**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HỒ CHÍ MINH KHOA CÔNG NGHỆ THÔNG TIN**

****

**BÁO CÁO CUỐI KỲ**

**Đề tài: ỨNG DỤNG CÁC THUẬT TOÁN TRÍ TUỆ NHÂN TẠO TRONG TRÒ CHƠI PACMAN**

Giảng viên hướng dẫn : Phan Thị Huyền Trang

Sinh viên thực hiện:

Nguyễn Minh Quốc Khánh MSSV 23110113

Nguyễn Hưng Nguyên MSSV 23110135

Nguyễn Bách Tùng MSSV 23110166

Lớp: 23110CLC3

Mã lớp: 251ARIN330585\_05CLC

***Thành phố Hồ Chí Minh, tháng 10 năm 2025***

DANH SÁCH THAM GIA ĐỀ TÀI

HỌC KÌ I, NĂM HỌC: 2025 – 2026

Lớp: 251ARIN330585\_05CLC

**Tên đề tài**: **ỨNG DỤNG CÁC THUẬT TOÁN TRÍ TUỆ NHÂN TẠO TRONG TRÒ CHƠI PACMAN**

|  |  |
| --- | --- |
| HỌ VÀ TÊN SINH VIÊN | MÃ SỐ SINH VIÊN |
| Nguyễn Minh Quốc Khánh | 23110113 |
| Nguyễn Hưng Nguyên | 23110135 |
| Nguyễn Bách Tùng | 23110166 |

Nhận xét của giảng viên

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

......................................................................................................................................................

Ngày 23 tháng 10 năm 2025

Giảng viên chấm điểm

**MỤC LỤC**

[PHẦN MỞ ĐẦU 1](#_Toc211246776)

[1. Lý do chọn đề tài 1](#_Toc211246777)

[2. Phạm vi và giới hạn của đề tài 2](#_Toc211246778)

[CHƯƠNG 1: TỒNG QUAN VỀ THUẬT TOÁN TÌM KIẾM 2](#_Toc211246779)

[1.1. Giải quyết vấn đề bằng tìm kiếm 2](#_Toc211246780)

[1.2. Phân loại các thuật toán tìm kiếm không có thông tin 3](#_Toc211246781)

[1.2.1. BFS (Breadth-First Search) 3](#_Toc211246782)

[1.2.2. DFS (Depth-First Search) 4](#_Toc211246783)

[1.2.3 UCS (Uniform Cost Search) 6](#_Toc211246784)

[1.2.4. IDS (Iterative Deepening Search) 7](#_Toc211246785)

[1.2.5. IDL (Iterative Deepening Learning) 9](#_Toc211246786)

[1.3. Phân loại các thuật toán tìm kiếm có thông tin 12](#_Toc211246787)

[1.3.1. Greedy Best-First Search 12](#_Toc211246788)

[1.3.2. Thuật toán A\* (A-Star Search) 14](#_Toc211246789)

[1.4. Phân loại các tìm kiếm cục bộ 17](#_Toc211246790)

[1.4.1. Hill-Climbing Search 17](#_Toc211246791)

[1.4.2. Simulated Annealing Search 19](#_Toc211246792)

[1.4.3. Beam Search 21](#_Toc211246793)

[1.4.4. Genetic Algorithm (Giải thuật di truyền) 23](#_Toc211246794)

[1.5. Phân loại các thuật toán tìm kiếm đối kháng 25](#_Toc211246795)

[1.5.1. Minimax Algorithm 25](#_Toc211246796)

[1.5.2. Alpha–Beta Pruning 27](#_Toc211246797)

[1.5.3. Expectimax Algorithm 29](#_Toc211246798)

[1.6. Phân loại các thuật toán tìm kiếm có thông tin 32](#_Toc211246799)

[1.6.1. Backtracking Search 32](#_Toc211246800)

[1.6.2. Forward Checking 34](#_Toc211246801)

[1.6.3. AC-3 (Arc Consistency Algorithm 3) 36](#_Toc211246802)

[1.6.4. And-Or Planning 39](#_Toc211246803)

[CHƯƠNG 2: THIẾT KẾ HỆ THỐNG VÀ GIAO DIỆN CỦA CHƯƠNG TRÌNH 41](#_Toc211246804)

[2.1. Cấu trúc tổng thể của chương trình 41](#_Toc211246805)

[2.2. Thiết kế giao diện người dùng 42](#_Toc211246806)

[2.2.1. giao diện chính của trò chơi 42](#_Toc211246807)

[2.2.2. Giao diện Menu chọn thuật toán AI 43](#_Toc211246808)

[2.2.3. Giao diện khi Pacman ăn chấm hoặc bị bắt 44](#_Toc211246809)

[KẾT LUẬN 45](#_Toc211246810)

[TÀI LIỆU THAM KHẢO 47](#_Toc211246811)

# 

# **PHẦN MỞ ĐẦU**

## **1. Lý do chọn đề tài**

Trí tuệ nhân tạo (AI) là một trong những lĩnh vực phát triển nhanh nhất của khoa học máy tính hiện nay, với mục tiêu giúp máy tính có khả năng tư duy, học hỏi và ra quyết định thông minh như con người. Trong đó, bài toán tìm kiếm là một trong những nền tảng cơ bản và quan trọng nhất, được ứng dụng trong rất nhiều lĩnh vực như robot tự hành, trò chơi điện tử, hệ thống gợi ý hay tối ưu hóa đường đi.

Khi quan sát các ứng dụng thực tế, có thể nhận thấy rằng phần lớn các vấn đề trong AI đều có thể biểu diễn dưới dạng bài toán tìm kiếm đường đi hoặc trạng thái tối ưu. Ví dụ như:

* Trong trò chơi, mỗi lượt di chuyển của nhân vật có thể được xem như việc tìm kiếm nước đi tối ưu trong không gian các trạng thái hợp lệ.
* Trong robot di chuyển, bài toán tìm đường đi ngắn nhất tránh vật cản chính là một dạng bài toán tìm kiếm có ràng buộc.

Từ ý tưởng đó, nhóm em đã xây dựng một trò chơi Pacman AI – nơi nhân vật Pacman có thể tự động điều hướng, ăn điểm, né tránh các Ghost và ra quyết định dựa trên thuật toán trí tuệ nhân tạo. Trò chơi tích hợp 17 thuật toán AI khác nhau, bao gồm cả các thuật toán tìm kiếm không có thông tin (BFS, DFS, UCS, IDS), \**thuật toán có thông tin (Greedy, A*)\*\*, thuật toán tối ưu cục bộ (Hill-Climbing, Simulated Annealing, Beam Search), thuật toán ra quyết định (Minimax, Alpha-Beta, Expectimax, And-Or) và thuật toán ràng buộc (CSP: Backtracking, Forward Checking, AC-3).

Việc áp dụng đa dạng các thuật toán này không chỉ giúp Pacman có thể thay đổi hành vi thông minh linh hoạt tùy theo chiến lược, mà còn giúp người học hiểu rõ cách hoạt động, ưu nhược điểm và phạm vi ứng dụng của từng thuật toán trong thực tế.

Nhận thấy tầm quan trọng của các thuật toán tìm kiếm trong AI cũng như tính ứng dụng trực quan khi thể hiện qua trò chơi, em đã chọn đề tài: **“Ứng dụng các thuật toán trí tuệ nhân tạo trong trò chơi Pacman”** để nghiên cứu, thực hành và minh họa sinh động mối liên hệ giữa lý thuyết AI và ứng dụng thực tế.

## **2. Phạm vi và giới hạn của đề tài**

Đề tài tập trung nghiên cứu, cài đặt và minh họa **các thuật toán trí tuệ nhân tạo trong trò chơi Pacman**, qua đó giúp hiểu rõ cơ chế hoạt động của từng thuật toán trong môi trường mô phỏng trực quan. Cụ thể, chương trình tích hợp **17 thuật toán AI** thuộc nhiều nhóm khác nhau như tìm kiếm không có thông tin (BFS, DFS, UCS, IDS, DLS), tìm kiếm có thông tin (Greedy, A\*), tối ưu cục bộ (Hill-Climbing, Simulated Annealing, Beam Search, Genetic) và ra quyết định hoặc ràng buộc (Minimax, Alpha-Beta, Expectimax, And-Or Planning, Backtracking, Forward Checking, AC-3). Toàn bộ hệ thống được xây dựng bằng **ngôn ngữ Python** và **thư viện Pygame**, cho phép Pacman tự động di chuyển, ăn điểm, né tránh Ghost, chuyển cấp độ và thay đổi thuật toán trong quá trình chơi để người học có thể trực quan so sánh hiệu quả giữa các phương pháp.

Tuy nhiên, đề tài vẫn có một số giới hạn nhất định. Trò chơi được xây dựng trong môi trường 2D đơn giản nên chưa thể hiện đầy đủ tính phức tạp của các môi trường thực tế hoặc 3D. Các thuật toán được cài đặt ở mức cơ bản, chưa tối ưu về hiệu năng và chưa mở rộng sang các lĩnh vực học sâu hay học tăng cường. Ngoài ra, đồ họa và âm thanh vẫn còn đơn giản, chủ yếu phục vụ mục đích **học tập, minh họa và thực hành thuật toán AI**, chưa hướng tới thương mại hóa hoặc phát triển thành sản phẩm trò chơi hoàn chỉnh.

# **CHƯƠNG 1: TỒNG QUAN VỀ THUẬT TOÁN TÌM KIẾM**

## **1.1. Giải quyết vấn đề bằng tìm kiếm**

Trong một không gian trạng thái của một bài toán, ta có thể xây dựng một cấu trúc đồ thị với các nút là các trạng thái của bài toán, các cung là phép chuyển trạng thái. Đồ thị này gọi là không gian trạng thái của bài toán.

Tìm kiếm trong không gian trạng thái là phương pháp được dùng phổ biến trong một lớp lớn các bài toán và lớp kỹ thuật để giải quyết những vấn đề quan trọng, được nghiên cứu và ứng dụng nhiều trong trí tuệ nhân tạo.

Các giải thuật tìm kiếm được chia thành 5 nhóm:

1. Tìm kiếm mù (không có thông tin)
2. Tìm kiếm heuristics (có thông tin)
3. Tìm kiếm cục bộ
4. Tìm kiếm đối kháng
5. Giải bài toán ràng buộc và lập kế hoạch

## **1.2. Phân loại các thuật toán tìm kiếm không có thông tin**

Các thuật toán tìm kiếm không có thông tin hoạt động không dùng hàm heuristic để định hướng; chúng chỉ dựa trên cấu trúc không gian trạng thái và chiến lược duyệt (chiều rộng, chiều sâu, chi phí, v.v.). Trong Pacman AI, nhóm thuật toán này thích hợp để tìm đường cơ bản, đảm bảo khám phá toàn bộ không gian đến một giới hạn nào đó hoặc tìm đường ngắn nhất trong trường hợp các bước có chi phí đồng nhất.

