được sử dụng trong các vấn đề cần khám phá nhiều khả năng để giải quyết một vấn đề( Tìm đường mê cung, Sudoku). Khi đi đến ngõ cụt, thuật toán sẽ quay lại điểm quyết định trước đó và khám phá một con đường khác cho đến khi tìm ra giải pháp hoặc tất cả các khả năng đã được khai thác hết.

Nguyên lý hoạt động của thuật toán quay lui: Hoạt động bằng cách sử dụng đệ quy khám phá tất cả giải pháp có thể thực hiện cho vấn đề. Chọn giải pháp ban đầu và khám phá tất cả phần mở rộng của giải pháp đó, nếu trả về kết quả thì dừng. Nếu không thì thuật toán quay lại và thử phần mở rộng khác.

Thuật toán quay lui:

B1: Chọn giải pháp ban đầu.

B2: Khám phá tất cả các khả năng mở rộng của giải pháp hiện tại.

B3: Nếu phần mở rộng dẫn đến kết quả, kết thúc.

B4: Nếu phần mở rộng không đưa đến giải pháp, quay lại và thử phần mở rộng khác

B5: Lặp lại bước 2-4 cho đến khi tìm được tất cả giải pháp khả thi.

**2.4.2 Thuật toán AC3**

Thuật toán AC3(Arc Consistency 3) là thuật toán để giải quyết bài toán ràng buộc(Constraint Satisfaction Problems – CSP). Thuật toán đảm bảo miền giá trị của các biến trong bài toán ràng buộc được kiểm tra và thu hẹp lại sao cho các cặp ràng buộc thỏa mãn ràng buộc. AC3 đảm bảo rằng không tồn tại giá trị nào trong miền của một biến mà không thể thỏa mãn được ràng buộc với miền của một biến khác liên quan.

Nguyên lý hoạt động của AC3:

Giả sử chúng ta có:

X : Tập hợp các biến

D(x): Miền giá trị của biến x thuộc X

C(x,y): Tập hợp ràng buộc giữa hai biến x và y

Các bước thực hiện:

1. Khởi tạo hàng đợi các arc: Đưa tất cả cặp ràng buộc (x,y) vào hàng đợi Q

2. Xử lý các arc trong hàng đợi:

+ Lấy arc(x, y) từ hàng đợi

+ Kiểm tra tính nhất quán(arc-consistency) giữa x và y: Loại bỏ các giá trị trong D(x) không thỏa mãn ràng buộc với bất kì giá trị nào trong D(y).

+ Nếu miền giá trị D(x) bị thay đổi: Thêm tất cả các arc liên quan đến x(ngoại trừ (x, y)) vào hàng đợi để kiểm tra lại

B3: kết thúc

Lặp lại quá trình cho đến khi hàng đợi rỗng

Nếu một miền D(x) rỗng, bài toán không có lời giải

Nếu tất cả miền giá trị đều còn lại ít nhất một giá trị, bài toán đạt được tính arc-consistent

