TRƯỜNG ĐH SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH
KHOA CÔNG NGHỆ THÔNG TIN

TIỂU LUẬN CUỐI KỲ

MÔN HỌC: TRÍ TUỆ NHÂN TẠO
ĐỀ TÀI: PACMAN GAME

Mã môn học: ARIN330585_04

Nhóm thực hiện: Nhóm 13

GVHD: TS. Phan Thị Huyền Trang

Danh sách sinh viên thực hiện

STT	MSSV	Họ và Tên
1	23110328	Lê Văn Chiến Thắng
2	23110301	Võ Thanh Sang
3	23110272	Trịnh Nguyễn Hoàng Nguyên

TP. Hồ Chí Minh, 12 tháng 5 năm 2025

DANH SÁCH SINH VIÊN THAM GIA THỰC HIỆN
HỌC KÌ 2 NĂM HỌC 2024 – 2025

Mã lớp: ARIN330585_04

Tên môn học: Trí Tuệ Nhân Tạo

Đề tài: Pacman Game

Nhóm thực hiện: Nhóm 13

DANH SÁCH SINH VIÊN THAM GIA

STT	**MSSV**	**Họ và tên**	**Nhiệm vụ**	**Tỷ lệ hoàn thành**
1	23110328	Lê Văn Chiến Thắng	Thuật toán: Backtracking, Minimax, Backtracking_Ver2, AlphaBetaPruning, Steepest Ascent Hill Climbing	100%
2	23110301	Võ Thanh Sang	Thuật toán: Q-Learning, Simulated Annealing, Beam Search, Greedy Search	100%
3	23110272	Trịnh Nguyễn Hoàng Nguyên	Thuật toán: And-Or Tree Search, UCS, DFS, BFS, A* Search	100%

Ghi chú:

- Nhóm trưởng: Lê Văn Chiến Thắng

NHẬN XÉT CỦA GIÁO VIÊN

……
……
……
……

Ngày … Tháng … Năm 2025

Giáo viên chấm điểm

MỤC LỤC

DANH MỤC VIẾT TẮT

Từ viết tắt	**Ý nghĩa đầy đủ**
AI	Artificial Intelligence
UCS	Uniform Cost Search
DFS	Depth-First Search
AC-3	Arc Consistency Algorithm 3
BFS	Breadth First Search

DANH MỤC HÌNH ẢNH

LỜI CẢM ƠN

Lời đầu tiên, nhóm chúng em muốn gửi lời cảm ơn chân thành đến Ban Giám hiệu nhà trường vì đã đưa môn Trí tuệ Nhân tạo vào chương trình học. Đây là một môn học mới mẻ, nhưng rất thú vị và thực sự cần thiết trong thời đại công nghệ bùng nổ như hiện nay. Nhờ vậy mà chúng em có cơ hội được tiếp cận với những kiến thức hiện đại mà trước giờ chỉ nghe trong sách báo hoặc trên mạng.

Tụi em cũng rất biết ơn cô **Phan Thị Huyền Trang**, người đã đồng hành và giảng dạy môn học này với sự tận tâm và nhiệt huyết. Cô luôn cố gắng truyền đạt kiến thức một cách dễ hiểu, vừa logic vừa gần gũi, giúp chúng nắm được những thuật toán, khái niệm phức tạp của AI. Ngoài bài giảng lý thuyết, cô còn đưa ra nhiều ví dụ thực tế giúp tụi em hiểu rõ hơn về cách mà AI đang được ứng dụng trong đời sống.

Trong quá trình làm đồ án, nhóm đã rất cố gắng để vận dụng những gì học được vào thực tế. Có lúc bí, có lúc rối, nhưng nhờ kiến thức từ cô giảng và sự hỗ trợ kịp thời mà chúng em đã hoàn thành được sản phẩm này. Dù chưa phải là hoàn hảo, nhưng đó là cả một quá trình học hỏi, thử sai và tiến bộ từng chút một.

Cuối cùng, chúng em muốn gửi lời cảm ơn chân thành đến cô vì đã luôn đồng hành, động viên và hỗ trợ tụi em suốt cả học kỳ. Những gì cô truyền đạt không chỉ là kiến thức, mà còn là động lực để tụi em tin rằng mình có thể làm được – chỉ cần cố gắng và không bỏ cuộc.

Cảm ơn cô rất nhiều!

PHẦN 1. MỞ ĐẦU

1.1. Phát biểu bài toán

Trong thời đại mà AI đang phát triển cực nhanh, những ứng dụng có yếu tố trí tuệ nhân tạo trong game ngày càng được quan tâm vì mang lại tính tương tác cao và trải nghiệm thông minh hơn cho người chơi. Thay vì game chỉ chạy theo kịch bản cố định, giờ đây các nhân vật có thể tự suy nghĩ, tự phản ứng và đó là điều làm nên sự thú vị.

Từ ý tưởng đó, nhóm chúng em quyết định xây dựng một phiên bản PacMan đặc biệt: Pacman tự điều khiển bằng AI. Trong game này, PacMan sẽ không cần người chơi điều khiển bằng tay, mà sẽ tự tìm đường đi tối ưu để có thể ăn được nhiều hạt thức ăn nhiều nhất, đồng thời tránh quái vật đuổi bắt. Điều đặc biệt là PacMan sẽ chọn đường đi bằng các thuật toán tìm kiếm thông minh nghĩa là nhân vật có khả năng "tính toán" trước khi di chuyển.

Bài toán đặt ra là làm sao để PacMan vừa di chuyển hiệu quả, vừa không bị bắt, đồng thời phải phản ứng nhanh với các thay đổi trong bản đồ. Nghe có vẻ đơn giản, nhưng để làm được điều đó, cần phải kết hợp nhiều kiến thức từ lý thuyết AI vào thực tế. Và đó cũng chính là thử thách mà tụi em muốn chinh phục trong dự án này.

1.2. Mục đích, yêu cầu cần thực hiện

Mục tiêu của đề tài này là tạo ra một hệ thống PacMan có thể tự chơi game mà không cần người điều khiển. Cụ thể là nhân vật PacMan sẽ được lập trình để biết tự động tìm đường đi, chọn hướng di chuyển hợp lý để vừa ăn hết các hạt thức ăn, vừa né tránh các con ma đang truy đuổi.

Ở mỗi cấp độ khác nhau, chúng em sẽ dùng một thuật toán tìm kiếm khác nhau như BFS, DFS, A*, Local Search, Minimax,... Nhờ đó, chúng em có thể so sánh cách mà mỗi thuật toán xử lý tình huống, xem thuật toán nào nhanh hơn, an toàn hơn hoặc hiệu quả hơn khi PacMan chơi trong các map có độ khó tăng dần.

Bên cạnh phần AI, nhóm cũng xây dựng một giao diện game đơn giản nhưng trực quan bằng thư viện pygame, giúp người dùng dễ dàng quan sát các đối tượng trong game như bản đồ, các bức tường, vị trí thức ăn, quái vật và nhân vật PacMan.

Một trong những yêu cầu tụi em đặt ra là PacMan phải di chuyển mượt mà, không bị giật lag, phản ứng kịp thời khi có ma đuổi hay bị kẹt đường. Ngoài ra, thuật toán cũng phải giúp PacMan ưu tiên đi ăn các hạt dễ lấy trước, tránh vòng vèo gây tốn thời gian hoặc tự đưa mình vào ngõ cụt.

Tóm lại, nhóm không chỉ muốn PacMan "biết đi", mà còn phải đi thông minh và sống sót càng lâu càng tốt, đúng tinh thần của một game có yếu tố AI.

1.3. Phạm vi và đối tượng

Đối tượng:

Dự án tập trung vào ba nhóm đối tượng chính:

- **PacMan:** Là nhân vật trung tâm của game, được lập trình như một tác nhân AI. PacMan sẽ tự động phân tích bản đồ, xác định vị trí thức ăn và đưa ra quyết định di chuyển hợp lý để ăn hết thức ăn và tránh va chạm với quái vật.
- **Ghost (quái vật):** Là đối thủ chính của PacMan. Ghost không đứng yên mà di chuyển liên tục theo hai cách khác nhau tùy level: random hoặc sử dụng A* để đuổi theo PacMan. Việc di chuyển của Ghost tạo nên yếu tố nguy hiểm và buộc PacMan phải có chiến lược né tránh thông minh.
- **Người chơi:** Không trực tiếp điều khiển nhân vật, mà sẽ tương tác qua giao diện menu để chọn bản đồ, chọn cấp độ và chọn thuật toán AI, từ đó quan sát quá trình PacMan tự động di chuyển và hoàn thành nhiệm vụ.

Phạm vi:

Dự án được triển khai với:

- **4 cấp độ (Level 1 đến Level 4)**, mỗi level có bản đồ riêng và độ khó tăng dần. Càng lên cao, số lượng ghost tăng, bản đồ phức tạp hơn và ghost di chuyển thông minh hơn.
- **Áp dụng 7 thuật toán tìm kiếm AI** vào game để điều khiển hành vi của PacMan, bao gồm:
  - ✓ Nhóm 1: Tìm kiếm không có thông tin (Uninformed Search)
    - Breadth-First Search (BFS): tìm kiếm theo chiều rộng.
    - Depth-First Search (DFS): tìm kiếm theo chiều sâu.
    - Uniform Cost Search (UCS): tìm kiếm theo chi phí.
  - ✓ Nhóm 2: Tìm kiếm có thông tin (Informed Search)
    - Greedy Search: tìm kiếm tham lam, ưu tiên các trạng thái gần thức ăn nhất.
    - A* Search: Kết hợp giữa chi phí thực tế và heuristic để tìm đường đi tối ưu.
  - ✓ Nhóm 3: Tìm kiếm cục bộ (Local Search)
    - Steepest-Ascent Hill Climbing: Luôn chọn trạng thái lân cận có giá trị tốt nhất.
    - Simulated Annealing: Cho phép chọn trạng thái kém hơn để tránh bị kẹt.
    - Beam Search: Mở rộng giới hạn số trạng thái trong mỗi bước dựa trên euristic.

- ✓ Nhóm 4: Tìm kiếm trong môi trường phức tạp (Nondeterministic Search)
  - Tree Search AND – OR: Tìm kiếm theo cây gồm các lựa chọn (OR) và các tình huống bắt buộc phải đúng đồng thời (AND).
- ✓ Nhóm 5: Tìm kiếm trong môi trường có ràng buộc (Constraint Satisfaction Search)
  - Backtracking Search + Forward Checking: Quay lui kết hợp kiểm tra trước để tránh thử các giá trị sai.
  - Backtracking Search + AC-3 (Ver2): Kết hợp quay lui với thuật toán AC-3 để loại bỏ sớm các giá trị không hợp lệ.
- ✓ Nhóm 6: Học tăng cường (Reinforcement Learning)
  - Q-Learning: Học dần dần qua các lần thử sai để xây dựng bảng hành động tốt nhất (Q-table).
- ✓ Nhóm 7: Tìm kiếm đối kháng (Adversarial Search)
  - Minimax: Chọn nước đi tốt nhất trong tình huống đối thủ luôn chơi theo hướng bất lợi.
  - AlphaBetaPruning: Rút gọn quá trình Minimax bằng cách loại bỏ các nhánh không cần thiết mà vẫn giữ kết quả đúng

Các thuật toán sẽ được phân phối theo từng cấp độ để đánh giá hiệu quả trong việc giúp PacMan hoàn thành nhiệm vụ trong môi trường có yếu tố nguy hiểm và thay đổi liên tục.