### **1.2.1. BFS (Breadth-First Search)**

Mô tả ý tưởng: BFS duyệt không gian trạng thái theo từng mức độ (độ sâu) từ nút bắt đầu: trước hết mở rộng mọi nút ở độ sâu 0, rồi mọi nút ở độ sâu 1, rồi 2, ... Cho đến khi tìm được nút mục tiêu.

Ứng dụng trong Pacman AI: Tìm đường ngắn nhất (theo số bước) từ vị trí hiện tại tới một dot hoặc vị trí đích trong mê cung khi mọi bước di chuyển có chi phí bằng nhau. Phù hợp để tìm dot gần nhất hoặc kiểm tra khả năng tiếp cận.

Tính chất:

* Hoàn chỉnh (Complete): Có, nếu không gian trạng thái hữu hạn hoặc nếu có mức sâu hữu hạn tới mục tiêu.
* Tối ưu (Optimal): Có khi chi phí mỗi bước là bằng nhau (unit cost).
* Độ phức tạp thời gian: O(b^d) — b là branching factor (số hành động khả dĩ), d là độ sâu của lời giải.
* Độ phức tạp bộ nhớ: O(b^d) — vì lưu toàn bộ frontier/level hiện tại.

Ưu/nhược điểm:

* Ưu: Tìm được đường ngắn nhất (với unit cost). Dễ triển khai.
* Nhược: Tiêu tốn nhiều bộ nhớ khi độ sâu lời giải lớn.

Thuật toán BFS viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def bfs\_move(self, maze):          """BFS: Breadth-First Search - Tìm viên chấm gần nhất"""          queue = deque([(self.x, self.y, [])])          visited = set()          while queue:              x, y, path = queue.popleft()              if (x, y) in visited:                  continue              visited.add((x, y))              # Kiểm tra nếu tìm thấy dot              if maze[y][x] == CellType.DOT:                  if path:                      return path[0]                  else:                      return self.first\_valid\_move(maze, [])              # Thêm các hướng có thể đi              for direction in Direction:                  dx, dy = direction.value                  new\_x, new\_y = x + dx, y + dy                  if (0 <= new\_x < MAZE\_WIDTH and 0 <= new\_y < MAZE\_HEIGHT and                      maze[new\_y][new\_x] != CellType.WALL and                      (new\_x, new\_y) not in visited):                      queue.append((new\_x, new\_y, path + [direction]))          return self.first\_valid\_move(maze, []) |

### **1.2.2. DFS (Depth-First Search)**

Mô tả ý tưởng: DFS đi sâu vào một nhánh cho tới khi đạt đến nút lá hoặc không thể đi tiếp, rồi quay lui (backtrack) để thử nhánh khác. Thường dùng ngăn xếp (stack) - có thể là đệ quy.

Ứng dụng trong Pacman AI: Thích hợp cho việc khám phá nhanh một đường dẫn (không đảm bảo ngắn nhất). Có thể dùng khi bộ nhớ hạn chế hoặc khi cần tìm một đường bất kỳ đến mục tiêu để thực thi nhanh.

Tính chất:

* Hoàn chỉnh: Không đảm bảo trên không gian vô hạn; có thể vô tận nếu tồn tại vòng lặp và không có kiểm soát. Có thể làm đầy đủ nếu dùng phiên bản có kiểm soát visited và không có vòng lặp.
* Tối ưu: Không (không đảm bảo ngắn nhất).
* Độ phức tạp thời gian: O(b^m) — m là độ sâu tối đa của cây tìm kiếm.
* Độ phức tạp bộ nhớ: O(b·m) cho phiên bản theo ngăn xếp (chỉ lưu đường dẫn hiện tại).

Ưu/nhược điểm:

* Ưu: Dùng ít bộ nhớ hơn BFS (khi triển khai theo ngăn xếp). Có thể tìm nhanh một giải pháp nếu nhánh đúng nằm sâu.
* Nhược: Có thể mắc kẹt trong nhánh sâu vô hạn hoặc vòng lặp; không đảm bảo tìm đường ngắn nhất.

Thuật toán DFS viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def dfs\_move(self, maze):          """DFS: Depth-First Search"""          stack = [(self.x, self.y, [])]          visited = set()            while stack:              x, y, path = stack.pop()                if (x, y) in visited:                  continue              visited.add((x, y))                # Kiểm tra nếu tìm thấy dot              if maze[y][x] == CellType.DOT:                  if path:                      return path[0]                  else:                      return self.first\_valid\_move(maze, [])                # Thêm các hướng              for direction in Direction:                  dx, dy = direction.value                  new\_x, new\_y = x + dx, y + dy                    if (0 <= new\_x < MAZE\_WIDTH and 0 <= new\_y < MAZE\_HEIGHT and                      maze[new\_y][new\_x] != CellType.WALL and                      (new\_x, new\_y) not in visited):                      stack.append((new\_x, new\_y, path + [direction]))            return self.first\_valid\_move(maze, []) |

### **1.2.3 UCS (Uniform Cost Search)**

Mô tả ý tưởng: UCS mở rộng nút có chi phí từ gốc đến nút nhỏ nhất (theo g(n)). Dùng hàng đợi ưu tiên (priority queue) theo chi phí g. Thích hợp khi bước di chuyển có chi phí khác nhau.

Ứng dụng trong Pacman AI: Tìm đường có chi phí nhỏ nhất khi các ô/điểm di chuyển có chi phí khác nhau (ví dụ: ô bình thường, ô chứa fruit, ô có penalty). Dùng khi muốn tối ưu tổng chi phí (không chỉ số bước).

Tính chất:

* Hoàn chỉnh: Có, nếu chi phí bước >= ε > 0 (không có chi phí 0 gây vòng lặp vô hạn).
* Tối ưu: Có (trả về đường có tổng chi phí nhỏ nhất).
* Độ phức tạp thời gian: O(b^{⌈C\*/ε⌉}) — phức tạp phụ thuộc vào chi phí tối ưu C\* và ε (chi phí nhỏ nhất).
* Độ phức tạp bộ nhớ: O(b^{⌈C\*/ε⌉}) — lưu frontier lớn.

Ưu/nhược điểm:

* Ưu: Tối ưu theo chi phí, phù hợp bài toán có trọng số.
* Nhược: Tốn bộ nhớ và thời gian với chi phí tổng lớn.

Thuật toán UCS viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def ucs\_move(self, maze):          """UCS: Uniform Cost Search - Đường đi tối ưu với chi phí"""          counter = 0          heap = [(0, counter, self.x, self.y, [])]          visited = set()          while heap:              cost, \_, x, y, path = heapq.heappop(heap)              counter += 1              if (x, y) in visited:                  continue              visited.add((x, y))              # Kiểm tra nếu tìm thấy dot              if maze[y][x] == CellType.DOT:                  if path:                      return path[0]                  else:                      return self.first\_valid\_move(maze, [])              # Thêm các hướng với chi phí              for direction in Direction:                  dx, dy = direction.value                  new\_x, new\_y = x + dx, y + dy                  if (0 <= new\_x < MAZE\_WIDTH and 0 <= new\_y < MAZE\_HEIGHT and                      maze[new\_y][new\_x] != CellType.WALL and                      (new\_x, new\_y) not in visited):                      # Chi phí dựa trên loại cell                      cell\_cost = 1                      if maze[new\_y][new\_x] == CellType.FRUIT:                          cell\_cost = 0.5  # Ưu tiên hoa quả                      elif maze[new\_y][new\_x] == CellType.POWER\_PELLET:                          cell\_cost = 0.3  # Ưu tiên power pellet                      heapq.heappush(heap, (cost + cell\_cost, counter, new\_x, new\_y, path + [direction]))                      counter += 1          return self.first\_valid\_move(maze, []) |

### **1.2.4. IDS (Iterative Deepening Search)**

Mô tả ý tưởng: IDS kết hợp ưu điểm của BFS (tối ưu theo độ sâu nhỏ nhất) và DFS (bộ nhớ thấp): thực hiện DLS (Depth-Limited Search) nhiều lần, tăng dần giới hạn độ sâu (limit) từ 0,1,2,... tới khi tìm được mục tiêu. Mỗi lần là một DFS có giới hạn.

Ứng dụng trong Pacman AI: Khi không biết trước độ sâu của lời giải nhưng muốn tiết kiệm bộ nhớ và vẫn có ưu thế tìm lời giải nông trước. Thích hợp cho mê cung có branching factor ổn định và khi muốn đảm bảo tìm được đường có ít bước nhất nhưng không muốn chi phí bộ nhớ của BFS.

Tính chất:

* Hoàn chỉnh: Có, nếu mỗi lần giới hạn tăng dần và không gian trạng thái hữu hạn.
* Tối ưu: Có (về số bước) nếu các bước có chi phí đồng nhất và ta tăng limit theo thứ tự tăng dần.
* Độ phức tạp thời gian: O(b^d) — có overhead do lặp lại nhiều lần; tuy nhiên tổng chi phí vẫn thuộc cùng b^d lớp.
* Độ phức tạp bộ nhớ: O(b·d) — như DFS giới hạn.

Ưu/nhược điểm:

* Ưu: Kết hợp ưu điểm của DFS (bộ nhớ thấp) và BFS (tìm lời giải nông trước).
* Nhược: Thời gian có overhead do nhiều lần khám phá lại các nút nông.