|  |
| --- |
| def is\_valid(grid, row, col, path):  rows, cols = len(grid), len(grid[0])  return 0 <= row < rows and 0 <= col < cols and grid[row][col] != '1' and (row, col) not in path  # AC-3 để làm sạch không gian tìm kiếm trước khi thực hiện backtracking  # Hàm AC-3 tối ưu hóa chỉ cần cập nhật khi có thay đổi  def ac3(grid, start, destination):  rows, cols = len(grid), len(grid[0])  directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Các hướng di chuyển: lên, xuống, trái, phải  possible\_moves = {(r, c): [] for r in range(rows) for c in range(cols)}  # Kiểm tra các ô hợp lệ có thể di chuyển từ mỗi ô  for r in range(rows):  for c in range(cols):  if grid[r][c] != '1': # Không phải tường  for direction in directions:  next\_r, next\_c = r + direction[0], c + direction[1]  if 0 <= next\_r < rows and 0 <= next\_c < cols and grid[next\_r][next\_c] != '1':  possible\_moves[(r, c)].append((next\_r, next\_c))  return possible\_moves  # Hàm backtracking tìm đường đi từ vị trí của Ghost đến Pacman và bao vây Pacman  def backtrack(grid, ghost\_position, pacman\_position, possible\_moves, max\_depth=100):  rows, cols = len(grid), len(grid[0])  def search(path, depth):  if depth > max\_depth: # Dừng lại nếu vượt quá độ sâu tối đa  print(f"Max depth reached at position {path[-1]}")  return None  current = path[-1] # Vị trí hiện tại  print(f"Current position of Pink Ghost: {current}") # In ra vị trí hiện tại của Ghost Pink    if current == pacman\_position:  print("Pacman reached!")  return path  # Duyệt qua các hướng di chuyển hợp lệ từ vị trí hiện tại  possible\_next\_steps = possible\_moves[current]  print(f"Possible moves from {current}: {possible\_next\_steps}")  for next\_row, next\_col in possible\_next\_steps:  if (next\_row, next\_col) not in path: # Kiểm tra nếu điểm chưa được duyệt  # Tìm kiếm tiếp từ điểm mới  result = search(path + [(next\_row, next\_col)], depth + 1)  if result: # Nếu tìm được đường đi hợp lệ  return result  print(f"No valid moves from {current}. Backtracking...") # In ra khi không còn hướng di chuyển hợp lệ  return None # Nếu không tìm thấy đường đi hợp lệ  return search([ghost\_position], 0) |

**5. Reinforcement Learning**

Reinforcement Learning(Học tăng cường): Là kỹ thuật máy học đào tạo chương trình để đưa ra quyết định nhằm thu về kết quả tối ưu nhất. Kỹ thuật bắt chước quy trình học thử và gặp lỗi sai mà con người sử dụng để đạt được mục tiêu để đạt được mục tiêu đã được đặt ra. Học tăng cường giúp phần mềm tập trung vào các hành động hướng tới mục tiêu, đồng thời bỏ qua các hành động làm xao lãng mục tiêu.

5.1 Nguyên lý hoạt động:

Các khái niệm chính:

Tác tử: Thuật toán Machine Learning.

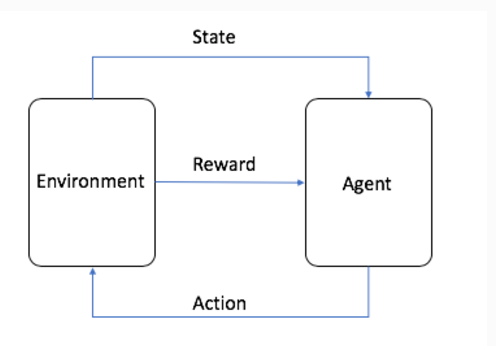
Môi trường: Không gian vấn đề thích ứng với các thuộc tính: Biến, giá trị ranh giới, quy tắc, hành động hợp lệ.

Hành động: Một bước mà tác tử RL thực hiện để điều hướng môi trường

Trạng thái: Là môi trường tại một thời điểm nhất định

Điểm: Là giá trị âm, dương hoặc không – được tính từ việc trừ(cho hình phạt) và cộng( cho phần thưởng) cho việc thực hiện một hành động

Phần thưởng tích lũy: tổng của các phần thưởng hoặc giá trị cuối cùng



# **IV) Thực nghiệm, đánh giá, phân tích kết quả**

## 1. Cài đặt khởi tạo chương trình

## 2. Kết quả thực nghiệm

# **V) Kết luận**

# **Tài liệu tham khảo**

<https://www.iostream.co/article/thuat-giai-a-DVnHj>

<https://users.soict.hust.edu.vn/huonglt/AI/Chuong%203.%20Tim%20kiem%20co%20ban.pdf>

<https://wiki.vnoi.info/algo/graph-theory/breadth-first-search.md>

<https://www.pygame.org/docs/>