PHẦN 2. CƠ SỞ LÝ THUYẾT

2.1. Giới thiệu về Trí tuệ nhân tạo trong trò chơi

Việc ứng dụng trí tuệ nhân tạo trong trò chơi có thể đã bắt đầu từ lâu do sự phát triển của con người. Trí tuệ nhân tạo (AI) đóng góp đáng kể vào việc nâng cao trải nghiệm chơi game, từ việc phát triển các nhân vật không người chơi (NPC) thông minh đến việc mô phỏng các môi trường phức tạp. AI trong trò chơi, từ góc độ của người chơi, tạo ra ấn tượng rằng các NPC sở hữu sự nhận thức và trí thông minh thực sự. Trong thực tế, khả năng tạo ra những trải nghiệm tuyệt vời như vậy là kết quả của việc lập kế hoạch kỹ lưỡng bởi các nhà phát triển game. Điều này không có nghĩa là việc triển khai AI trong trò chơi là một nhiệm vụ đơn giản. Các nhà sáng tạo trò chơi cần phải dự đoán tất cả các kết quả có thể xảy ra để đưa ra các giải pháp hiệu quả nhất trong những kịch bản đầy rủi ro tiếp theo.

Trong bối cảnh trò chơi Pacman AI, trí tuệ nhân tạo được triển khai nhằm điều khiển nhân vật Pacman một cách thông minh khi di chuyển trong mê cung, tìm kiếm và thu thập tất cả các viên thức ăn trong khi phải tránh né các con ma đang truy đuổi. Để đạt được điều này, trò chơi tích hợp nhiều thuật toán tìm kiếm và ra quyết định như Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), Greedy Search, Beam Search, Simulated Annealing, Stochastic Hill Climbing, Backtracking with Forward checking, Backtracking with AC-3, Minimax, Alpha-Beta Pruning, Q-Learning, A* và AND-OR Search. Những thuật toán này đại diện cho các nhóm kỹ thuật tìm kiếm không có thông tin, có thông tin, tìm kiếm trong môi trường ràng buộc, tìm kiếm trong môi trường phức tạp , tìm kiếm cục bộ, đối kháng và học củng cố. Chúng cho phép Pacman đưa ra các hành động phù hợp trong từng tình huống cụ thể, từ việc tối ưu hóa lộ trình đến thức ăn, tránh ghost, cho đến việc học hỏi từ kinh nghiệm trong quá trình chơi để cải thiện hiệu suất.

Việc kết hợp các thuật toán AI trong trò chơi không chỉ tạo nên trải nghiệm tương tác hấp dẫn, mà còn góp phần minh họa rõ ràng cách các phương pháp trí tuệ nhân tạo có thể áp dụng vào các môi trường giả lập. Ngoài việc điều khiển hành vi nhân vật, AI trong game còn đóng vai trò trong việc thiết kế cấp độ chơi, tạo nên môi trường sinh động và mang lại trải nghiệm cá nhân hóa cho người chơi. Trong dự án này, AI chủ yếu tập trung giải quyết bài toán tìm đường – một vấn đề kinh điển trong lĩnh vực trí tuệ nhân tạo – nhằm thể hiện rõ sự khác biệt và hiệu quả của từng thuật toán trong môi trường trò chơi thực tế.

2.2. Thư viện hỗ trợ lập trình, ngôn ngữ lập trình

2.2.1. Ngôn ngữ lập trình Python

Python là một trong những ngôn ngữ lập trình phổ biến và được ưa chuộng nhất hiện nay nhờ cú pháp đơn giản, dễ đọc, dễ học và khả năng ứng dụng rộng rãi. Ban đầu được thiết kế với mục tiêu giúp lập trình viên viết mã rõ ràng và ngắn gọn hơn, Python ngày nay đã trở thành một công cụ mạnh mẽ trong nhiều lĩnh vực khác nhau, đặc biệt là trong trí tuệ nhân tạo và phát triển trò chơi.

Trong các dự án game có tích hợp các thuật toán AI như Pacman, Python được đánh giá là một lựa chọn phù hợp nhờ vào các đặc điểm nổi bật như khả năng tổ chức mã nguồn linh hoạt, hỗ trợ đa dạng thư viện và cấu trúc dữ liệu. Một trong những thư viện phổ biến được sử dụng để phát triển game 2D bằng Python là pygame. Thư viện này hỗ trợ hiển thị đồ họa, xử lý sự kiện từ bàn phím, âm thanh và tạo hiệu ứng chuyển động cho các đối tượng trong trò chơi.

Bên cạnh đó, Python cũng tích hợp sẵn các thư viện như heapq, deque, random, giúp việc triển khai các thuật toán như tìm kiếm theo chiều rộng (BFS), tìm kiếm chi phí đồng nhất (UCS), tìm kiếm có thông tin (A*), leo đồi (Hill Climbing), mô phỏng ủ nhiệt (Simulated Annealing), học củng cố (Q-Learning) và nhiều thuật toán khác trở nên dễ dàng và trực quan.

Với cộng đồng phát triển lớn mạnh và tài liệu học tập phong phú, Python mang lại lợi thế rõ rệt trong việc thử nghiệm và triển khai các thuật toán trí tuệ nhân tạo trong môi trường mô phỏng như game. Đây cũng là lý do vì sao ngôn ngữ này thường được lựa chọn cho các dự án học thuật, nghiên cứu và phát triển phần mềm giáo dục.

2.2.2. Trình soạn thảo mã nguồn Visual Studio Code

Visual Studio Code (VS Code) là một trình soạn thảo mã nguồn miễn phí và đa nền tảng do Microsoft phát triển, lần đầu ra mắt vào năm 2015. Không giống như các môi trường phát triển tích hợp (IDE) truyền thống, VS Code được thiết kế để cung cấp trải nghiệm viết mã nhanh, nhẹ nhưng vẫn mạnh mẽ nhờ khả năng mở rộng linh hoạt và tích hợp nhiều tính năng hiện đại. Theo Wikipedia, công cụ này đã nhanh chóng trở thành một trong những trình soạn thảo mã phổ biến nhất thế giới, được sử dụng rộng rãi trong cả cộng đồng chuyên nghiệp lẫn học thuật.

Một trong những điểm nổi bật giúp VS Code trở thành lựa chọn lý tưởng trong phát triển game là hệ sinh thái tiện ích mở rộng (extensions) phong phú. Đối với ngôn ngữ Python – ngôn ngữ chính được sử dụng trong trò chơi Pacman AI – VS Code hỗ trợ rất tốt thông qua các tiện ích như Python Extension và Pylance. Những tiện ích này cung cấp chức năng tự động hoàn thành mã, phát hiện lỗi cú

pháp, kiểm tra kiểu dữ liệu, và đặc biệt là tích hợp trình gỡ lỗi trực tiếp, rất hữu ích trong quá trình viết và thử nghiệm các thuật toán trí tuệ nhân tạo.

Trong quá trình xây dựng trò chơi Pacman, VS Code cho phép tổ chức mã nguồn theo cấu trúc thư mục rõ ràng, hỗ trợ quản lý nhiều file thuật toán như BFS, DFS, A*, Q-Learning, Minimax,...cũng như các lớp đối tượng như nhân vật, ghost, tường, thức ăn,...Việc tích hợp terminal ngay trong giao diện chính giúp người phát triển có thể biên dịch, chạy thử hoặc theo dõi lỗi trực tiếp mà không cần chuyển đổi ứng dụng. Ngoài ra, VS Code còn hỗ trợ hệ thống quản lý phiên bản Git, cho phép dễ dàng theo dõi thay đổi trong mã nguồn, đặc biệt hữu ích khi làm việc nhóm hoặc triển khai theo từng giai đoạn.

Giao diện người dùng của VS Code cũng được đánh giá cao nhờ thiết kế hiện đại, dễ sử dụng, hỗ trợ tùy chỉnh giao diện theo chủ đề, chia nhiều cửa sổ làm việc, hỗ trợ kéo thả file trực tiếp. Với các tính năng nổi bật kể trên, Visual Studio Code không chỉ là công cụ phù hợp cho lập trình chuyên nghiệp, mà còn là lựa chọn đáng tin cậy trong các dự án học tập và nghiên cứu như trò chơi Pacman ứng dụng AI.

2.2.3. Những thư viện chính hỗ trợ cho việc lập trình

Trong quá trình xây dựng trò chơi Pacman tích hợp trí tuệ nhân tạo, nhiều thư viện Python đã được sử dụng để hỗ trợ việc xử lý logic, đồ họa, cấu trúc dữ liệu và thuật toán. Dưới đây là những thư viện quan trọng được sử dụng xuyên suốt trong các tệp mã nguồn:

- ✓ **pygame:** Thư viện chính để xây dựng giao diện đồ họa, xử lý sự kiện, vẽ bản đồ, hiển thị Pacman, ghost, thức ăn và điều khiển toàn bộ vòng lặp game. pygame xuất hiện trong hầu hết các file như main.py, Player.py, Menu.py..,
- ✓ **collections:** Cụ thể là deque, được sử dụng trong các thuật toán tìm kiếm như BFS và DFS để triển khai hàng đợi và ngăn xếp hiệu quả.
- ✓ **heapq và queue.PriorityQueue:** Được dùng trong các thuật toán có yếu tố chi phí như Uniform Cost Search (UCS), A*, Beam Search, Q-Learning,... nhằm xử lý hàng đợi ưu tiên.
- ✓ **random:** Hỗ trợ sinh giá trị ngẫu nhiên trong các thuật toán như Stochastic Hill Climbing, Simulated Annealing, và Random Ghost Movement trong main.py.
- ✓ **math:** Một số thuật toán sử dụng các hàm toán học để tính xác suất, hàm mục tiêu hoặc kiểm soát nhiệt độ (trong Simulated Annealing).
- ✓ **copy:** Thư viện này được dùng để sao chép sâu (deep copy) trạng thái bản đồ khi cần giả lập nhiều nhánh đi trong các thuật toán như Backtracking, Minimax, Alpha-Beta Pruning.

- ✓ **os:** Được sử dụng trong Menu.py để duyệt các file bản đồ từ thư mục Input, phục vụ cho chức năng chọn level và map.

Ngoài các thư viện chuẩn, một số module tự định nghĩa như utils.py cũng đóng vai trò quan trọng, cung cấp các hàm kiểm tra hợp lệ vị trí (isValid, isValid2), tính khoảng cách Manhattan, tìm thức ăn gần nhất và xử lý logic phụ trợ cho nhiều thuật toán.

Việc kết hợp hiệu quả giữa các thư viện chuẩn và thư viện do người lập trình tự xây dựng giúp đảm bảo mã nguồn vừa tối ưu, dễ mở rộng, vừa dễ đọc, hỗ trợ tốt cho quá trình kiểm thử và triển khai nhiều thuật toán AI một cách tách biệt và rõ ràng.