Thuật toán IDS viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def ids\_move(self, maze):          """IDS: Iterative Deepening Search"""          for depth in range(1, 10):  # Giới hạn depth              result = self.dfs\_move\_with\_depth(maze, depth)              if result:                  return result          return self.first\_valid\_move(maze, [])        def dfs\_move\_with\_depth(self, maze, max\_depth):          """DFS với giới hạn depth"""          stack = [(self.x, self.y, [], 0)]          visited = set()            while stack:              x, y, path, depth = stack.pop()                if depth >= max\_depth:                  continue                if (x, y) in visited:                  continue              visited.add((x, y))                # Kiểm tra nếu tìm thấy dot              if maze[y][x] == CellType.DOT:                  if path:                      return path[0]                  else:                      return self.first\_valid\_move(maze, [])                # Thêm các hướng              for direction in Direction:                  dx, dy = direction.value                  new\_x, new\_y = x + dx, y + dy                    if (0 <= new\_x < MAZE\_WIDTH and 0 <= new\_y < MAZE\_HEIGHT and                      maze[new\_y][new\_x] != CellType.WALL and                      (new\_x, new\_y) not in visited):                      stack.append((new\_x, new\_y, path + [direction], depth + 1))            return None |

### **1.2.5. IDL (Iterative Deepening Learning)**

Mô tả ý tưởng: IDL (Iterative Deepening Learning) là sự kết hợp giữa thuật toán tìm kiếm sâu dần (Iterative Deepening Search - IDS) và cơ chế học hỏi (Learning).  
Khác với IDS, vốn chỉ lặp lại tìm kiếm từ độ sâu 0 đến khi tìm được lời giải, IDL cho phép lưu lại thông tin từ các lần tìm kiếm trước, chẳng hạn như:

* các nút đã mở rộng,
* chi phí ước lượng của các trạng thái,
* hướng di chuyển có triển vọng nhất.

Nhờ đó, ở những lần lặp sau, thuật toán không cần tìm lại từ đầu hoàn toàn, giúp giảm thời gian tìm kiếm và cải thiện hiệu suất.

Ứng dụng trong Pacman AI: Trong trò chơi Pacman AI, IDL được sử dụng để Pacman dần học cách tìm đường hiệu quả hơn qua nhiều ván chơi hoặc lần di chuyển khác nhau.  
Ví dụ: Khi Pacman liên tục gặp các tình huống tương tự (mê cung có cùng cấu trúc, vị trí Ghost tương tự), thuật toán IDL sẽ sử dụng thông tin từ lần tìm kiếm trước (như hướng tốt nhất hoặc chi phí ngắn nhất) để tăng tốc độ ra quyết định cho các lần sau.

Tính chất:

* Hoàn chỉnh: Có, nếu không gian trạng thái hữu hạn và có thể đến được mục tiêu.
* Tối ưu: Có thể đạt gần tối ưu, phụ thuộc vào chiến lược cập nhật kiến thức học được.
* Độ phức tạp thời gian: O(b^d) trong lần đầu, nhưng giảm dần qua các lần lặp nhờ học từ lịch sử.
* Độ phức tạp bộ nhớ: O(b·d) — tương tự IDS, tuy nhiên có thêm bộ nhớ cho việc lưu dữ liệu học tập.

Ưu/nhược điểm:

* Ưu điểm:
  + Kết hợp khả năng tiết kiệm bộ nhớ của IDS và khả năng thích nghi của Learning.
  + Cải thiện hiệu năng tìm kiếm trong môi trường lặp lại, chẳng hạn như Pacman chơi nhiều ván liên tiếp.
* Nhược điểm:
  + Cần thêm cơ chế lưu trữ, cập nhật và truy xuất dữ liệu học, tăng độ phức tạp lập trình.
  + Hiệu quả phụ thuộc vào chất lượng dữ liệu học được (ví dụ: nếu Pacman học từ tình huống sai, có thể dẫn đến quyết định kém).

Thuật toán IDL viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def idl\_move(self, maze):          """IDL: Iterative Deepening Limited - Tăng dần depth với giới hạn linh hoạt"""          max\_depth\_limit = 25  # Giới hạn depth tối đa            for depth in range(1, max\_depth\_limit + 1):              result = self.\_depth\_limited\_search(maze, depth)              if result:                  return result                # Dừng sớm nếu không tìm thấy dot trong vùng gần              if depth > 10 and not self.\_has\_nearby\_dots(maze, depth):                  break            return self.first\_valid\_move(maze, [])        def \_depth\_limited\_search(self, maze, max\_depth):          """DLS với cải tiến cho IDL"""          stack = [(self.x, self.y, [], 0)]          visited = set()            while stack:              x, y, path, depth = stack.pop()                if depth >= max\_depth:                  continue                if (x, y) in visited:                  continue              visited.add((x, y))                # Kiểm tra nếu tìm thấy dot              if maze[y][x] == CellType.DOT:                  if path:                      return path[0]                  else:                      return self.first\_valid\_move(maze, [])                # Thêm các hướng với heuristic ordering              directions = list(Direction)              # Sắp xếp directions theo khoảng cách đến dot gần nhất              if hasattr(self, '\_nearest\_dot'):                  directions.sort(key=lambda d: self.\_direction\_heuristic(d, x, y))                for direction in directions:                  dx, dy = direction.value                  new\_x, new\_y = x + dx, y + dy                    if (0 <= new\_x < MAZE\_WIDTH and 0 <= new\_y < MAZE\_HEIGHT and                      maze[new\_y][new\_x] != CellType.WALL and                      (new\_x, new\_y) not in visited):                      stack.append((new\_x, new\_y, path + [direction], depth + 1))            return None        def \_has\_nearby\_dots(self, maze, radius):          """Kiểm tra có dot nào trong bán kính radius không"""          for y in range(max(0, self.y - radius), min(MAZE\_HEIGHT, self.y + radius + 1)):              for x in range(max(0, self.x - radius), min(MAZE\_WIDTH, self.x + radius + 1)):                  if maze[y][x] == CellType.DOT:                      return True          return False        def \_direction\_heuristic(self, direction, x, y):          """Heuristic để sắp xếp directions theo độ ưu tiên"""          dx, dy = direction.value          new\_x, new\_y = x + dx, y + dy            # Tìm dot gần nhất nếu chưa cache          if not hasattr(self, '\_nearest\_dot') or self.\_nearest\_dot is None:              self.\_find\_nearest\_dot()            if self.\_nearest\_dot:              # Manhattan distance đến dot gần nhất              return abs(new\_x - self.\_nearest\_dot[0]) + abs(new\_y - self.\_nearest\_dot[1])          return 0        def \_find\_nearest\_dot(self):          """Tìm và cache dot gần nhất"""          dots = []          for y in range(MAZE\_HEIGHT):              for x in range(MAZE\_WIDTH):                  if hasattr(self, 'maze') and self.maze[y][x] == CellType.DOT:                      dots.append((x, y))            if dots:              self.\_nearest\_dot = min(dots, key=lambda d: abs(self.x - d[0]) + abs(self.y - d[1]))          else:              self.\_nearest\_dot = None |

## **1.3. Phân loại các thuật toán tìm kiếm có thông tin**

Các thuật toán tìm kiếm có thông tin (Informed Search) sử dụng hàm heuristic (h(n)) để định hướng quá trình tìm kiếm, giúp rút ngắn thời gian và tối ưu hóa hiệu quả so với các phương pháp tìm kiếm mù (Uninformed Search). Hàm heuristic thường là một ước lượng của khoảng cách hoặc chi phí còn lại từ một trạng thái hiện tại đến đích, ví dụ như khoảng cách Manhattan trong mê cung Pacman.

Trong Pacman AI, nhóm này giúp Pacman không cần duyệt toàn bộ bản đồ, mà vẫn chọn được đường đi hợp lý, nhanh và gần tối ưu, đặc biệt khi cần né Ghost hoặc tối ưu thời gian di chuyển đến dot gần nhất.

### **1.3.1. Greedy Best-First Search**

Mô tả ý tưởng: Thuật toán Greedy Best-First Search là một dạng tìm kiếm heuristic, trong đó việc mở rộng nút tiếp theo được chọn dựa hoàn toàn trên giá trị hàm heuristic h(n) – tức là ước lượng khoảng cách từ nút n đến mục tiêu. Thuật toán luôn chọn trạng thái mà theo ước lượng là “gần đích nhất”, nên được gọi là “tham lam” (Greedy).

Hàm đánh giá: Greedy Best-First Search sử dụng hàm:

trong đó:

* h(n): giá trị ước lượng khoảng cách từ trạng thái n đến đích (thường tính bằng khoảng cách Manhattan hoặc Euclidean trong Pacman).

Ứng dụng trong Pacman AI: Greedy Search giúp Pacman di chuyển nhanh về phía dot gần nhất hoặc né tránh Ghost bằng cách chọn bước đi sao cho ước lượng khoảng cách đến đích là nhỏ nhất.  
Ví dụ: Khi Pacman muốn ăn dot ở góc trên bên phải mê cung, thuật toán sẽ luôn chọn hướng có h(n) nhỏ hơn (tức là tiến gần hơn đến dot đó).

Tính chất:

* Hoàn chỉnh: Không (có thể rơi vào vòng lặp nếu không có kiểm tra visited).
* Tối ưu: Không (vì chỉ nhìn trước mắt, không tính chi phí đã đi).
* Độ phức tạp thời gian: O(b^m), phụ thuộc vào heuristic và cấu trúc mê cung.
* Độ phức tạp bộ nhớ: O(b^m), lưu các nút đã mở.

Ưu/nhược điểm:

* Ưu điểm: Tốc độ nhanh, dễ cài đặt, cho đường đi “tốt tương đối” trong nhiều trường hợp.
* Nhược điểm: Có thể mắc kẹt ở cực tiểu cục bộ, hoặc đi vòng xa hơn do không tính đến chi phí đã đi.

Thuật toán Greedy viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def greedy\_move(self, maze, ghosts):          """Greedy Best-First Search - Chạy xa ghost nhất"""          if not ghosts:              return self.first\_valid\_move(maze, ghosts)            # Tìm ghost gần nhất          min\_distance = float('inf')          nearest\_ghost = None            for ghost in ghosts:              distance = abs(self.x - ghost.x) + abs(self.y - ghost.y)              if distance < min\_distance:                  min\_distance = distance                  nearest\_ghost = ghost            if not nearest\_ghost:              return self.first\_valid\_move(maze, ghosts)            # Tìm hướng xa ghost nhất          best\_direction = None          max\_distance = -1            for direction in Direction:              dx, dy = direction.value              new\_x, new\_y = self.x + dx, self.y + dy                if (0 <= new\_x < MAZE\_WIDTH and 0 <= new\_y < MAZE\_HEIGHT and                  maze[new\_y][new\_x] != CellType.WALL):                    distance = abs(new\_x - nearest\_ghost.x) + abs(new\_y - nearest\_ghost.y)                  if distance > max\_distance:                      max\_distance = distance                      best\_direction = direction            return best\_direction if best\_direction else self.first\_valid\_move(maze, ghosts) |

### **1.3.2. Thuật toán A\* (A-Star Search)**

Mô tả ý tưởng: Nếu như thuật toán Greedy chỉ quan tâm đến ước lượng còn lại (h(n)) mà bỏ qua chi phí đã đi (g(n)), thì thuật toán A\* khắc phục điểm yếu này bằng cách kết hợp cả hai yếu tố.  
A\* được xem là một trong những thuật toán tìm kiếm mạnh nhất và phổ biến nhất trong trí tuệ nhân tạo, vì nó đảm bảo đường đi ngắn nhất (tối ưu) nếu hàm heuristic được thiết kế phù hợp.

Hàm đánh giá: A\* sử dụng hàm đánh giá:

trong đó:

* g(n): chi phí thực tế từ nút gốc đến nút n.
* h(n): chi phí ước lượng còn lại từ n đến đích (hàm heuristic).
* f(n): tổng chi phí ước lượng để đi từ gốc → n → đích.

Tính chất của hàm heuristic:

* Hàm h(n) được gọi là chấp nhận được (admissible) nếu:

với là chi phí thật ngắn nhất từ n đến đích.

→ Khi đó, A\* luôn tìm được lời giải tối ưu.

Ứng dụng trong Pacman AI: Trong trò chơi Pacman AI, thuật toán A\* được sử dụng để tìm đường đi ngắn nhất giữa Pacman và dot mục tiêu, hoặc né tránh Ghost theo đường an toàn nhất.

Ví dụ: Pacman có thể dùng A\* để xác định lộ trình ngắn nhất tới viên dot gần nhất, đồng thời tránh các ô có Ghost hoặc vùng nguy hiểm. A\* cũng có thể được dùng để điều hướng Ghost trong các chiến lược đối kháng.

Tính chất:

* Hoàn chỉnh: Có (nếu branching factor hữu hạn và chi phí ≥ ε > 0).
* Tối ưu: Có, nếu h(n) chấp nhận được.
* Độ phức tạp thời gian: O(b^d), phụ thuộc vào độ chính xác của heuristic.
* Độ phức tạp bộ nhớ: O(b^d), cần lưu toàn bộ frontier.

Ưu/nhược điểm:

* Ưu điểm: Tìm được đường đi tối ưu; hoạt động hiệu quả nếu heuristic tốt.
* Nhược điểm: Tiêu tốn bộ nhớ, đặc biệt khi không gian trạng thái lớn; phụ thuộc mạnh vào chất lượng của h(n).

Thuật toán A\* viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def a\_star\_move(self, maze, ghosts):          """A\* Search - Cân bằng giữa ăn chấm và né ghost"""          # Tìm dot gần nhất          dots = []          for y in range(MAZE\_HEIGHT):              for x in range(MAZE\_WIDTH):                  if maze[y][x] == CellType.DOT:                      dots.append((x, y))            if not dots:              return self.greedy\_move(maze, ghosts)            # Tìm dot gần nhất          target\_dot = min(dots, key=lambda dot: abs(self.x - dot[0]) + abs(self.y - dot[1]))            # A\* pathfinding          counter = 0          heap = [(0, counter, self.x, self.y, [])]          visited = set()            while heap:              f\_cost, \_, x, y, path = heapq.heappop(heap)              counter += 1              if (x, y) in visited:                  continue              visited.add((x, y))              # Kiểm tra nếu đến target              if (x, y) == target\_dot:                  if path:                      return path[0]                  else:                      return self.first\_valid\_move(maze, ghosts)              # Thêm các hướng              for direction in Direction:                  dx, dy = direction.value                  new\_x, new\_y = x + dx, y + dy                  if (0 <= new\_x < MAZE\_WIDTH and 0 <= new\_y < MAZE\_HEIGHT and                      maze[new\_y][new\_x] != CellType.WALL and                      (new\_x, new\_y) not in visited):                      # g = cost từ start                      g\_cost = len(path) + 1                      # h = heuristic (Manhattan distance + ghost avoidance)                      h\_cost = abs(new\_x - target\_dot[0]) + abs(new\_y - target\_dot[1])                      # Thêm penalty nếu gần ghost                      for ghost in ghosts:                          ghost\_distance = abs(new\_x - ghost.x) + abs(new\_y - ghost.y)                          if ghost\_distance < 3:  # Nếu quá gần ghost                              h\_cost += 10                      f\_cost = g\_cost + h\_cost                      heapq.heappush(heap, (f\_cost, counter, new\_x, new\_y, path + [direction]))                      counter += 1          return self.greedy\_move(maze, ghosts) |

## **1.4. Phân loại các tìm kiếm cục bộ**

Các thuật toán tìm kiếm cục bộ (Local Search**)** không mở rộng toàn bộ không gian trạng thái, mà chỉ tập trung vào một trạng thái hiện tại (current state) và tìm cách cải thiện nghiệm này bằng cách di chuyển sang các trạng thái “lân cận” tốt hơn theo một tiêu chí đánh giá (evaluation function).

Khác với các thuật toán tìm kiếm có cấu trúc cây (như BFS, A\*), local search không lưu trữ đường đi đầy đủ, mà chỉ quan tâm đến việc cải thiện nghiệm hiện tại. Trong trò chơi Pacman AI, nhóm thuật toán này giúp Pacman tìm hướng đi tốt nhất dựa trên tình huống hiện tại, đặc biệt khi không cần tìm đường tối ưu tuyệt đối mà chỉ cần hành vi hợp lý, nhanh và thích ứng.

**1.4.1. Hill-Climbing Search**

Mô tả ý tưởng: Thuật toán Hill-Climbing (leo đồi) là một phương pháp tìm kiếm cục bộ, trong đó Pacman bắt đầu từ trạng thái hiện tại và liên tục di chuyển sang trạng thái lân cận có giá trị đánh giá tốt hơn (cao hơn hoặc thấp hơn tùy bài toán tối đa hóa hay tối thiểu hóa).  
Quá trình này lặp lại cho đến khi không còn trạng thái lân cận nào tốt hơn — khi đó, thuật toán đạt đến cực đại (hoặc cực tiểu) cục bộ.

Hàm đánh giá (evaluation function):

hoặc

tùy mục tiêu của Pacman là đến gần dot nhất hoặc tránh Ghost an toàn nhất.

Ứng dụng trong Pacman AI:  
Hill-Climbing giúp Pacman liên tục chọn hướng đi giúp cải thiện tình huống hiện tại, ví dụ:

* Chọn bước di chuyển làm giảm khoảng cách đến dot gần nhất.
* Né hướng có Ghost đang đến gần.

Tính chất:

* Hoàn chỉnh: Không (có thể kẹt tại cực tiểu cục bộ).
* Tối ưu: Không đảm bảo.
* Thời gian: O(b·m) — b là số hướng khả dĩ, m là số bước tối đa.
* Bộ nhớ: O(1) — chỉ lưu trạng thái hiện tại.

Ưu/nhược điểm:

* Ưu: Nhanh, tiết kiệm bộ nhớ, dễ triển khai.
* Nhược: Dễ kẹt ở cực tiểu cục bộ hoặc đỉnh bằng (plateau), không đảm bảo tìm được đường tốt nhất.

Thuật toán Hill-Climbing viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def hill\_climbing\_move(self, maze, ghosts):          """Hill-Climbing - Leo dốc tìm giá trị tốt nhất"""          current\_pos = (self.x, self.y)          current\_score = self.\_evaluate\_position(current\_pos, [(g.x, g.y) for g in ghosts], maze)            best\_direction = None          best\_score = current\_score            # Thử tất cả các hướng có thể đi          for direction in Direction:              dx, dy = direction.value              new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)                if self.\_is\_valid\_move(new\_pos, maze):                  score = self.\_evaluate\_position(new\_pos, [(g.x, g.y) for g in ghosts], maze)                    # Chọn hướng có điểm số tốt hơn (leo dốc)                  if score > best\_score:                      best\_score = score                      best\_direction = direction            # Nếu không tìm thấy hướng tốt hơn, di chuyển ngẫu nhiên để thoát khỏi local maximum          if best\_direction is None:              return self.first\_valid\_move(maze, ghosts)            return best\_direction |

**1.4.2. Simulated Annealing Search**

Mô tả ý tưởng:  
Simulated Annealing (tôi luyện mô phỏng) là một cải tiến của Hill-Climbing, được lấy cảm hứng từ quá trình nung chảy và làm nguội kim loại trong vật lý.  
Thuật toán cho phép Pacman thỉnh thoảng chấp nhận di chuyển tới trạng thái “xấu hơn” (giảm giá trị f) với xác suất phụ thuộc vào nhiệt độ T, nhằm tránh mắc kẹt tại cực tiểu cục bộ.

Xác suất chấp nhận bước đi “xấu hơn” được tính bằng công thức:

trong đó:

* ΔE là độ giảm giá trị hàm đánh giá (mức “xấu hơn”),
* T là nhiệt độ hiện tại, giảm dần theo thời gian.

Ứng dụng trong Pacman AI:  
Pacman có thể dùng Simulated Annealing để:

* Khám phá đường đi mới trong mê cung khi các hướng gần đều bị Ghost chặn.
* Tìm cách thoát khỏi vùng nguy hiểm, chấp nhận đi lùi tạm thời để có lợi hơn sau này.

Tính chất:

* Hoàn chỉnh: Có xác suất cao (nếu T giảm chậm).
* Tối ưu: Có thể đạt tối ưu toàn cục nếu T giảm đủ chậm.
* Thời gian: O(k) — k là số lần lặp.
* Bộ nhớ: O(1).

Ưu/nhược điểm:

* Ưu: Có thể thoát khỏi cực tiểu cục bộ, cho nghiệm gần tối ưu.
* Nhược: Cần chọn lịch giảm nhiệt độ (cooling schedule) phù hợp; có thể chậm nếu T giảm quá từ từ.