2.3. Tổng quan về Hill Climbing Search

2.3.1. Khái niệm và nguyên lý hoạt động

Thuật toán Hill Climbing, hay còn gọi là "leo đồi", là một **kỹ thuật tìm kiếm cục bộ đơn giản** nhưng khá thú vị và được ứng dụng rộng rãi trong các bài toán tối ưu trong trí tuệ nhân tạo. Ý tưởng cốt lõi của nó cũng giống như tên gọi: ta tưởng tượng mình đang đứng trên một sườn núi, trong một màn sương mù dày đặc, và mục tiêu là phải leo lên đỉnh cao nhất có thể – nhưng khổ cái là mình không nhìn thấy xa, chỉ quan sát được các điểm xung quanh gần đó. Không có bản đồ, không có trí nhớ, chỉ biết "thấy chỗ nào cao hơn thì bước tới".

Về mặt kỹ thuật, Hill Climbing bắt đầu từ một trạng thái khởi đầu. Từ đó, nó sẽ tìm trong các trạng thái lân cận (neighboring states) xem có trạng thái nào "tốt hơn" – tức là có giá trị đánh giá cao hơn. Nếu có, nó sẽ dịch chuyển đến đó và lặp lại quá trình. Còn nếu không tìm thấy hướng đi nào tốt hơn nữa, thì đơn giản là đứng lại – vì đã tới đỉnh (dù có thể chỉ là đỉnh cục bộ chứ không phải đỉnh cao nhất toàn cục).

Điểm đặc biệt ở Hill Climbing là nó **không cần nhớ quá khứ**, cũng chẳng xây dựng cây tìm kiếm đồ sộ như các thuật toán khác (ví dụ như A*). Nó chỉ quan tâm đến hiện tại và những gì "ngay bên cạnh", nên **tiết kiệm được rất nhiều bộ nhớ.** Tuy nhiên, chính điều đó lại là điểm yếu chết người: Hill Climbing **rất dễ bị mắc kẹt** ở những tình huống "nửa vời".

Cụ thể, có ba tình huống dễ khiến Hill Climbing "chết đứng":

1. ***Đỉnh cục bộ (Local Maximum):*** là điểm mà không có lân cận nào tốt hơn, nhưng lại không phải lời giải tốt nhất toàn cục.
2. ***Cao nguyên (Plateau):*** là vùng phẳng lì mà mọi trạng thái lân cận đều có giá trị giống nhau, khiến thuật toán không biết nên bước hướng nào.

3. ***Sườn dốc (Ridge):*** là khu vực mà lời giải tối ưu nằm ở xa nhưng đường đi tới đó lại không hiện ra ngay trong các bước lân cận.

"Hill-climbing search keeps only a single current state in memory, and it is the most common form of local search. It does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia." (Russell & Norvig, 2020, p. 236). Đó là câu ví von rất hay của Russell và Norvig rằng: "Hill Climbing giống như việc cố tìm đỉnh Everest khi bạn bị mất trí nhớ và lại còn đang ở trong một làn sương mù dày đặc." Nghe có vẻ bi quan, nhưng nó diễn tả rất đúng cách mà thuật toán này hoạt động.

Nguyên lý hoạt động tổng quát của Hill Climbing:

- ✓ Bắt đầu từ một trạng thái ban đầu.
- ✓ Đánh giá tất cả các trạng thái lân cận.
- ✓ Chọn trạng thái tốt nhất (hoặc một trạng thái tốt hơn).
- ✓ Di chuyển đến trạng thái đó nếu nó cải thiện so với hiện tại.
- ✓ Lặp lại, cho đến khi không thể cải thiện thêm → dừng.

Hill Climbing hoạt động **dựa trên hàm đánh giá (evaluation function)**. Với các bài toán cần tối đa hóa, thuật toán sẽ luôn tìm trạng thái có giá trị lớn hơn. Đôi khi, người ta sử dụng âm của hàm chi phí để biến bài toán tối thiểu hóa thành tối đa hóa, nhằm thuận tiện cho việc "leo đồi".

Ví dụ: bài toán 8 quân hậu: Một ứng dụng cổ điển của Hill Climbing là bài toán đặt 8 quân hậu sao cho không con nào tấn công nhau. Trạng thái lân cận được tạo ra bằng cách di chuyển từng quân hậu sang các vị trí khác trong cùng cột. Giá trị đánh giá ở đây là số lượng cặp quân hậu không tấn công nhau. Thuật toán sẽ lần lượt chọn nước đi sao cho tăng được giá trị này, đến khi không còn cải thiện nào nữa thì dừng lại – dù có thể mới chỉ là một lời giải gần đúng.

2.3.2. Phân loại thuật toán Hill Climbing

2.3.2.1. Simple Hill Climbing (Leo đồi đơn giản)

Simple Hill Climbing – hay còn gọi là leo đồi đơn giản – là biến thể cơ bản nhất của thuật toán Hill Climbing. Ý tưởng của nó cũng mộc mạc như chính cái tên: chỉ cần thấy có hướng nào "cao hơn hiện tại" là bước tới liền, không cân nhắc gì thêm.

Cụ thể, ở mỗi vòng lặp, thuật toán xét các trạng thái lân cận (neighbors) của trạng thái hiện tại, rồi chọn ngay trạng thái đầu tiên có giá trị đánh giá tốt hơn mà nó tìm thấy. Nếu tìm được, nó lập tức di chuyển đến đó mà không cần so sánh với các lựa chọn khác. Ngược lại, nếu không có trạng thái nào tốt hơn, thuật toán sẽ dừng

lại – coi như đã lên tới đỉnh (dù có thể chỉ là một cái "gò đất" trong khi lời giải tối ưu ở tít phía xa).

Trong sách, Russell & Norvig mô tả Simple Hill Climbing như sau: "Hill-climbing search keeps only a single current state in memory, and it is the most common form of local search. It does not look ahead beyond the immediate neighbors of the current state." (Russell & Norvig, 2020, p. 236). Điều này làm cho Simple Hill Climbing rất nhanh – vì nó không cần duyệt hết tất cả lân cận, chỉ cần tìm được một cái "tạm được" là đi ngay. Tuy nhiên, chính sự đơn giản này lại khiến nó trở nên dễ bị kẹt ở các cực trị địa phương. Nếu trạng thái đầu tiên mà nó gặp tốt hơn một chút, nhưng không phải tốt nhất trong số các lựa chọn, thì thuật toán vẫn chọn nó mà bỏ qua các lựa chọn tiềm năng hơn.

Hãy tưởng tượng một người đang leo núi nhưng lại có tính hấp tấp: cứ thấy đoạn đường trước mặt cao hơn là leo ngay, không cần biết có con đường nào khác dễ đi hơn hoặc dẫn tới đỉnh cao hơn. Nếu đi đúng hướng thì không sao, còn nếu đi nhầm thì chỉ có nước... đứng yên ở lưng chừng núi mà nghĩ mình đã đến đỉnh.

Nguyên lý hoạt động:

1. Bắt đầu tại một trạng thái ban đầu.
2. Duyệt qua từng trạng thái lân cận.
3. Nếu thấy một trạng thái tốt hơn → di chuyển đến đó ngay.
4. Nếu không tìm được trạng thái nào tốt hơn → dừng.

Ưu điểm:

- ✓ Rất dễ cài đặt, logic đơn giản.
- ✓ Tốc độ thực thi nhanh vì chỉ xét đến trạng thái tốt đầu tiên.

Nhược điểm:

- ✓ Không đảm bảo tìm ra lời giải tốt nhất.
- ✓ Dễ bị "kẹt cứng" ở cực trị địa phương hoặc cao nguyên.
- ✓ Không tận dụng hết thông tin từ các lân cận.

Trong thực tế, Simple Hill Climbing chỉ phù hợp với không gian trạng thái đơn giản, ít cạm bẫy (tức là ít đỉnh cục bộ). Nếu bài toán có cấu trúc phức tạp, nhiều đường cụt hoặc vùng bằng phẳng, thì thuật toán này thường không hiệu quả.

2.3.2.2. Steepest-Ascent Hill Climbing (Leo đồi dốc nhất)

Steepest-Ascent Hill Climbing – hay còn gọi là "leo đồi dốc nhất" – là biến thể cải tiến của thuật toán Hill Climbing cơ bản. Khác với Simple Hill Climbing vốn chọn ngay trạng thái đầu tiên tốt hơn, Steepest-Ascent lại thận trọng hơn: nó sẽ duyệt qua toàn bộ các trạng thái lân cận hiện tại, đánh giá từng cái một, rồi chọn ra trạng thái tốt nhất trong số đó để di chuyển tới.

Nói cách khác, thay vì "vừa thấy chỗ nào cao là leo liền", Steepest-Ascent sẽ "soi hết bốn phía trước đã, rồi mới quyết định hướng nào dốc nhất để leo". Đây là cách giúp thuật toán tránh được những quyết định hấp tấp, và có khả năng tiếp cận gần hơn với lời giải tối ưu nếu không gian trạng thái cho phép.

Russell & Norvig mô tả: "Hill climbing is sometimes called greedy local search because it grabs a better state as soon as it sees one, but selects the best of the successors in terms of evaluation function."(Russell & Norvig, 2020, p. 237)

Điều này cho thấy rằng, Steepest-Ascent vẫn là tìm kiếm cục bộ mang tính tham lam, nhưng nó ít "mù quáng" hơn so với Simple Hill Climbing. Tuy nhiên, dù chọn hướng đi kỹ càng hơn, thuật toán vẫn không tránh được những hạn chế cố hữu của Hill Climbing nói chung – đặc biệt là khi rơi vào các vùng cao nguyên bằng phẳng hoặc đỉnh cục bộ sâu.

Nguyên lý hoạt động:

1. Bắt đầu tại một trạng thái hiện tại.
2. Sinh tất cả các trạng thái lân cận.
3. Đánh giá tất cả, chọn ra trạng thái có giá trị cao nhất.
4. Nếu trạng thái đó tốt hơn hiện tại, chuyển tới nó.
5. Nếu không có lân cận nào tốt hơn, dừng lại (đã "kẹt").

Ưu điểm:

- ✓ So với biến thể đơn giản, khả năng lựa chọn chính xác hướng đi cao hơn, vì luôn chọn phương án tốt nhất có thể.
- ✓ Tốt hơn trong các không gian trạng thái có nhiều nhánh xấu xen kẽ.

Nhược điểm:

- ✓ Vẫn có thể kẹt ở local maximum hoặc plateau nếu các trạng thái xung quanh đều có giá trị không cao hơn hoặc bằng nhau.
- ✓ Mỗi bước phải duyệt hết các trạng thái lân cận → chậm hơn Simple Hill Climbing.

So sánh ngắn với Simple Hill Climbing:

Đặc điểm	Simple HC	Steepest-Ascent HC
Cách chọn lân cận	Gặp cái tốt là đi	Chọn cái tốt nhất trong tất cả
Tốc độ	Nhanh hơn	Chậm hơn do phải duyệt hết
Dễ bỏ qua lựa chọn tốt hơn sau	Có	Giảm được tình trạng này
Dễ kẹt local max	Có	Vẫn có nhưng ít hơn

2.3.2.3. Stochastic Hill Climbing (Leo đồi ngẫu nhiên)

Stochastic Hill Climbing là một trong những biến thể linh hoạt và thực tế hơn của thuật toán leo đồi. Khác với phương pháp Steepest-Ascent vốn tìm trong tất cả các hướng đi lân cận để chọn ra đường tốt nhất, thì Stochastic Hill Climbing không duyệt toàn bộ mà chọn ngẫu nhiên một bước đi lên trong số các hướng tốt hơn hiện tại.

Điểm đặc trưng của biến thể này nằm ở việc chấp nhận yếu tố xác suất trong quá trình ra quyết định. Thay vì luôn chọn cái "tốt nhất", thuật toán sẽ đánh giá các nước đi có thể cải thiện trạng thái hiện tại, rồi chọn một trong số đó một cách ngẫu nhiên. Tùy vào cách xây dựng thuật toán, xác suất lựa chọn này có thể được điều chỉnh dựa trên độ dốc của trạng thái – tức là những hướng "leo dốc mạnh hơn" có thể được ưu tiên hơn một chút.

Sách Artificial Intelligence: A Modern Approach trình bày rõ rằng: "Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move." (Russell & Norvig, 2020, p. 240). Tuy điều này khiến cho thuật toán chậm hội tụ hơn so với Steepest-Ascent, nhưng lại có ưu điểm là tăng khả năng thoát khỏi bẫy local maximum hoặc vượt qua các cao nguyên bằng phẳng (plateaus). Trong một số không gian trạng thái có nhiều hướng di chuyển hợp lệ, giải pháp ngẫu nhiên đôi khi lại tìm được lời giải tốt hơn.

Ví dụ, trong một bài toán có hàng ngàn trạng thái kế tiếp (successors), việc tính toán hết tất cả để tìm hướng đi tốt nhất sẽ tốn thời gian. Khi đó, Stochastic Hill Climbing sẽ hoạt động hiệu quả hơn, vì nó chỉ cần thử ngẫu nhiên vài lựa chọn là có thể tìm được hướng đi lên.

Nguyên lý hoạt động:

1. Tại mỗi bước, sinh ra một số trạng thái lân cận tốt hơn.
2. Chọn ngẫu nhiên một trong số đó, không cần phải là tốt nhất.
3. Di chuyển tới trạng thái đó, lặp lại quá trình.
4. Kết thúc khi không còn trạng thái tốt hơn nào có thể chọn.

Ưu điểm:

- ✓ Giảm đáng kể khả năng bị kẹt ở cực trị địa phương hoặc cao nguyên.
- ✓ Hiệu quả trong không gian trạng thái lớn và phức tạp.
- ✓ Không cần duyệt toàn bộ successor → tiết kiệm chi phí tính toán.

Nhược điểm:

- ✓ Kết quả không ổn định do phụ thuộc vào yếu tố ngẫu nhiên.
- ✓ Có thể chậm hơn hoặc cần lặp lại nhiều lần để tìm được lời giải tốt.

Stochastic Hill Climbing thường được kết hợp với các chiến lược khác như First-Choice Hill Climbing hoặc Random-Restart, giúp tăng tính đa dạng trong hành vi tìm kiếm và tăng xác suất tiếp cận được lời giải tối ưu toàn cục.

So sánh tổng thể:

Đặc điểm	Simple HC	Steepest-Ascent HC	Stochastic HC
Cách chọn bước tiếp theo	Gặp tốt thì đi	Chọn tốt nhất	Chọn ngẫu nhiên trong số tốt hơn
Nguy cơ kẹt local max	Cao	Trung bình	Thấp hơn
Tính ổn định kết quả	Cao	Cao	Thấp (phụ thuộc random)
Tốc độ	Nhanh	Chậm hơn	Trung bình

2.3.3. Ưu điểm và hạn chế của Hill Climbing

Hill Climbing là một trong những chiến lược tìm kiếm cục bộ phổ biến và dễ áp dụng nhất trong trí tuệ nhân tạo. Nhờ vào tính chất đơn giản trong thiết kế và tiết kiệm tài nguyên khi triển khai, thuật toán này thường được lựa chọn cho những bài toán tối ưu có không gian trạng thái lớn, nơi việc tìm kiếm theo kiểu truyền thống (như BFS hay A*) trở nên không khả thi vì chi phí bộ nhớ quá cao.

Ưu điểm:

- ✓ **Đơn giản và dễ cài đặt:** Hill Climbing có cấu trúc thuật toán ngắn gọn, dễ hiểu và dễ triển khai trong hầu hết các ngôn ngữ lập trình. Chỉ cần một hàm đánh giá (evaluation function) và một bộ sinh lân cận (neighbor generator) là có thể bắt đầu chạy.
- ✓ **Tiết kiệm bộ nhớ:** Thuật toán chỉ giữ một trạng thái tại một thời điểm, không cần lưu lại toàn bộ cây tìm kiếm hoặc danh sách các trạng thái đã xét như trong BFS hoặc A*. Điều này giúp giảm áp lực về tài nguyên, đặc biệt khi không gian trạng thái lớn.
- ✓ **Tốc độ thực thi cao trong không gian nhỏ:** Trong các bài toán có ít cực trị địa phương hoặc khi gần lời giải, Hill Climbing có thể hội tụ rất nhanh. Nhất là với biến thể Simple hoặc Steepest-Ascent, nó có thể tìm lời giải tốt mà không cần duyệt quá nhiều.
- ✓ **Phù hợp với các môi trường thời gian thực hoặc bị giới hạn tài nguyên:** Với yêu cầu phản hồi nhanh và xử lý nhẹ, Hill Climbing là lựa chọn hiệu quả cho game AI, robot đơn giản, hoặc các hệ thống nhúng.

Hạn chế:

- ✓ **Dễ bị kẹt tại local maxima (cực trị địa phương):** Nếu thuật toán gặp một trạng thái tốt hơn lân cận nhưng không phải tối ưu toàn cục, nó có thể "dừng lại giữa chừng" mà không tiến xa hơn.
- ✓ **Không xử lý tốt cao nguyên (plateau):** Trong trường hợp không có sự thay đổi rõ rệt giữa các trạng thái lân cận (giá trị đánh giá bằng nhau), thuật toán có thể "đi lòng vòng" mà không tìm được hướng đi lên.
- ✓ **Không thể đi lùi để thoát bẫy:** Hill Climbing không chấp nhận các bước đi tạm thời xấu hơn để có thể tìm hướng tốt hơn về sau (khác với Simulated Annealing). Điều này khiến nó thiếu tính linh hoạt trong không gian có nhiều vùng lồi lõm.
- ✓ **Phụ thuộc hoàn toàn vào hàm đánh giá:** Nếu hàm đánh giá được thiết kế chưa chuẩn hoặc không phản ánh đúng chất lượng của trạng thái, Hill Climbing sẽ đưa ra quyết định sai lệch.

PHẦN 3. PHÂN TÍCH VÀ THIẾT KẾ GIẢI PHÁP

3.1. Tìm kiếm không có thông tin (Uninformed Search)

3.1.1. Breadth-First Search (BFS) - Tìm kiếm theo chiều rộng

3.1.1.1. Mô tả

- ✓ Thuật toán **Breadth-First Search (BFS) – Tìm kiếm theo chiều rộng** là một phương pháp tìm kiếm không có thông tin (Uninformed Search), hoạt động theo nguyên lý duyệt theo chiều rộng – mở rộng lần lượt từng lớp nút trước khi đi sâu hơn. Thuật toán này thường được sử dụng trong môi trường đảm bảo, không có đối kháng, nơi mọi hành động đều dẫn đến trạng thái xác định và không có yếu tố ngẫu nhiên.
- ✓ Trong project này, thuật toán được áp dụng cho *Level 1* để giúp Pacman tìm được đường ngắn nhất (về số bước) đến viên thức ăn gần nhất, đồng thời đảm bảo hiệu quả trong môi trường đơn giản, không có sự can thiệp của quái vật.
- ✓ Ý tưởng chính của thuật toán là sử dụng một hàng đợi (queue) để duyệt lần lượt các ô theo chiều rộng, lưu lại đường đi qua ma trận *trace,* và chọn food gần nhất theo khoảng cách Manhattan.

3.1.1.2. Ý tưởng thuật toán

Hàm a: find_nearest_food

Input:

- ✓ _food_Position: danh sách các vị trí thức ăn trên bản đồ.
- ✓ start_row, start_col: vị trí hiện tại của Pacman.

Output: [food_row, food_col, _id]: tọa độ của viên thức ăn gần nhất và chỉ số của nó trong danh sách.

Mô tả cách hoạt động:

- ✓ Duyệt toàn bộ danh sách _food_Position.
- ✓ Tính khoảng cách Manhattan từ vị trí Pacman đến từng viên thức ăn.
- ✓ So sánh để tìm ra viên food có khoảng cách ngắn nhất.
- ✓ Trả về tọa độ và chỉ số của viên food gần nhất.
- ✓ Hàm này đóng vai trò xác định mục tiêu cụ thể để thuật toán BFS hướng tới.

Hàm b: BFS

Input:

- ✓ _map: ma trận bản đồ hiện tại.
- ✓ _food_Position: danh sách các vị trí của thức ăn.
- ✓ start_row, start_col: vị trí hiện tại của Pacman.
- ✓ N, M: kích thước bản đồ.

Output:

- ✓ Danh sách các tọa độ từ vị trí Pacman đến thức ăn gần nhất.

- ✓ Trả về [] nếu không còn food nào.

Mô tả cách hoạt động:

- ✓ Khởi tạo ma trận visited[N][M] để đánh dấu các ô đã được duyệt.
- ✓ Khởi tạo ma trận trace[N][M] để lưu vết đường đi (vị trí cha của mỗi ô).
- ✓ Gọi find_nearest_food để xác định vị trí food gần nhất từ vị trí Pacman.
- ✓ Nếu không còn food (_id == -1) thì trả về danh sách rỗng.
- ✓ Tạo hàng đợi lt chứa vị trí bắt đầu, đánh dấu vị trí đó là đã duyệt.
- ✓ Bắt đầu vòng lặp BFS:
  - Lấy phần tử đầu tiên ra khỏi hàng đợi.
  - Nếu vị trí đó là food cần tìm → kết thúc vòng lặp.
  - Duyệt 4 hướng xung quanh:
    - Nếu ô mới hợp lệ và chưa duyệt:
    - Thêm vào hàng đợi.
    - Đánh dấu visited.
    - Ghi lại vị trí cha trong trace.
  - Nếu không tìm được đường đi đến food:
    - Xóa viên food đó khỏi _food_Position.
    - Gọi lại BFS để tìm food khác.
  - Nếu tìm thấy food:
    - Dùng trace để truy ngược từ food về vị trí ban đầu.
    - Tạo danh sách result chứa các bước di chuyển hợp lệ theo thứ tự.

Nhận xét về thuật toán

Ưu điểm:

- ✓ Luôn tìm được đường đi ngắn nhất về số bước đến food gần nhất.
- ✓ Hoạt động tốt trong bản đồ nhỏ hoặc có ít chướng ngại vật.
- ✓ Cài đặt đơn giản, dễ kiểm soát và debug.

Nhược điểm:

- ✓ Không tận dụng được thông tin hướng dẫn như khoảng cách đến food (không dùng heuristic).
- ✓ Tốn bộ nhớ do phải lưu visited, trace, và hàng đợi toàn cục.
- ✓ Nếu có nhiều food trên bản đồ, thuật toán chỉ xử lý một food tại một thời điểm, không tối ưu toàn cục.