Thuật toán Simulated Annealing viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def simulated\_annealing\_move(self, maze, ghosts):          """Simulated Annealing - Cho phép di chuyển xấu hơn với xác suất giảm dần"""          if not hasattr(self, 'temperature'):              self.temperature = 100.0  # Nhiệt độ ban đầu              self.cooling\_rate = 0.95  # Tốc độ làm lạnh            current\_pos = (self.x, self.y)          current\_score = self.\_evaluate\_position(current\_pos, [(g.x, g.y) for g in ghosts], maze)            # Chọn một hướng ngẫu nhiên để thử          directions = [d for d in Direction if self.\_is\_valid\_move(              (current\_pos[0] + d.value[0], current\_pos[1] + d.value[1]), maze)]            if not directions:              return self.first\_valid\_move(maze, ghosts)            candidate\_direction = random.choice(directions)          dx, dy = candidate\_direction.value          new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)          new\_score = self.\_evaluate\_position(new\_pos, [(g.x, g.y) for g in ghosts], maze)            # Quyết định có chấp nhận di chuyển không          if new\_score > current\_score:              # Di chuyển tốt hơn - luôn chấp nhận              selected\_direction = candidate\_direction          else:              # Di chuyển xấu hơn - chấp nhận với xác suất              delta = new\_score - current\_score              probability = math.exp(delta / max(self.temperature, 0.1))                if random.random() < probability:                  selected\_direction = candidate\_direction              else:                  # Từ chối, chọn hướng tốt nhất có thể                  selected\_direction = self.greedy\_move(maze, ghosts)            # Làm lạnh          self.temperature \*= self.cooling\_rate          if self.temperature < 1.0:              self.temperature = 100.0  # Reset nhiệt độ            return selected\_direction |

**1.4.3. Beam Search**

Mô tả ý tưởng: Beam Search là một thuật toán tìm kiếm heuristic có kiểm soát độ rộng, chỉ giữ lại k trạng thái tốt nhất tại mỗi bước (k gọi là *beam width*).  
Thay vì mở rộng toàn bộ không gian (như BFS), Beam Search chỉ tập trung vào các ứng viên “hứa hẹn nhất” dựa trên giá trị hàm heuristic.

Hàm đánh giá:

tương tự như Greedy, nhưng chỉ duy trì một số lượng giới hạn các trạng thái “tốt nhất”.

Ứng dụng trong Pacman AI: Beam Search giúp Pacman tìm đường nhanh trong bản đồ lớn bằng cách:

* Duy trì một “chùm” hướng đi tiềm năng tốt nhất.
* Bỏ qua những hướng ít hứa hẹn, giúp tiết kiệm thời gian xử lý.

Tính chất:

* Hoàn chỉnh: Không (vì có thể bỏ qua hướng dẫn đến đích).
* Tối ưu: Không đảm bảo, phụ thuộc vào giá trị k.
* Thời gian: O(k·b·d),
* Bộ nhớ: O(k·d).

Ưu/nhược điểm:

* Ưu: Tốc độ nhanh hơn BFS/A\* rất nhiều khi k nhỏ; phù hợp cho môi trường lớn.
* Nhược: Có thể bỏ lỡ lời giải tối ưu nếu beam width quá nhỏ.

Thuật toán Beam Search viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def beam\_search\_move(self, maze, ghosts, beam\_width=3):          """Beam Search - Tìm kiếm với độ rộng beam giới hạn"""          # Khởi tạo beam với vị trí hiện tại          beam = [(self.\_evaluate\_position((self.x, self.y), [(g.x, g.y) for g in ghosts], maze),                   self.x, self.y, [])]            # Tìm kiếm với độ sâu giới hạn          for depth in range(5):  # Giới hạn độ sâu              new\_beam = []                for score, x, y, path in beam:                  # Mở rộng từ vị trí hiện tại                  for direction in Direction:                      dx, dy = direction.value                      new\_x, new\_y = x + dx, y + dy                        if self.\_is\_valid\_move((new\_x, new\_y), maze):                          new\_score = self.\_evaluate\_position((new\_x, new\_y), [(g.x, g.y) for g in ghosts], maze)                          new\_path = path + [direction]                          new\_beam.append((new\_score, new\_x, new\_y, new\_path))                # Giữ lại beam\_width trạng thái tốt nhất              new\_beam.sort(key=lambda x: x[0], reverse=True)              beam = new\_beam[:beam\_width]                # Kiểm tra nếu tìm thấy dot              for score, x, y, path in beam:                  if maze[y][x] == CellType.DOT and path:                      return path[0]            # Trả về hướng tốt nhất từ beam          if beam and beam[0][3]:              return beam[0][3][0]            return self.first\_valid\_move(maze, ghosts) |

**1.4.4. Genetic Algorithm (Giải thuật di truyền)**

Mô tả ý tưởng:  
Giải thuật di truyền (Genetic Algorithm – GA) là một thuật toán tối ưu hóa dựa trên cơ chế tiến hóa tự nhiên của sinh học.  
GA hoạt động trên một quần thể các nghiệm (population), và thông qua các quá trình chọn lọc (selection), lai ghép (crossover), đột biến (mutation), nó dần dần tạo ra các cá thể tốt hơn theo thời gian.

Quá trình hoạt động cơ bản:

1. Khởi tạo một quần thể ngẫu nhiên các lời giải.
2. Đánh giá độ thích nghi (fitness) của từng cá thể.
3. Chọn ra các cá thể tốt nhất để sinh sản (selection).
4. Lai ghép và đột biến để tạo ra thế hệ mới.
5. Lặp lại cho đến khi đạt điều kiện dừng (ví dụ: đạt điểm cao nhất hoặc đủ số thế hệ).

Hàm đánh giá (Fitness function):  
Trong Pacman AI, hàm này thường là:

→ Càng ăn được nhiều điểm, càng gần mục tiêu và càng tránh xa Ghost thì fitness càng cao.

Ứng dụng trong Pacman AI:  
Trong Pacman, GA có thể được dùng để:

* Tối ưu chuỗi hành động (sequence of moves) giúp Pacman đạt điểm cao nhất trong một thời gian giới hạn.
* Tìm đường đi tốt nhất trong mê cung có nhiều dot và Ghost di chuyển.
* Huấn luyện chiến lược di chuyển (Pacman “học” cách né Ghost và chọn hướng tốt nhất sau nhiều thế hệ).

Tính chất:

* Hoàn chỉnh: Không (mang tính xác suất).
* Tối ưu: Gần tối ưu (không đảm bảo tuyệt đối).
* Thời gian: Phụ thuộc số thế hệ và kích thước quần thể.
* Bộ nhớ: O(population\_size).

Ưu/nhược điểm:

* Ưu điểm:
  + Có thể tìm nghiệm gần tối ưu trong không gian lớn, phức tạp.
  + Linh hoạt và thích nghi với nhiều loại bài toán khác nhau.
* Nhược điểm:
  + Thời gian huấn luyện lâu.
  + Cần điều chỉnh thông số (tỉ lệ lai ghép, đột biến, kích thước quần thể) phù hợp.

Thuật toán Genetic Algorithm viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def genetic\_move(self, maze, ghosts):          """Genetic Algorithm - Thuật toán di truyền"""          if not ghosts:              return self.greedy\_move(maze, ghosts)            # Tạo population của các moves          population = []          for \_ in range(20):              moves = [random.choice(list(Direction)) for \_ in range(5)]              fitness = self.\_evaluate\_move\_sequence(moves, maze, ghosts)              population.append((moves, fitness))            # Sắp xếp theo fitness          population.sort(key=lambda x: x[1], reverse=True)            # Trả về move tốt nhất          return population[0][0][0] if population else self.first\_valid\_move(maze, ghosts) |

## **1.5. Phân loại các thuật toán tìm kiếm đối kháng**

Trong nhiều trò chơi hoặc môi trường có nhiều tác nhân (agents), mỗi tác nhân đều có mục tiêu riêng, thậm chí đối lập nhau. Khi Pacman cố gắng tối đa hóa điểm số hoặc sống sót lâu nhất, các Ghost lại tìm cách bắt được Pacman.  
Đây là bài toán tìm kiếm đối kháng (Adversarial Search) — một dạng đặc biệt của tìm kiếm, trong đó cần xem xét hành động của cả đối thủ khi ra quyết định.

### **1.5.1. Minimax Algorithm**

Mô tả ý tưởng:  
Thuật toán Minimax được sử dụng trong môi trường có hai người chơi đối kháng:

* Một bên là Max (trong Pacman là Pacman) — cố gắng tối đa hóa điểm số hoặc lợi ích.
* Một bên là Min (Ghost) — cố gắng tối thiểu hóa lợi ích của Pacman.

Thuật toán xây dựng cây trò chơi (game tree), trong đó:

* Các nút MAX đại diện cho lượt đi của Pacman.
* Các nút MIN đại diện cho lượt đi của Ghost.
* Các lá có giá trị điểm đánh giá (utility), mô tả kết quả của trạng thái (ví dụ: số điểm, khoảng cách tới Ghost, số dot còn lại…).

Mục tiêu là chọn hành động ở mức gốc giúp Pacman đạt giá trị Minimax lớn nhất, nghĩa là giả định Ghost sẽ luôn chơi “hoàn hảo” để giảm điểm của Pacman.

Công thức:

Ứng dụng trong Pacman AI:  
Pacman dùng Minimax để dự đoán hành động của Ghost và chọn hướng đi giúp tối đa hóa cơ hội sống sót hoặc điểm số.  
Ví dụ: nếu Pacman thấy Ghost sắp di chuyển đến gần, nó có thể chọn đường vòng xa hơn để giảm nguy cơ bị bắt, dù mất thêm vài bước.

Tính chất:

* Hoàn chỉnh: Có, nếu cây trò chơi hữu hạn.
* Tối ưu: Có, nếu cả hai bên chơi hoàn hảo.
* Độ phức tạp thời gian: O(b^m) — b là số hành động mỗi lượt, m là độ sâu tìm kiếm.
* Độ phức tạp bộ nhớ: O(b·m).

Ưu/nhược điểm:

* Ưu: Cung cấp chiến lược ra quyết định hợp lý trong môi trường đối kháng.
* Nhược: Chi phí tính toán lớn, đặc biệt khi có nhiều Ghost hoặc độ sâu lớn.