3.1.1.3. Mã giả

```
Hàm BFS(map, food_Position, start_row, start_col, N, M):
    Tạo ma trận visited[N][M], gán tất cả giá trị = False
    Tạo ma trận trace[N][M], mỗi phần tử là [-1, -1]

    [food_row, food_col] ← tìm thức ăn gần nhất từ food_Position và vị trí bắt đầu

    Khởi tạo queue ← danh sách rỗng
    Đặt biến found ← False

    Đánh dấu visited[start_row][start_col] = True
    Thêm [start_row, start_col] vào queue

    Trong khi queue không rỗng:
        Lấy [row, col] ← phần tử đầu tiên của queue
        Xóa phần tử này khỏi queue

        Nếu [row, col] == [food_row, food_col]:
            found ← True
            Thoát khỏi vòng lặp

        Với mỗi hướng di chuyển [d_r, d_c] trong 4 hướng:
            new_row ← row + d_r
            new_col ← col + d_c

            Nếu ô (new_row, new_col) hợp lệ và chưa được duyệt:
                Đánh dấu visited[new_row][new_col] = True
                Thêm [new_row, new_col] vào queue
                trace[new_row][new_col] ← [row, col]

    Nếu found == False:
        Xóa viên thức ăn khỏi danh sách food_Position
        Gọi lại BFS(map, food_Position, start_row, start_col, N, M)

    Ngược lại (đã tìm được food):
        Tạo danh sách result ← [[food_row, food_col]]
        [row, col] ← trace[food_row][food_col]

        Trong khi row ≠ -1:
            Chèn [row, col] vào đầu danh sách result
            [row, col] ← trace[row][col]

        Trả về result
```

Hình 1: Mã giả thuật toán BFS

3.1.2. Depth-First Search (DFS) - Tm kiếm theo chiều sâu

3.1.2.1. Mô tả

- ✓ Thuật toán **Depth-First Search (DFS) – Tìm kiếm theo chiều sâu** là một phương pháp tìm kiếm không có thông tin (uninformed search), hoạt động theo chiến lược đi càng sâu càng tốt trước khi quay lui để thử nhánh khác.
  DFS thường được sử dụng trong môi trường không xác định độ sâu, không có đối thủ (ghost), và không cần đánh giá chi phí đường đi.
- ✓ Trong project này, thuật toán DFS được áp dụng cho Level 1 để giúp Pacman tìm đường đi đến viên thức ăn đầu tiên mà nó bắt gặp, theo chiều sâu của bản đồ.
- ✓ Ý tưởng chính của thuật toán là sử dụng đệ quy để mở rộng đường đi theo chiều sâu. Khi đến được food, thuật toán dừng lại và trả về đường đi đã tìm thấy.

3.1.2.2. Ý tưởng thuật toán

Hàm a: Deque_DFS

Input:

- ✓ _map: bản đồ hiện tại dưới dạng ma trận 2D.
- ✓ _food_Position: danh sách các vị trí thức ăn.

- ✓ row, col: vị trí hiện tại của Pacman.
- ✓ N, M: kích thước bản đồ.
- ✓ visited: ma trận đánh dấu đã duyệt.
- ✓ trace: danh sách lưu đường đi hiện tại.

Output: Trả về 1 nếu tìm thấy food, 0 nếu không tìm thấy.

Mô tả cách hoạt động:

- ✓ Nếu ô hiện tại đã được duyệt → return 0.
- ✓ Đánh dấu vị trí hiện tại là đã duyệt, thêm vào trace.
- ✓ Nếu ô hiện tại là food → return 1 (tìm thấy food).
- ✓ Duyệt 4 hướng xung quanh:
  - Nếu ô mới hợp lệ và chưa được duyệt:
    - Gọi đệ quy đến ô mới.
    - Nếu tìm được food trong nhánh đó → return 1.
    - Nếu không tìm thấy food → loại bỏ ô khỏi trace (quay lui).
- ✓ Nếu duyệt hết các nhánh mà không đến được food → return 0.

Hàm b: DFS

Input:

- ✓ _map: bản đồ hiện tại.
- ✓ _food_Position: danh sách thức ăn.
- ✓ start_row, start_col: vị trí hiện tại của Pacman.
- ✓ N, M: kích thước bản đồ.

Output:

- ✓ Trả về danh sách trace – đường đi đến food nếu tìm thấy.
- ✓ Nếu không có đường đi, trả về [].

Mô tả cách hoạt động:

- ✓ Khởi tạo ma trận visited[N][M] với giá trị False.
- ✓ Khởi tạo danh sách trace rỗng.
- ✓ Gọi Deque_DFS(...) để bắt đầu tìm đường từ vị trí Pacman.
- ✓ Nếu kết quả trả về là 1 → đã tìm thấy đường đi → trả về trace.
- ✓ Ngược lại → không tìm thấy đường → trả về danh sách rỗng.

Nhận xét về thuật toán

Ưu điểm:

- ✓ Cài đặt đơn giản, sử dụng đệ quy dễ hiểu.
- ✓ Ít tốn bộ nhớ hơn BFS vì không cần lưu toàn bộ mức độ các nút (không cần queue).
- ✓ Phù hợp với bản đồ nhỏ hoặc cần kiểm tra khả năng tồn tại đường đi.

Nhược điểm:

- ✓ Không đảm bảo tìm đường ngắn nhất, vì đi sâu có thể dẫn đến đường vòng.
- ✓ Có thể rơi vào vòng lặp hoặc đi sai nhánh nếu không xử lý tốt visited.
- ✓ Với bản đồ lớn, nguy cơ bị **stack overflow** do đệ quy sâu.

3.1.2.3. Mã giả

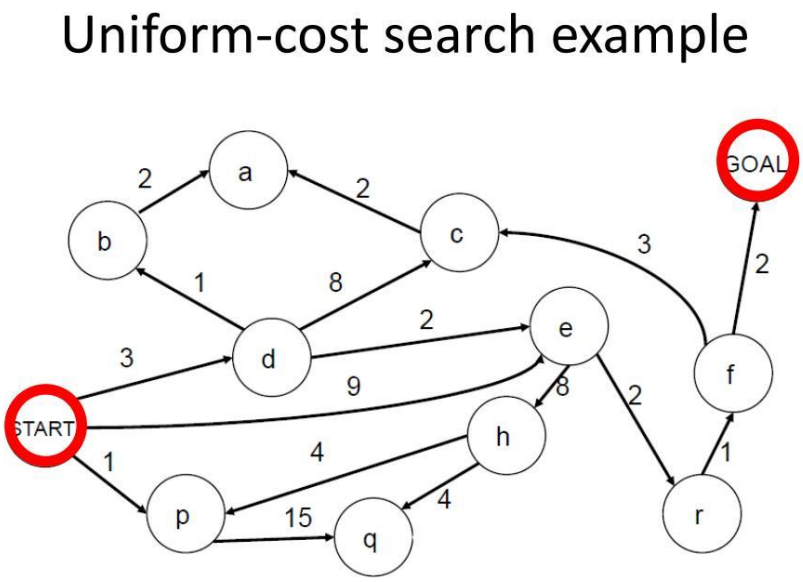
```
Thuật toán DFS(map, food_Position, start_row, start_col, N, M):

    Khởi tạo visited[N][M] ← False
    Khởi tạo trace ← danh sách rỗng

    Gọi hàm Deque_DFS(map, food_Position, start_row, start_col, N, M, visited, trace)

    Nếu kết quả trả về là 1 (đã tìm thấy FOOD):
        Trả về trace (đường đi từ vị trí bắt đầu đến FOOD)
    Ngược lại:
        Trả về danh sách rỗng
Hàm Deque_DFS(map, food_Position, row, col, N, M, visited, trace):

    Nếu ô (row, col) đã được thăm → trả về 0

    Đánh dấu visited[row][col] ← True
    Thêm (row, col) vào trace

    Nếu ô hiện tại là FOOD → trả về 1 (đã tìm thấy mục tiêu)

    Với mỗi hướng di chuyển (dr, dc) trong DDX:
        new_row ← row + dr
        new_col ← col + dc

        Nếu vị trí (new_row, new_col) hợp lệ và chưa thăm:
            res ← Deque_DFS(...) đệ quy
            Nếu res == 1 → trả về 1
            Ngược lại: xóa bước cuối cùng khỏi trace
    Trả về 0 (không tìm thấy FOOD trong nhánh này)
```

Hình 2: Mã giả thuật toán DFS

3.1.3 Uniform Cost Search (UCS) - Tìm kiếm theo chi phí thống nhất

3.1.3.1. Mô tả

- ✓ Thuật toán **Uniform Cost Search (UCS) - Tìm kiếm theo chi phí thống nhất** là một phương pháp tìm kiếm không có thông tin trước (uninformed search), hoạt động dựa trên nguyên tắc mở rộng nút có chi phí thấp nhất trước. UCS tương tự như thuật toán Dijkstra, nhưng được áp dụng trong các bài toán tìm kiếm tổng quát.
- ✓ Trong project này, UCS được sử dụng ở Level 1 để giúp Pacman tìm đường đi với tổng chi phí thấp nhất đến viên thức ăn gần nhất, đặc biệt hữu ích khi các bước di chuyển có chi phí khác nhau (ví dụ: di chuyển qua các loại địa hình khác nhau).
- ✓ Ý tưởng chính của UCS là sử dụng một hàng đợi ưu tiên (priority queue) để luôn chọn mở rộng nút có tổng chi phí từ điểm bắt đầu đến nó là nhỏ nhất. Quá trình tiếp tục cho đến khi tìm được đường đi đến mục tiêu với chi phí tối ưu.



Hình 3: Ví dụ về thuật toán UCS

3.1.3.2. Ý tưởng thuật toán

Hàm a: find_nearest_food

Input:

- ✓ _food_Position: danh sách tọa độ các viên thức ăn trên bản đồ.
- ✓ start_row, start_col: vị trí hiện tại của Pacman.

Output: [food_row, food_col, _id]: tọa độ của viên thức ăn gần nhất và chỉ số của nó trong danh sách.

Mô tả cách hoạt động:

- ✓ Duyệt qua toàn bộ danh sách thức ăn.
- ✓ Tính khoảng cách Manhattan từ vị trí hiện tại của Pacman đến từng viên thức ăn.
- ✓ Chọn viên thức ăn có khoảng cách nhỏ nhất và trả về tọa độ cùng chỉ số của nó.

Hàm b: UCS

Input:

- ✓ _map: ma trận bản đồ hiện tại.
- ✓ _food_Position: danh sách các vị trí của thức ăn.
- ✓ start_row, start_col: vị trí hiện tại của Pacman.
- ✓ N, M: kích thước bản đồ.

Output:

- ✓ Danh sách các tọa độ từ vị trí hiện tại đến thức ăn gần nhất với tổng chi phí thấp nhất.
- ✓ Trả về [] nếu không còn thức ăn nào.

Mô tả cách hoạt động:

- ✓ Khởi tạo ma trận visited[N][M] để đánh dấu các ô đã được duyệt.
- ✓ Khởi tạo ma trận trace[N][M] để lưu vết đường đi.
- ✓ Khởi tạo ma trận cost[N][M] để lưu tổng chi phí từ điểm bắt đầu đến mỗi ô, gán tất cả giá trị ban đầu là vô cùng lớn.
- ✓ Gọi find_nearest_food để xác định viên thức ăn gần nhất.
- ✓ Nếu không còn thức ăn nào (_id == -1), trả về danh sách rỗng.
- ✓ Khởi tạo hàng đợi ưu tiên pq và thêm vào đó vị trí bắt đầu với chi phí 0.
- ✓ Đặt cost[start_row][start_col] = 0.
- ✓ Trong khi hàng đợi không rỗng:
  - Lấy phần tử có chi phí thấp nhất ra khỏi hàng đợi.
  - Nếu vị trí hiện tại là viên thức ăn cần tìm, dừng vòng lặp.
  - Duyệt qua 4 hướng di chuyển từ vị trí hiện tại:
    - Nếu ô mới hợp lệ và chưa được duyệt:
    - Tính tổng chi phí mới = chi phí hiện tại + chi phí di chuyển đến ô mới.
    - Nếu tổng chi phí mới nhỏ hơn cost[new_row][new_col]:
    - Cập nhật cost[new_row][new_col].
    - Cập nhật trace[new_row][new_col] là vị trí hiện tại.
    - Thêm ô mới vào hàng đợi với tổng chi phí mới.
  - Nếu không tìm được đường đi đến viên thức ăn:
    - Xóa viên thức ăn đó khỏi danh sách _food_Position.
    - Gọi lại hàm UCS để tìm viên thức ăn gần nhất còn lại.
  - Nếu tìm thấy đường đi:
    - Duyệt ngược từ vị trí viên thức ăn về vị trí bắt đầu thông qua trace để tái tạo đường đi.
    - Trả về danh sách các tọa độ theo thứ tự từ vị trí bắt đầu đến viên thức ăn.

Nhận xét về thuật toán

Ưu điểm:

- ✓ Đảm bảo tìm được đường đi với tổng chi phí thấp nhất, phù hợp với các bài toán có chi phí di chuyển khác nhau.
- ✓ Hoàn chỉnh: luôn tìm được giải pháp nếu tồn tại.

Nhược điểm:

- ✓ Tốn bộ nhớ và thời gian hơn so với BFS, đặc biệt trong bản đồ lớn hoặc có nhiều chướng ngại vật.
- ✓ Không sử dụng heuristic, nên có thể mở rộng nhiều nút không cần thiết.

3.1.3.3. Mã giả

```
Thuật toán UCS(map, food_pos, row, col, N, M):
    Khởi tạo:
        visited[N][M] ← False
        trace ← ánh xạ lưu đường đi
        cost ← ánh xạ chi phí từ điểm bắt đầu
        from_pos ← (row, col)
        to_pos ← (tọa độ viên FOOD gần nhất)
    Nếu from_pos == to_pos → trả về []
    Tìm tất cả vị trí MONSTER và lưu vào ghost_pos
    Khởi tạo PriorityQueue q
    cost[from_pos] ← 0
    Thêm vào q: (0, from_pos)
    Trong khi q chưa rỗng:
        Lấy current ← node có chi phí nhỏ nhất trong q
        Nếu current đã thăm → bỏ qua
        Đánh dấu current đã thăm
        Nếu current == to_pos:
            Truy vết từ current về from_pos thông qua trace
            Trả về path
        Với mỗi hướng (dr, dc) trong DDX:
            new_r ← r + dr
            new_c ← c + dc
            Nếu vị trí mới không hợp lệ → bỏ qua
            Kiểm tra nếu (new_r, new_c) cách MONSTER ≤ 1 → bỏ qua vì nguy hiểm
            Tính new_cost ← cost[current] + 1
            Nếu next_pos chưa có trong cost hoặc new_cost < cost[next_pos]:
                Cập nhật cost[next_pos] ← new_cost
                trace[next_pos] ← current
                Thêm vào q: (new_cost, next_pos)
    Nếu không tìm được đường → trả về []
```

Hình 4: Mã giả thuật toán UCS

3.2. Tìm kiếm có thông tin (Informed Search / Heuristic Search)

3.2.1. Greedy Search - Tìm kiếm tham lam

3.2.1.1. Mô tả

- ✓ Thuật toán Greedy (Tìm kiếm tham lam) là một phương pháp tìm kiếm có thông tin, thường được sử dụng trong môi trường xác định, có thể ước lượng khoảng cách đến mục tiêu bằng heuristic mà cụ thể là dựa trên khoảng cách Manhattan.
- ✓ Thuật toán tìm kiếm tham lam – Greedy này được áp dụng cho Level 2 để giúp Pacman tìm đường đến thức ăn gần nhất một cách nhanh chóng.
- ✓ Ý tưởng chính của thuật toán là tại mỗi bước, Pacman luôn chọn đi đến ô có khoảng cách gần nhất với mục tiêu (FOOD), mà không xét toàn cục hay quay lại đường cũ.

3.2.1.2. Ý tưởng thuật toán

Hàm: Greedy (_map, food_pos, row, col, N, M)

Input:

- ✓ _map: ma trận bản đồ của trò chơi.

- ✓ food_pos: danh sách vị trí các ô chứa FOOD (thức ăn).
- ✓ row, col: vị trí hiện tại của Pacman.
- ✓ N, M: kích thước của bản đồ.

Output:

- ✓ Một danh sách các ô tạo thành đường đi từ Pacman đến FOOD gần nhất.
- ✓ Nếu không tìm được đường đi, trả về danh sách rỗng [].

Mô tả cách hoạt động:

1. Tìm vị trí FOOD gần nhất bằng cách duyệt qua tất cả các food và chọn vị trí có khoảng cách Manhattan nhỏ nhất.
2. Khởi tạo một hàng đợi ưu tiên (PriorityQueue), trong đó ưu tiên các trạng thái gần FOOD hơn.
3. Dùng biến visited để đánh dấu các ô đã đi và trace để lưu lại đường đi.
4. Lặp:
    - Lấy ô có heuristic thấp nhất từ hàng đợi ưu tiên.
    - Nếu ô hiện tại là FOOD thì truy ngược bằng trace để tạo path và trả kết quả.
    - Nếu không, mở rộng 4 hướng: nếu hợp lệ và chưa thăm thì tính khoảng cách heuristic rồi thêm vào hàng đợi.
5. Nếu duyệt hết mà không đến được FOOD thì trả về đường đi rỗng [].

Điểm nổi bật:

- ✓ Dùng heuristic (Manhattan) để định hướng tìm kiếm.
- ✓ Không quan tâm đến số bước đã đi (g(n)), chỉ dùng h(n) tức f(n) = h(n)
- ✓ Không đảm bảo đường đi tối ưu, nhưng rất nhanh trong môi trường đơn giản.

Nhận xét về thuật toán so với các thuật toán cùng nhóm

Ưu điểm:

- ✓ Cài đặt đơn giản, tốc độ nhanh.
- ✓ Tìm được kết quả tốt trong các bản đồ không quá phức tạp.
- ✓ Dễ mở rộng và kết hợp với các thuật toán khác.

Nhược điểm:

- ✓ Không đảm bảo tìm được đường đi ngắn nhất.
- ✓ Dễ bị kẹt nếu bản đồ rẽ nhánh phức tạp.
- ✓ Không xét đến chi phí đã đi (khác A*).

3.2.1.3. Mã giả

```
Hàm Greedy (_map, food_pos, row, col, N, M):
    Nếu food_pos rỗng:
        Trả về []
    // Tìm food gần nhất dựa trên khoảng cách Manhattan
    Đặt min_h ← +∞
    goal ← null
    Duyệt mỗi food trong food_pos:
        h ← khoảng cách Manhattan từ (row, col) đến food
        Nếu h < min_h:
            min_h ← h
            goal ← food
    Nếu goal là null:
        Trả về []
    // Khởi tạo các cấu trúc cần thiết
    visited ← ma trận N×M toàn False
    trace ← dictionary rỗng
    hàng_đợi ← PriorityQueue
    Thêm (Manhattan(row, col, goal), (row, col)) vào hàng_đợi
    Đánh dấu visited[row][col] ← True
    // Bắt đầu tìm kiếm
    Trong khi hàng_đợi không rỗng:
        Lấy ra (r, c) có giá trị ưu tiên nhỏ nhất từ hàng_đợi
        Nếu (r, c) == goal:
            path ← [(r, c)]
            Trong khi (r, c) khác (row, col):
                (r, c) ← trace[(r, c)]
                Thêm (r, c) vào đầu path
            Trả về path
        Cho mỗi hướng (d_r, d_c) trong DDX:
            new_r ← r + d_r
            new_c ← c + d_c
            Nếu isValid(_map, new_r, new_c, N, M) và chưa visited[new_r][new_c]:
                visited[new_r][new_c] ← True
                trace[(new_r, new_c)] ← (r, c)
                h ← Manhattan(new_r, new_c, goal)
                Thêm (h, (new_r, new_c)) vào hàng_đợi
    // Không tìm được đường
    Trả về []
```

Hình 5: Mã giả thuật toán Greedy

3.2.2. A* Search

3.2.2.1. Mô tả

- ✓ Thuật toán **A*** là một phương pháp **tìm kiếm có thông tin trước (informed search)**, kết hợp giữa chi phí thực tế đã đi (g(n)) và chi phí ước lượng còn lại đến đích (h(n), heuristic).

  Khác với UCS chỉ dựa vào chi phí thực tế, A* sử dụng công thức **f(n) = g(n) + h(n)** để mở rộng các nút có tiềm năng tốt nhất, tức là vừa gần điểm xuất phát vừa gần mục tiêu.
- ✓ Trong project này, **A*** được sử dụng ở **Level 3** để giúp Pacman **tìm đường đi ngắn nhất đến viên thức ăn gần nhất**, kết hợp giữa tính toán chi phí di chuyển và ước lượng khoảng cách còn lại.

  Heuristic được dùng là **khoảng cách Manhattan**, rất phù hợp trong môi trường lưới như bản đồ Pacman.

- ✓ **Ý tưởng chính** của A* là sử dụng một **hàng đợi ưu tiên (priority queue)** để luôn chọn mở rộng nút có tổng f(n) thấp nhất. Nhờ đó, thuật toán vừa hướng dẫn được đường đi thông minh hơn, vừa đảm bảo tối ưu hóa chi phí di chuyển.

3.2.2.2. Ý tưởng thuật toán

Hàm a: find_nearest_food

Input:

- ✓ _food_Position: danh sách các viên thức ăn trên bản đồ.
- ✓ start_row, start_col: vị trí hiện tại của Pacman.

Output: [food_row, food_col]: tọa độ viên thức ăn gần nhất (theo khoảng cách Manhattan).

Mô tả cách hoạt động:

- ✓ Duyệt toàn bộ danh sách thức ăn.
- ✓ Với mỗi viên food, tính khoảng cách Manhattan đến vị trí Pacman.
- ✓ Chọn viên có khoảng cách nhỏ nhất.
- ✓ Trả về tọa độ của viên food gần nhất để làm mục tiêu cho thuật toán A*.

Hàm b: AStar

Input:

- ✓ _map: bản đồ hiện tại dạng ma trận 2D.
- ✓ _food_Position: danh sách vị trí các viên thức ăn.
- ✓ start_row, start_col: vị trí hiện tại của Pacman.
- ✓ N, M: kích thước bản đồ.