Thuật toán Minimax Algorithm viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def minimax\_move(self, maze, ghosts, depth=3):          """Minimax Algorithm"""          if not ghosts:              return self.greedy\_move(maze, ghosts)            def minimax(pos, ghost\_positions, depth, maximizing):              if depth == 0:                  return self.\_evaluate\_position(pos, ghost\_positions, maze)                if maximizing:  # Pacman's turn                  best\_score = float('-inf')                  for direction in Direction:                      dx, dy = direction.value                      new\_pos = (pos[0] + dx, pos[1] + dy)                      if self.\_is\_valid\_move(new\_pos, maze):                          score = minimax(new\_pos, ghost\_positions, depth - 1, False)                          best\_score = max(best\_score, score)                  return best\_score              else:  # Ghosts' turn                  worst\_score = float('inf')                  for ghost\_pos in ghost\_positions:                      for direction in Direction:                          dx, dy = direction.value                          new\_ghost\_pos = (ghost\_pos[0] + dx, ghost\_pos[1] + dy)                          if self.\_is\_valid\_move(new\_ghost\_pos, maze):                              new\_ghost\_positions = [new\_ghost\_pos if gp == ghost\_pos else gp for gp in ghost\_positions]                              score = minimax(pos, new\_ghost\_positions, depth - 1, True)                              worst\_score = min(worst\_score, score)                  return worst\_score            best\_direction = None          best\_score = float('-inf')          current\_pos = (self.x, self.y)          ghost\_positions = [(g.x, g.y) for g in ghosts]            for direction in Direction:              dx, dy = direction.value              new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)              if self.\_is\_valid\_move(new\_pos, maze):                  score = minimax(new\_pos, ghost\_positions, depth - 1, False)                  if score > best\_score:                      best\_score = score                      best\_direction = direction            return best\_direction if best\_direction else self.first\_valid\_move(maze, ghosts) |

### **1.5.2. Alpha–Beta Pruning**

Mô tả ý tưởng:  
Alpha–Beta Pruning là phiên bản cải tiến của Minimax, giúp cắt bỏ (prune) những nhánh không cần thiết trong cây trò chơi, mà vẫn đảm bảo kết quả giống Minimax.  
Ý tưởng là giữ hai giá trị:

* α (alpha): giá trị tốt nhất mà MAX có thể đạt được cho đến hiện tại.
* β (beta): giá trị tốt nhất mà MIN có thể đạt được cho đến hiện tại.

Khi phát hiện rằng một nhánh không thể cải thiện kết quả (α ≥ β), thuật toán sẽ dừng mở rộng nhánh đó.

Công thức:  
Giống Minimax, nhưng thêm điều kiện cắt tỉa:

* Nếu tại nút MAX: if value >= beta → break (vì MIN sẽ không chọn nhánh tệ hơn).
* Nếu tại nút MIN: if value <= alpha → break (vì MAX sẽ không chọn nhánh kém hơn).

Ứng dụng trong Pacman AI:  
Alpha–Beta giúp Pacman ra quyết định nhanh hơn, đặc biệt khi có nhiều Ghost hoặc khi tìm kiếm sâu.  
Ví dụ: Khi Pacman có nhiều hướng di chuyển, Alpha–Beta giúp loại bỏ những hướng chắc chắn kém hiệu quả (ví dụ, đường cụt có Ghost gần).

Tính chất:

* Hoàn chỉnh: Có, nếu cây trò chơi hữu hạn.
* Tối ưu: Có, kết quả giống Minimax.
* Thời gian: Trung bình O(b^(m/2)) nếu cắt tỉa tốt.
* Bộ nhớ: O(b·m).

Ưu/nhược điểm:

* Ưu: Giảm đáng kể thời gian tính toán so với Minimax.
* Nhược: Hiệu quả cắt tỉa phụ thuộc vào thứ tự duyệt nút (nếu duyệt tốt → tiết kiệm cực lớn).

Thuật toán Alpha–Beta Pruning viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def alpha\_beta\_move(self, maze, ghosts, depth=3):          """Alpha-Beta Pruning - Cải tiến Minimax"""          if not ghosts:              return self.greedy\_move(maze, ghosts)            def alpha\_beta(pos, ghost\_positions, depth, alpha, beta, maximizing):              if depth == 0:                  return self.\_evaluate\_position(pos, ghost\_positions, maze)                if maximizing:  # Pacman's turn                  max\_eval = float('-inf')                  for direction in Direction:                      dx, dy = direction.value                      new\_pos = (pos[0] + dx, pos[1] + dy)                      if self.\_is\_valid\_move(new\_pos, maze):                          eval\_score = alpha\_beta(new\_pos, ghost\_positions, depth - 1, alpha, beta, False)                          max\_eval = max(max\_eval, eval\_score)                          alpha = max(alpha, eval\_score)                          if beta <= alpha:                              break  # Alpha-beta pruning                  return max\_eval              else:  # Ghosts' turn                  min\_eval = float('inf')                  for ghost\_pos in ghost\_positions:                      for direction in Direction:                          dx, dy = direction.value                          new\_ghost\_pos = (ghost\_pos[0] + dx, ghost\_pos[1] + dy)                          if self.\_is\_valid\_move(new\_ghost\_pos, maze):                              new\_ghost\_positions = [new\_ghost\_pos if gp == ghost\_pos else gp for gp in ghost\_positions]                              eval\_score = alpha\_beta(pos, new\_ghost\_positions, depth - 1, alpha, beta, True)                              min\_eval = min(min\_eval, eval\_score)                              beta = min(beta, eval\_score)                              if beta <= alpha:                                  break  # Alpha-beta pruning                  return min\_eval            best\_direction = None          best\_score = float('-inf')          current\_pos = (self.x, self.y)          ghost\_positions = [(g.x, g.y) for g in ghosts]          alpha = float('-inf')          beta = float('inf')            for direction in Direction:              dx, dy = direction.value              new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)              if self.\_is\_valid\_move(new\_pos, maze):                  score = alpha\_beta(new\_pos, ghost\_positions, depth - 1, alpha, beta, False)                  if score > best\_score:                      best\_score = score                      best\_direction = direction                  alpha = max(alpha, score)            return best\_direction if best\_direction else self.first\_valid\_move(maze, ghosts) |

### **1.5.3. Expectimax Algorithm**

Mô tả ý tưởng:  
Khác với Minimax (coi đối thủ luôn “chơi hoàn hảo” để hại ta), Expectimax giả định đối thủ hành động ngẫu nhiên theo xác suất.  
Vì vậy, thay vì chọn giá trị min, Expectimax dùng giá trị kỳ vọng (expected value) của các hành động của đối thủ.

Công thức:

Ứng dụng trong Pacman AI:  
Expectimax phù hợp khi Ghost không di chuyển có chủ đích, mà chọn hướng ngẫu nhiên.  
Thuật toán giúp Pacman đưa ra chiến lược tối ưu trung bình trong môi trường không chắc chắn.  
Ví dụ: nếu Ghost di chuyển ngẫu nhiên 50% sang trái, 50% xuống, Pacman sẽ chọn hướng tối ưu dựa trên giá trị kỳ vọng trung bình của các khả năng đó.

Tính chất:

* Hoàn chỉnh: Có, nếu cây trò chơi hữu hạn.
* Tối ưu: Có, theo giá trị kỳ vọng (không theo “tốt nhất” tuyệt đối).
* Thời gian: O(b^m).
* Bộ nhớ: O(b·m).

Ưu/nhược điểm:

* Ưu: Phù hợp môi trường ngẫu nhiên hoặc không thể dự đoán hoàn toàn hành vi đối thủ.
* Nhược: Không đảm bảo an toàn tuyệt đối — chỉ tốt “trung bình”.

Thuật toán Expectimax Algorithm viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def expectimax\_move(self, maze, ghosts, depth=3):          """Expectimax - Tính toán giá trị kỳ vọng cho các sự kiện ngẫu nhiên"""          if not ghosts:              return self.greedy\_move(maze, ghosts)            def expectimax(pos, ghost\_positions, depth, maximizing):              if depth == 0:                  return self.\_evaluate\_position(pos, ghost\_positions, maze)                if maximizing:  # Pacman's turn                  max\_score = float('-inf')                  for direction in Direction:                      dx, dy = direction.value                      new\_pos = (pos[0] + dx, pos[1] + dy)                      if self.\_is\_valid\_move(new\_pos, maze):                          score = expectimax(new\_pos, ghost\_positions, depth - 1, False)                          max\_score = max(max\_score, score)                  return max\_score              else:  # Ghosts' turn (random/probabilistic)                  total\_score = 0                  valid\_moves = 0                    for ghost\_pos in ghost\_positions:                      for direction in Direction:                          dx, dy = direction.value                          new\_ghost\_pos = (ghost\_pos[0] + dx, ghost\_pos[1] + dy)                          if self.\_is\_valid\_move(new\_ghost\_pos, maze):                              new\_ghost\_positions = [new\_ghost\_pos if gp == ghost\_pos else gp for gp in ghost\_positions]                              score = expectimax(pos, new\_ghost\_positions, depth - 1, True)                              total\_score += score                              valid\_moves += 1                    # Trả về giá trị kỳ vọng                  return total\_score / max(valid\_moves, 1)            best\_direction = None          best\_score = float('-inf')          current\_pos = (self.x, self.y)          ghost\_positions = [(g.x, g.y) for g in ghosts]            for direction in Direction:              dx, dy = direction.value              new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)              if self.\_is\_valid\_move(new\_pos, maze):                  score = expectimax(new\_pos, ghost\_positions, depth - 1, False)                  if score > best\_score:                      best\_score = score                      best\_direction = direction            return best\_direction if best\_direction else self.first\_valid\_move(maze, ghosts) |

## **1.6. Phân loại các thuật toán tìm kiếm có thông tin**

Trong trí tuệ nhân tạo, nhiều bài toán có thể được biểu diễn dưới dạng CSP (Constraint Satisfaction Problem) – *bài toán thỏa mãn ràng buộc*, trong đó ta cần tìm giá trị cho một tập biến sao cho thỏa mãn tất cả các điều kiện đã cho. Bên cạnh đó, Planning (lập kế hoạch) là quá trình xác định chuỗi hành động tối ưu để đạt được mục tiêu từ trạng thái ban đầu.

Trong trò chơi Pacman AI, nhóm thuật toán này giúp Pacman lên kế hoạch di chuyển thông minh hơn, không chỉ “chạy theo dot gần nhất”, mà còn biết tránh Ghost, tối ưu đường đi, và xử lý đồng thời nhiều mục tiêu (nhiều dot, nhiều ràng buộc).

### **1.6.1. Backtracking Search**

Mô tả ý tưởng:

Backtracking (quay lui) là phương pháp giải CSP cổ điển, trong đó ta gán giá trị cho các biến tuần tự, và quay lui (backtrack) khi phát hiện có xung đột với ràng buộc.  
Thuật toán này dựa trên nguyên tắc thử – sai có kiểm soát: thử một giá trị hợp lệ, nếu không hợp lệ hoặc không tìm được nghiệm thì quay lại để thử giá trị khác.

Ứng dụng trong Pacman AI:

Trong Pacman, Backtracking được dùng để tìm chuỗi di chuyển hợp lệ (path) từ vị trí hiện tại đến dot mục tiêu, sao cho:

* Pacman không đi vào tường,
* không gặp Ghost,
* và đến được đích trong số bước giới hạn.