Output:

- ✓ Danh sách path chứa các tọa độ từ Pacman đến food gần nhất theo đường đi tối ưu.
- ✓ Trả về danh sách rỗng nếu không tìm được đường đi.

Mô tả cách hoạt động:

Khởi tạo:

- ✓ visited: ma trận đánh dấu ô đã duyệt.
- ✓ trace: dictionary lưu vết đường đi (để truy ngược từ goal về start).
- ✓ cost: lưu tổng chi phí từ điểm bắt đầu đến mỗi ô (g(n)).
- ✓ queue: hàng đợi ưu tiên, ưu tiên theo f(n) = g(n) + h(n).

Xác định mục tiêu:

- ✓ Gọi find_nearest_food để lấy tọa độ viên thức ăn gần nhất (end).

Bắt đầu từ điểm start:

- ✓ Đặt cost[start] = 0 và thêm vào queue với f = h(start).

Vòng lặp tìm kiếm:

- ✓ Trong khi hàng đợi chưa rỗng:

- Lấy node có f(n) nhỏ nhất ra.
- Đánh dấu là đã duyệt.
- Nếu node đó là food → bắt đầu truy vết ngược bằng trace.

✓ Nếu chưa tới goal, duyệt 4 hướng (trên, dưới, trái, phải):

- Nếu ô hợp lệ và chưa duyệt:
- Tính g(n) mới = cost[current] + 1
- Tính h(n) = khoảng cách Manhattan đến food
- Tính f(n) = g(n) + h(n)
- Cập nhật trace, cost, và đưa node mới vào hàng đợi

Kết thúc:

✓ Nếu đến được food, dùng trace để truy ngược đường đi → trả về path.

✓ Nếu không đến được → trả về danh sách rỗng.

Nhận xét về thuật toán

Ưu điểm:

✓ **Tối ưu**: Đảm bảo tìm được đường đi ngắn nhất nếu hàm heuristic là admissible.

✓ **Hiệu quả**: Sử dụng thông tin heuristic để hướng dẫn tìm kiếm, giảm số lượng nút cần mở rộng.

✓ **Linh hoạt**: Có thể điều chỉnh hàm heuristic để phù hợp với từng bài toán cụ thể.

Nhược điểm:

✓ **Tốn bộ nhớ**: Cần lưu trữ nhiều thông tin như g_cost, trace, và hàng đợi ưu tiên.

✓ **Phụ thuộc vào heuristic**: Nếu hàm heuristic không tốt, hiệu quả của thuật toán giảm.

✓ **Không phù hợp với không gian tìm kiếm rất lớn**: Trong trường hợp không gian tìm kiếm quá lớn, A* có thể không khả thi do yêu cầu bộ nhớ và thời gian.

3.2.2.3. Mã giả

```
Thuật toán AStar(map, food_Position, start_row, start_col, N, M):
    Khởi tạo:
        visited[N][M] ← False
        trace ← ánh xạ để lưu đường đi
        cost ← lưu chi phí từ điểm bắt đầu đến mỗi ô
        path ← danh sách rỗng
        queue ← PriorityQueue sắp theo f(n) = g(n) + h(n)
    Gọi hàm find_nearest_food(...) → trả về (food_row, food_col)
    start ← (start_row, start_col)
    end ← (food_row, food_col)
    cost[start] ← 0
    Thêm vào queue: (f(start), start), trong đó f = g + h = 0 + heuristic(start, end)
    Trong khi queue không rỗng:
        current ← node có f nhỏ nhất trong queue
        Đánh dấu current là đã thăm
        Nếu current == end:
            Truy vết đường đi từ end → start qua trace
            Đảo ngược đường đi và trả về path
        Với mỗi hướng di chuyển (dr, dc) trong DDX:
            new_row ← current_row + dr
            new_col ← current_col + dc
            Nếu (new_row, new_col) hợp lệ và chưa thăm:
                g ← cost[current] + 1
                h ← Manhattan(new_row, new_col, food_row, food_col)
                f ← g + h
                Cập nhật cost[(new_row, new_col)] ← g
                Thêm vào queue: (f, (new_row, new_col))
                Cập nhật trace[(new_row, new_col)] ← current
    Nếu không tìm thấy đường đi → trả về path rỗng
```

Hình 6: Mã giả thuật toán A Star

3.3. Tìm kiếm cục bộ (Local Search)

3.3.1. Steepest-Ascent Hill Climbing - Leo đồi

3.3.1.1. Mô tả

Steepest-Ascent Hill Climbing là một thuật toán tìm kiếm cục bộ (local search) dựa trên nguyên tắc leo đồi, trong đó Pacman luôn chọn bước đi tiếp theo có **giá trị heuristic cao nhất** trong số các lựa chọn lân cận. Thuật toán chỉ quan tâm đến vùng lân cận hiện tại (không nhớ trạng thái toàn cục), phù hợp với môi trường có **tầm nhìn hạn chế** như trong Level 3 của trò chơi.

Trong thuật toán này, mỗi ô trên bản đồ được gán một điểm số heuristic (cost) tùy theo:

- ✓ Mức độ gần thức ăn (càng gần → điểm càng cao)
- ✓ Mức độ gần Monster (càng gần → điểm càng thấp, hoặc âm vô cùng nếu quá gần)

Pacman sau đó chọn ô kề cận có giá trị **f(n) = h(n) - v(n)** lớn nhất, trong đó:

- ✓ h(n): điểm heuristic tại ô đó
- ✓ v(n): số lần đã đi qua ô đó (được lưu trong _visited) — để tránh quay lại nhiều lần

3.3.1.2. Ý tưởng thuật toán

- **Hàm update_heuristic**
  - ✓ **Input:**

    _map, start_row, start_col: vị trí ban đầu của Pacman
    current_row, current_col: ô đang xét hiện tại
    N, M: kích thước bản đồ
    depth: độ sâu tìm kiếm (giới hạn trong 3 lớp lân cận)
    visited: danh sách ô đã duyệt
    _type: loại đối tượng đang xử lý (FOOD hoặc MONSTER)
    cost: ma trận điểm heuristic cần cập nhật
  - ✓ **Output:** Không trả về giá trị, nhưng cập nhật trực tiếp ma trận cost.
  - ✓ **Mô tả hoạt động:** Đây là hàm lan truyền ảnh hưởng heuristic từ các ô có thức ăn hoặc Monster:
    - Nếu gặp thức ăn, tăng điểm cost tại các ô lân cận theo mức giảm dần:

      Depth 2 → +35
      Depth 1 → +10
      Depth 0 → +5
    - Nếu gặp Monster, làm giảm mạnh giá trị cost:

      Depth 2 hoặc 1 → -∞ (không nên đi)
      Depth 0 → -100
    - Sau khi gán điểm tại ô hiện tại, hàm tiếp tục gọi đệ quy để duyệt các ô lân cận ở độ sâu nhỏ hơn. Tránh đi lại ô cũ bằng visited.
- **Hàm calc_heuristic**
  - ✓ **Input:**

    _map, start_row, start_col: vị trí bắt đầu của Pacman
    current_row, current_col: ô đang xét hiện tại
    N, M: kích thước bản đồ
    depth: giới hạn độ sâu tìm kiếm (ban đầu là 3)
    visited: tập các ô đã xét
    cost: ma trận điểm cần cập nhật
    _visited: ma trận số lần Pacman đã đi qua mỗi ô
  - ✓ **Output:** Không trả về giá trị, chỉ cập nhật cost.
  - ✓ **Mô tả hoạt động:** Hàm này dùng đệ quy DLS (Depth-Limited Search) để duyệt toàn bộ phạm vi tầm nhìn của Pacman (3 bước).
    - Với mỗi ô kề cận:
    - Nếu gặp thức ăn: gọi update_heuristic để lan truyền điểm cộng.
    - Nếu gặp Monster: gọi update_heuristic để lan truyền điểm trừ.

- ❖ Sau mỗi bước, cost tại ô đang xét bị giảm đi giá trị tương ứng với số lần đã đi qua ô đó (trừ _visited[r][c]), tránh lặp lại.

❖ **Hàm SA_HillClimbing**

- ✓ **Input:**

  _map: bản đồ hiện tại

  food_pos: danh sách vị trí thức ăn

  start_row, start_col: vị trí hiện tại của Pacman

  N, M: kích thước bản đồ

  _visited: ma trận số lần Pacman đã đi qua từng ô
- ✓ **Output:** Vị trí mới mà Pacman nên di chuyển tới [new_row, new_col]
- ✓ **Mô tả hoạt động:** Gọi calc_heuristic để tính cost cho toàn bộ vùng Pacman có thể quan sát.
  - ❖ Sau đó duyệt qua 4 ô kề cận:
  - ❖ Nếu hợp lệ (không phải tường), tính điểm f(n) = cost[r][c] - _visited[r][c]
  - ❖ Lưu lại ô có điểm lớn nhất → trả về ô đó làm bước đi tiếp theo.

Nhận xét

Ưu điểm:

- ❖ Đơn giản, nhanh, phù hợp với môi trường có tầm nhìn giới hạn như Level 3.
- ❖ Luôn chọn hướng đi tốt nhất tại thời điểm hiện tại.
- ❖ Heuristic được thiết kế hợp lý giúp Pacman ưu tiên thức ăn và tránh Monster.

Nhược điểm:

- ❖ Dễ rơi vào điểm cực trị cục bộ (local maximum): Pacman có thể chọn ô tốt nhất trước mắt nhưng bỏ lỡ đường đi tối ưu hơn ở xa.
- ❖ Không xét lại trạng thái trước đó → có thể bị mắc kẹt nếu không còn hướng đi tốt hơn.
- ❖ Không có backtracking nên nếu bị chặn hoặc lặp lại, Pacman có thể đi vòng hoặc đứng yên nếu không có lựa chọn tốt hơn.

3.3.1.3. Mã giả

```
Hàm SA_HillClimbing(map, food_pos, start_row, start_col, N, M, _visited):
    Khởi tạo visited là rỗng
    Khởi tạo cost là ma trận N x M với giá trị 0

    Gọi calc_heuristic(map, start_row, start_col, start_row, start_col, N, M, depth=3, visited, cost, _visited)

    max_f = -∞
    result = []

    Cho mỗi hướng [d_r, d_c] trong DDX:
        new_row = start_row + d_r
        new_col = start_col + d_c
        Nếu hợp lệ và không phải tường:
            score = cost[new_row][new_col] - _visited[new_row][new_col]
            Nếu score > max_f:
                max_f = score
                result = [new_row, new_col]

    Trả về result
```

Hình 7: Mã giả thuật toán Steepest-Ascent Hill Climbing

3.3.2. Simulated Annealing - Ủ mô phỏng

3.3.2.1. Mô tả

- ✓ Thuật toán Simulated Annealing for PacMan là một phương pháp tìm kiếm cục bộ và được tích hợp thêm khả năng né quái dựa vào cách tính toán đường đi tối ưu, thường được sử dụng trong các môi trường có nhiều cạm bẫy, trạng thái không hoàn hảo và yêu cầu né tránh vật cản (quái vật, tường).
- ✓ Thuật toán được áp dụng cho Level 3 để giúp Pacman vừa tìm được đường đi hiệu quả, vừa tránh quái (MONSTER), ưu tiên đến gần thức ăn (FOOD) và thoát khỏi trạng thái mắc kẹt.
- ✓ Ý tưởng chính của thuật toán là: Sử dụng bản đồ heuristic đánh giá mức độ nguy hiểm và lợi ích xung quanh, sau đó áp dụng Simulated Annealing để chọn bước đi tốt, hoặc đôi khi chấp nhận bước đi xấu với xác suất giảm dần theo nhiệt độ T.