Thuật toán sẽ thử từng hướng di chuyển (lên, xuống, trái, phải), kiểm tra tính hợp lệ, và quay lui khi gặp bế tắc.

Tính chất:

* Hoàn chỉnh: Có, nếu không gian trạng thái hữu hạn.
* Tối ưu: Không đảm bảo.
* Thời gian: O(b^d) — vì có thể phải duyệt nhiều nhánh.
* Bộ nhớ: O(b·d).

Ưu/nhược điểm:

* Ưu: Dễ triển khai, hiệu quả với không gian nhỏ, có thể tìm được mọi nghiệm.
* Nhược: Hiệu suất giảm khi số biến hoặc ràng buộc lớn; có thể lặp lại tính toán nhiều lần.

Thuật toán Backtracking Search viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def backtracking\_move(self, maze, ghosts):          # Tìm tất cả các dots          dots = []          for y in range(MAZE\_HEIGHT):              for x in range(MAZE\_WIDTH):                  if maze[y][x] == CellType.DOT:                      dots.append((x, y))            if not dots:              return self.first\_valid\_move(maze, ghosts)            # Tìm dot gần nhất          target = min(dots, key=lambda d: abs(self.x - d[0]) + abs(self.y - d[1]))            # CSP: Variables = positions in path, Domain = directions, Constraints = valid moves          def backtrack\_search(current\_pos, target\_pos, path, visited, depth=0):              """Recursive backtracking tìm path đến target"""              if depth > 20:  # Giới hạn độ sâu                  return None                if current\_pos == target\_pos:                  return path                # Thử tất cả các hướng (domain của variable)              for direction in Direction:                  dx, dy = direction.value                  new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)                    # Kiểm tra constraints                  if (self.\_is\_valid\_move(new\_pos, maze) and                      new\_pos not in visited and                      not self.\_is\_ghost\_position(new\_pos, ghosts)):                        # Assign value to variable                      visited.add(new\_pos)                      new\_path = path + [direction]                        # Recursive backtrack                      result = backtrack\_search(new\_pos, target, new\_path, visited, depth + 1)                        if result is not None:                          return result                        # Backtrack (unassign)                      visited.remove(new\_pos)                return None            # Thực hiện backtracking search          path = backtrack\_search((self.x, self.y), target, [], {(self.x, self.y)})            if path and len(path) > 0:              return path[0]            return self.first\_valid\_move(maze, ghosts) |

### **1.6.2. Forward Checking**

Mô tả ý tưởng:  
Forward Checking là cải tiến của Backtracking, trong đó sau khi gán giá trị cho một biến, thuật toán sẽ kiểm tra trước (look ahead) để loại bỏ các giá trị không còn hợp lệ khỏi miền giá trị của các biến chưa gán.  
Nhờ đó, thuật toán có thể phát hiện sớm xung đột và tránh lặp lại các bước vô ích.

Ứng dụng trong Pacman AI:  
Khi Pacman chọn bước đi, Forward Checking sẽ kiểm tra trước xem bước đó có khiến tương lai không còn nước hợp lệ hay không (ví dụ: bị kẹt, không còn đường né Ghost).  
Nếu có, hướng đó sẽ bị loại bỏ trước khi thực hiện.

Tính chất:

* Hoàn chỉnh: Có, với không gian hữu hạn.
* Tối ưu: Không đảm bảo, nhưng nhanh hơn Backtracking.
* Thời gian: O(b·d) với nhiều phép kiểm tra trước.
* Bộ nhớ: O(b·d).

Ưu/nhược điểm:

* Ưu: Giảm đáng kể số nhánh cần duyệt; phát hiện lỗi sớm.
* Nhược: Tốn thêm chi phí kiểm tra trước mỗi lần gán.

Thuật toán Forward Checking viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def forward\_checking\_move(self, maze, ghosts):          """          Forward Checking - CSP với kiểm tra ràng buộc trước          Cải tiến backtracking bằng cách loại bỏ các giá trị không hợp lệ          từ domain của các variables chưa assign          """          # Tìm target dot          dots = []          for y in range(MAZE\_HEIGHT):              for x in range(MAZE\_WIDTH):                  if maze[y][x] == CellType.DOT:                      dots.append((x, y))            if not dots:              return self.first\_valid\_move(maze, ghosts)            target = min(dots, key=lambda d: abs(self.x - d[0]) + abs(self.y - d[1]))            def get\_valid\_directions(pos, visited, ghosts):              """Lấy domain của directions còn hợp lệ (Forward Checking)"""              valid\_dirs = []              for direction in Direction:                  dx, dy = direction.value                  new\_pos = (pos[0] + dx, pos[1] + dy)                    # Check constraints                  if (self.\_is\_valid\_move(new\_pos, maze) and                      new\_pos not in visited and                      not self.\_is\_ghost\_position(new\_pos, ghosts)):                      valid\_dirs.append(direction)                return valid\_dirs            def forward\_check\_search(current\_pos, target\_pos, path, visited, depth=0):              """CSP search với forward checking"""              if depth > 20:                  return None                if current\_pos == target\_pos:                  return path                # Forward checking: chỉ thử các directions còn hợp lệ              valid\_directions = get\_valid\_directions(current\_pos, visited, ghosts)                if not valid\_directions:  # Domain rỗng -> backtrack                  return None                # Sắp xếp directions theo heuristic (gần target hơn)              valid\_directions.sort(                  key=lambda d: abs((current\_pos[0] + d.value[0]) - target\_pos[0]) +                                abs((current\_pos[1] + d.value[1]) - target\_pos[1])              )                for direction in valid\_directions:                  dx, dy = direction.value                  new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)                    visited.add(new\_pos)                  new\_path = path + [direction]                    result = forward\_check\_search(new\_pos, target\_pos, new\_path, visited, depth + 1)                    if result is not None:                      return result                    visited.remove(new\_pos)                return None            path = forward\_check\_search((self.x, self.y), target, [], {(self.x, self.y)})            if path and len(path) > 0:              return path[0]            return self.first\_valid\_move(maze, ghosts) |

### **1.6.3. AC-3 (Arc Consistency Algorithm 3)**

Mô tả ý tưởng:  
AC-3 (Arc Consistency Algorithm 3) là một thuật toán phổ biến trong CSP, giúp duy trì tính nhất quán cung (arc consistency) giữa các biến.  
Một bài toán được gọi là “nhất quán cung” nếu với mọi giá trị của một biến X, luôn tồn tại ít nhất một giá trị hợp lệ tương ứng cho biến Y sao cho ràng buộc giữa X và Y được thỏa mãn.

Nguyên tắc hoạt động:

* Thuật toán xem mỗi cặp biến (X, Y) có ràng buộc là một cung (arc).
* Với mỗi cung, nếu có giá trị trong X không thể thỏa mãn với mọi giá trị trong Y, giá trị đó bị loại bỏ.
* Quá trình lặp lại cho đến khi không còn thay đổi nào xảy ra.

Ứng dụng trong Pacman AI:  
Trong Pacman, AC-3 có thể dùng để:

* Duy trì các ràng buộc giữa các vị trí hợp lệ, ví dụ: “Pacman không thể ở cùng vị trí với Ghost”, hoặc “Pacman chỉ có thể di chuyển đến ô không có tường”.
* Loại bỏ sớm các hướng hoặc vùng không thể đi tới.

Tính chất:

* Hoàn chỉnh: Có, vì đảm bảo nhất quán cục bộ.
* Tối ưu: Không; AC-3 chỉ loại bỏ giá trị không hợp lệ, không tìm đường.
* Độ phức tạp thời gian: O(e·d³), với e là số cung, d là kích thước miền giá trị.
* Bộ nhớ: O(e).

Ưu/nhược điểm:

* Ưu: Giảm đáng kể không gian tìm kiếm, giúp Backtracking/Forward Checking nhanh hơn.
* Nhược: Chỉ đảm bảo nhất quán cục bộ, không đảm bảo tìm được lời giải hoàn chỉnh.

Thuật toán AC-3 viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def ac3\_move(self, maze, ghosts):          """          AC-3 (Arc Consistency 3) - Thuật toán CSP với arc consistency          Loại bỏ các giá trị không nhất quán từ domain trước khi tìm kiếm          """          # Tìm target dot          dots = []          for y in range(MAZE\_HEIGHT):              for x in range(MAZE\_WIDTH):                  if maze[y][x] == CellType.DOT:                      dots.append((x, y))            if not dots:              return self.first\_valid\_move(maze, ghosts)            target = min(dots, key=lambda d: abs(self.x - d[0]) + abs(self.y - d[1]))            # Tìm path bằng AC-3 + backtracking          def ac3\_search(current\_pos, target\_pos, path, visited\_nodes, depth=0):              """Search với AC-3 consistency"""              if depth > 20:                  return None                if current\_pos == target\_pos:                  return path                # Lấy valid directions              valid\_dirs = []              for direction in Direction:                  dx, dy = direction.value                  new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)                    if (self.\_is\_valid\_move(new\_pos, maze) and                      new\_pos not in visited\_nodes and                      not self.\_is\_ghost\_position(new\_pos, ghosts)):                      valid\_dirs.append(direction)                # Sắp xếp theo heuristic (Manhattan distance đến target)              valid\_dirs.sort(                  key=lambda d: abs((current\_pos[0] + d.value[0]) - target\_pos[0]) +                                abs((current\_pos[1] + d.value[1]) - target\_pos[1])              )                for direction in valid\_dirs:                  dx, dy = direction.value                  new\_pos = (current\_pos[0] + dx, current\_pos[1] + dy)                    # Check arc consistency                  visited\_nodes.add(new\_pos)                  new\_path = path + [direction]                    result = ac3\_search(new\_pos, target\_pos, new\_path, visited\_nodes, depth + 1)                    if result is not None:                      return result                    visited\_nodes.remove(new\_pos)                return None            path = ac3\_search((self.x, self.y), target, [], {(self.x, self.y)})            if path and len(path) > 0:              return path[0]            return self.first\_valid\_move(maze, ghosts) |

### **1.6.4. And-Or Planning**

Mô tả ý tưởng:  
And-Or Planning là thuật toán lập kế hoạch (Planning) trong môi trường có điều kiện rẽ nhánh – nơi một hành động có thể dẫn đến nhiều kết quả khác nhau (ví dụ: thành công hoặc thất bại).

* OR nodes: đại diện cho hành động lựa chọn (Pacman chọn một trong nhiều hướng).
* AND nodes: đại diện cho tập các hành động phải thực hiện đồng thời hoặc kế tiếp nhau để đạt được mục tiêu.

Thuật toán tìm kiếm trong cấu trúc cây And-Or, xây dựng kế hoạch hoàn chỉnh bao gồm mọi nhánh có thể xảy ra.

Ứng dụng trong Pacman AI:  
And-Or Planning được dùng để Pacman lập kế hoạch hành động trong môi trường không chắc chắn, ví dụ:

* Khi Pacman có thể chọn giữa “ăn dot gần” hoặc “né Ghost”, và cần xem xét cả hai kịch bản.
* Hoặc khi một hành động (di chuyển sang phải) có thể thành công hoặc thất bại nếu Ghost chặn đường.

Tính chất:

* Hoàn chỉnh: Có, nếu tất cả các nhánh được mở rộng.
* Tối ưu: Có thể, nếu chi phí hành động được tính.
* Độ phức tạp: O(b^m) với b là số hành động, m là độ sâu kế hoạch.

Ưu/nhược điểm:

* Ưu: Mô tả rõ ràng kế hoạch trong môi trường rẽ nhánh; linh hoạt khi có điều kiện không chắc chắn.
* Nhược: Cấu trúc cây phức tạp, tốn thời gian và bộ nhớ khi môi trường có nhiều khả năng.

Thuật toán And-Or Planning viết bằng ngôn ngữ python áp dụng trong game pacman

|  |
| --- |
| def and\_or\_planning\_move(self, maze, ghosts):          """And-Or Planning - Lập kế hoạch có điều kiện"""          dots = [(x, y) for y in range(MAZE\_HEIGHT) for x in range(MAZE\_WIDTH) if maze[y][x] == CellType.DOT]          if not dots:              return self.first\_valid\_move(maze, ghosts)          target = min(dots, key=lambda d: abs(self.x - d[0]) + abs(self.y - d[1]))          # A\* but if a path would step into a ghost, record an OR branch to wait/replan          counter = 0          open\_heap = [(0, counter, self.x, self.y, [])]          closed = set()          ghost\_set = {(g.x, g.y) for g in ghosts}          while open\_heap:              f, \_, x, y, path = heapq.heappop(open\_heap)              counter += 1              if (x, y) in closed:                  continue              closed.add((x, y))              if (x, y) == target:                  if path:                      return path[0]                  else:                      return self.first\_valid\_move(maze, ghosts)              for direction in Direction:                  dx, dy = direction.value                  nx, ny = x + dx, y + dy                  if not (0 <= nx < MAZE\_WIDTH and 0 <= ny < MAZE\_HEIGHT):                      continue                  if maze[ny][nx] == CellType.WALL:                      continue                  # If the next cell is occupied by a ghost, consider it an OR (replanning) - skip for now                  if (nx, ny) in ghost\_set:                      continue                  g\_cost = len(path) + 1                  h\_cost = abs(nx - target[0]) + abs(ny - target[1])                  heapq.heappush(open\_heap, (g\_cost + h\_cost, counter, nx, ny, path + [direction]))                  counter += 1          # If planning fails (ghosts block), fallback to deterministic safe move          return self.first\_valid\_move(maze, ghosts) |

# **CHƯƠNG 2: THIẾT KẾ HỆ THỐNG VÀ GIAO DIỆN CỦA CHƯƠNG TRÌNH**

## **2.1. Cấu trúc tổng thể của chương trình**

A screen shot of a computer

AI-generated content may be incorrect.

**Tóm tắt các file chính**

| File | Vai trò |
| --- | --- |
| main.py | Điểm khởi chạy game. Gọi class Game và thực thi vòng lặp chính. |
| game.py | Chứa logic game: sinh maze, xử lý va chạm, cập nhật điểm, hiển thị UI. |
| pacman.py | Xử lý trí tuệ nhân tạo của Pacman (18 thuật toán). |
| ghost.py | Điều khiển hành vi của các quái vật (ghosts). |
| constants.py | Định nghĩa các giá trị cố định (màu sắc, kích thước, hướng, thuật toán). |
| sound\_manager.py | Quản lý và phát âm thanh cho game. |

Link github quan sát code:

## **2.2. Thiết kế giao diện người dùng**

Giao diện người dùng (UI) của trò chơi Pacman AI được thiết kế bằng Pygame, với phong cách đơn giản, dễ nhìn, tập trung vào việc quan sát quá trình Pacman di chuyển theo từng thuật toán. Người dùng có thể chọn thuật toán, theo dõi điểm số, xem cấp độ (level) và tình trạng của Pacman và Ghosts trong thời gian thực.

### **2.2.1. giao diện chính của trò chơi**

* Màn hình chính gồm khu vực mê cung (maze), điểm số, thuật toán hiện hành và trạng thái game (playing / paused / game over).
* Pacman di chuyển trên bản đồ tự sinh, trong khi các Ghost di chuyển ngẫu nhiên.
* Khi người chơi chọn thuật toán khác, giao diện sẽ hiển thị tên thuật toán mới (ví dụ: *A* Search\*, *Greedy*, *Genetic*, v.v.)

A video game with a game

AI-generated content may be incorrect.

### **2.2.2. Giao diện Menu chọn thuật toán AI**

**b. Menu chọn thuật toán AI**

* Khi nhấn phím **Tab** hoặc mở menu (ấn M), người dùng có thể chọn **thuật toán AI** mà Pacman sẽ sử dụng.
* Danh sách hiển thị gồm các nhóm như:
  + BFS, DFS, UCS, IDS, IDL
  + A\*, Greedy, Beam Search
  + Hill-Climbing, Genetic, Backtracking, AC-3, v.v.

Khi chọn, game sẽ tự cập nhật và Pacman bắt đầu chạy theo thuật toán mới.

A screenshot of a computer

AI-generated content may be incorrect.

### **2.2.3. Giao diện khi Pacman ăn chấm hoặc bị bắt**

* Khi Pacman ăn chấm (dot), điểm số được cập nhật và âm thanh được phát qua SoundManager.

A yellow object with a black face

AI-generated content may be incorrect.

* Nếu Pacman bị Ghost bắt, giao diện hiển thị hiệu ứng *“Game Over”* và âm thanh thua cuộc.

A screenshot of a video game

AI-generated content may be incorrect.

# **KẾT LUẬN**

Qua quá trình nghiên cứu và xây dựng trò chơi Pacman AI, nhóm đã triển khai thành công một hệ thống mô phỏng trí tuệ nhân tạo có tính trực quan cao, giúp người học dễ dàng quan sát và đánh giá cách các thuật toán tìm kiếm hoạt động trong môi trường thực tế. Trò chơi được phát triển bằng Python và Pygame, áp dụng các nguyên lý lập trình hướng đối tượng (OOP) để phân tách rõ ràng giữa các thành phần như Game, PacmanAI, Ghost, SoundManager và Constants. Đặc biệt, hệ thống tích hợp 18 thuật toán AI thuộc nhiều nhóm như tìm kiếm không thông tin, có thông tin, đối kháng, cục bộ, CSP và tiến hóa, giúp Pacman có thể tự di chuyển thông minh, tối ưu hóa đường đi và phản ứng linh hoạt trước sự di chuyển ngẫu nhiên của Ghost. Việc kết hợp giữa âm thanh, đồ họa, thuật toán và logic giúp dự án không chỉ mang tính thực hành cao mà còn hỗ trợ trực quan hóa các khái niệm phức tạp của môn Trí tuệ nhân tạo.

Mặc dù còn một số hạn chế như hiệu suất của các thuật toán CSP và Genetic chưa tối ưu, hay Ghost chưa có trí tuệ thật sự để tạo ra đối kháng sâu, nhưng dự án Pacman AI đã hoàn thành xuất sắc mục tiêu ban đầu, thể hiện sự hiểu biết vững chắc về các thuật toán tìm kiếm, khả năng áp dụng lý thuyết vào thực tế, và tư duy thiết kế phần mềm rõ ràng. Trong tương lai, hệ thống có thể mở rộng bằng cách tối ưu tốc độ xử lý, bổ sung AI cho Ghost, trực quan hóa heuristic hoặc phát triển thành công cụ giảng dạy tương tác cho sinh viên ngành Công nghệ thông tin. Dự án này không chỉ giúp nhóm em củng cố kiến thức về AI và lập trình, mà còn thể hiện cách mà trí tuệ nhân tạo có thể được nhìn thấy, trải nghiệm và hiểu được thông qua từng hành động của Pacman trong mê cung.

Định hướng phát triển trong tương lai:

* Cải thiện hiệu suất:
  + Áp dụng đa luồng (multithreading) hoặc tối ưu heuristic để giảm thời gian tìm kiếm.
  + Tận dụng GPU (CUDA / OpenCL) để tăng tốc tính toán nếu triển khai trên hệ thống mạnh hơn.
* Nâng cao trí tuệ của Ghost:
  + Cài đặt thuật toán A\* hoặc Minimax cho Ghost, giúp tạo ra trò chơi đối kháng đúng nghĩa (*Pacman vs AI Ghosts*).
  + Thêm cơ chế “phối hợp” giữa các Ghost – ví dụ, một Ghost đuổi, một Ghost chặn đường.
* Tăng tính tương tác và tùy chỉnh:
  + Cho phép người dùng điều chỉnh tham số thuật toán (depth, population size, mutation rate, heuristic weight) trực tiếp trong menu.
  + Tạo bảng thống kê tự động về hiệu năng mỗi thuật toán sau khi chơi xong.
* Trực quan hóa sâu hơn:
  + Hiển thị đường đi mà thuật toán tìm được (bằng highlight trên mê cung).
  + Màu hóa bản đồ theo giá trị heuristic (h(n)) hoặc cost (g(n)) để giúp người học hiểu sâu hơn.
* Phát triển hướng học tập / nghiên cứu:
  + Biến game thành module giảng dạy AI trong trường, cho phép sinh viên thử nghiệm nhiều thuật toán khác nhau.
  + Mở rộng sang môi trường 3D hoặc web-based (sử dụng Unity, WebGL, hay Pyodide).

# **TÀI LIỆU THAM KHẢO**

1. Alberto Artasanchez Prateek Joshi, Artificial Intelligence with Python, Packt Publishing Ltd, 2nd Edition, 2020.
2. Stuart J. Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 4th Edition, Pearson Education ©, 2021.
3. Từ Minh Phương, Giáo trình Nhập môn trí tuệ nhân tạo, Học Viện Công nghệ Bưu chính Viễn thông, 2014.
4. George F. Luger, William A. Stubblefield – Albuquerque – Artificial Intelligence – Wesley Publishing Company, Inc – 1997.