3.3.2.2. Ý tưởng thuật toán

Hàm a: update_heuristic (_map, current_row, current_col, N, M, depth, visited, _type, heuristic_map)

Input:

- ✓ _map: bản đồ game hiện tại
- ✓ current_row, current_col: vị trí của một FOOD hoặc MONSTER
- ✓ N, M: kích thước bản đồ
- ✓ depth: độ sâu lan tỏa điểm
- ✓ visited: danh sách ô đã duyệt
- ✓ _type: loại đối tượng (FOOD hoặc MONSTER)
- ✓ heuristic_map: bản đồ điểm heuristic đang cập nhật

Output: Cập nhật trực tiếp vào heuristic_map để đánh giá mức độ thưởng/phạt của bản đồ

Mô tả cách hoạt động:

Hàm sử dụng đệ quy để lan tỏa ảnh hưởng của FOOD hoặc MONSTER ra xung quanh trong bán kính depth.

- ✓ Nếu là FOOD, càng gần càng cộng điểm cao.
- ✓ Nếu là MONSTER, càng gần càng trừ điểm mạnh (thậm chí vô cực âm).
- ✓ Dừng lan khi depth < 0 hoặc ô đã duyệt.
- ✓ Gọi tiếp các ô hàng xóm với depth – 1.

Từ đó tạo ra được bản đồ đánh giá nguy hiểm/an toàn (heuristic_map).

Hàm b: calc_heuristic(_map, N, M)

Input:

- ✓ _map: bản đồ PacMan hiện tại
- ✓ N, M: kích thước bản đồ

Output: heuristic_map: ma trận N×M với điểm heuristic của từng ô trong bản đồ

Mô tả cách hoạt động:

Hàm khởi tạo một bản đồ điểm heuristic rỗng và duyệt qua toàn bộ bản đồ:

- ✓ Gặp FOOD thì gọi update_heuristic(..., FOOD)
- ✓ Gặp MONSTER thì gọi update_heuristic(..., MONSTER)

Hàm c: SimulatedAnnealingForPacMan(_map, start_row, start_col, N, M, _visited, heuristic_map, prev_pos=None, T=1000, stuck_counter=0)

Input:

- ✓ _map: bản đồ PacMan
- ✓ start_row, start_col: vị trí PacMan hiện tại
- ✓ N, M: kích thước bản đồ
- ✓ _visited: ma trận số lần Pacman đã đi qua các ô
- ✓ heuristic_map: bản đồ điểm heuristic được tính từ calc_heuristic
- ✓ prev_pos: ô trước đó vừa đi (nếu có)
- ✓ T: nhiệt độ hiện tại (mặc định 1000)
- ✓ stuck_counter: bộ đếm số lần bị mắc kẹt (mặc định 0)

Output:

- ✓ [move]: bước đi mới của PacMan
- ✓ move: vị trí vừa chọn (tuple)
- ✓ T: nhiệt độ mới sau khi di chuyển
- ✓ stuck_counter: bộ đếm cập nhật

Mô tả cách hoạt động:

1. Tạo danh sách neighbors gồm các ô hợp lệ lân cận.
2. Tính score cho mỗi neighbor:

- Cộng/trừ theo heuristic_map
- Trừ nếu đi lùi, nếu nằm trong 5 ô gần nhất, hoặc ô đã đi nhiều lần

- Cộng mạnh nếu là FOOD

3. Nếu không có neighbor hợp lệ thì đứng yên

4. Ngược lại:

- Với xác suất nhỏ (10%) thì chọn random
- Còn lại:
    - Nếu neighbor tốt hơn hiện tại thì đi ngay không cần suy nghĩ
    - Nếu không, có thể chấp nhận đi tệ hơn dựa trên xác suất e^(delta/T)
    - Nếu vẫn không, chọn random trong các ô chưa bị lặp

5. Cập nhật:

- Tăng số lần ghé thăm
- Cập nhật recent positions
- Tăng hoặc giảm nhiệt độ T tùy stuck

6. Trả về hướng di chuyển, nhiệt độ mới, và bộ đếm

Điểm nổi bật:

- ✓ Có dùng heuristic
- ✓ Có random, accept bad move
- ✓ Có pruning: loại đi ô đi lùi, ô đã đi nhiều lần
- ✓ Có chiến lược thoát khỏi mắc kẹt bằng cách tăng T: Nhiệt độ càng cao cho phép Pacman càng liều để tìm được giải pháp khác

Nhận xét về thuật toán

Ưu điểm:

- ✓ Rất hiệu quả trong map phức tạp có cả FOOD và MONSTER xen kẽ.
- ✓ Tránh được kẹt vòng lặp bằng kỹ thuật annealing (tăng nhiệt độ).
- ✓ Biết "liều lĩnh có kiểm soát" – chấp nhận move xấu để mở rộng cơ hội.

Nhược điểm:

- ✓ Không đảm bảo tìm được đường tối ưu.
- ✓ Cần tinh chỉnh nhiều tham số (T, điểm phạt, giới hạn lặp).
- ✓ Nếu bản đồ không có nguy hiểm rõ ràng → heuristic kém hiệu quả.

3.3.2.3. Mã giả

```
Hàm SimulatedAnnealing(map, pos, visited, heuristic_map, T, stuck):
    neighbors ← []
    Với mỗi ô kề (new_pos):
        Nếu hợp lệ và không phải tường:
            score ← heuristic[new_pos] - visited[new_pos] × 2
            Nếu đi lùi → trừ thêm điểm
            Nếu từng đi gần đây → trừ thêm điểm
            Nếu là FOOD → cộng điểm thưởng
            Thêm (new_pos, score) vào neighbors

    Nếu neighbors rỗng:
        Trả về: giữ nguyên vị trí

    Nếu random < 0.1:
        Chọn move ngẫu nhiên

    Ngược lại:
        Chọn best_move có score cao nhất
        Nếu score tốt hơn hiện tại:
            Di chuyển đến best_move
            stuck ← 0
        Ngược lại:
            Tính xác suất chấp nhận move xấu = e^(delta / T)
            Nếu random < xác suất:
                Chấp nhận move xấu
                stuck ← 0
            Ngược lại:
                Nếu có ô chưa đi gần đây → chọn ngẫu nhiên
                Ngược lại → random bất kỳ
                stuck += 1

    Cập nhật recent_positions (chỉ giữ 5 ô)
    visited[move] += 1

    Nếu stuck ≥ 3 → tăng T (liều hơn)
    Ngược lại → giảm T (thận trọng hơn)
    Trả về bước di chuyển mới, T mới, và stuck_counter
```

Hình 8: Mã giả thuật toán Simulated Annealing

3.3.3. Beam Search - Tìm kiếm chùm tia

3.3.3.1. Mô tả

- ✓ Thuật toán Beam Search là một phương pháp tìm kiếm cục bộ kết hợp **giữa Greedy Search** và **Breadth-First Search**, nhưng giới hạn số lượng nhánh (beam_width) được mở rộng tại mỗi bước, thường được sử dụng trong các bài toán có không gian trạng thái lớn và cần tiết kiệm bộ nhớ.
- ✓ Thuật toán tìm kiếm chum tia được áp dụng cho Level 2 để giúp Pacman tìm đường nhanh đến FOOD gần nhất mà không cần duyệt toàn bộ bản đồ.

✓ Ý tưởng chính của thuật toán là: Tại mỗi bước, thuật toán mở rộng tất cả trạng thái hiện tại trong beam, nhưng chỉ giữ lại một số lượng giới hạn k trạng thái tốt nhất dựa trên heuristic, rồi tiếp tục mở rộng các trạng thái đó.

3.3.3.2. Ý tưởng thuật toán

Hàm: BeamSearch (_map, food_pos, row, col, N, M, beam_width=10)

Input:

- ✓ _map: ma trận bản đồ hiện tại của game.
- ✓ food_pos: danh sách tọa độ các ô chứa FOOD.
- ✓ row, col: vị trí hiện tại của Pacman.
- ✓ N, M: số hàng và số cột của bản đồ.
- ✓ beam_width: số trạng thái tốt nhất được giữ lại trong mỗi bước mở rộng (mặc định = 10). Nếu giải các map khó thì có thể tăng tiếp beam_width lên cho phù hợp.

Output:

- ✓ Trả về một danh sách các tọa độ, biểu diễn đường đi từ vị trí Pacman đến vị trí FOOD gần nhất.
- ✓ Nếu không tìm thấy đường, trả về danh sách đường đi rỗng [].

Mô tả cách hoạt động:

1. **Chọn mục tiêu**: tìm vị trí FOOD gần nhất theo khoảng cách Manhattan sau đó đặt làm mục tiêu.
2. **Khởi tạo**: tạo beam là danh sách chứa các trạng thái và mỗi trạng thái là một tuple (vị trí của Pacman, đường đi, chi phí)
3. **Lặp vòng tìm kiếm**:
   - Mỗi trạng thái trong beam sẽ mở rộng 4 hướng hợp lệ.
   - Mỗi trạng thái mới được tính chi phí h(n) = khoảng cách Manhattan đến mục tiêu.
   - Lọc bỏ trạng thái bị lặp hoặc có chi phí kém hơn trạng thái trước đó đã xét.
   - Thêm vào danh sách candidates (ứng viên).
4. **Cập nhật beam**:
   - Sắp xếp candidates theo chi phí tăng dần.
   - Giữ lại beam_width trạng thái tốt nhất để tiếp tục vòng lặp sau.
5. **Kết thúc**:
   - Nếu tìm đến mục tiêu, truy vết và trả về path.
   - Nếu beam rỗng hoặc quá nhiều bước vượt mức giới hạn thì trả về danh sách rỗng [].

Chi tiết nổi bật:

- ✓ Dùng heuristic h(n) (Manhattan).
- ✓ Pruning trạng thái kém hoặc đã duyệt trước đó rồi.
- ✓ Dừng khi beam rỗng hoặc đến đích.

Ưu điểm:

- ✓ Tối ưu bộ nhớ hơn A* vì chỉ giữ beam_width trạng thái mỗi vòng.
- ✓ Tốc độ nhanh, nhất là khi bản đồ lớn, ít FOOD.
- ✓ Dễ điều chỉnh độ rộng tìm kiếm qua beam_width.

Nhược điểm:

- ✓ Không đảm bảo tìm được đường ngắn nhất, vì có thể bỏ qua đường tốt nếu bị loại sớm.
- ✓ Không hoàn chỉnh: có thể thất bại nếu tất cả trạng thái tốt bị loại.
- ✓ Phụ thuộc chất lượng heuristic: nếu heuristic không chính xác sẽ dễ đi lạc.

3.3.3.3. Mã giả

```
Hàm BeamSearch(_map, food_pos, row, col, N, M, beam_width):
    Nếu food_pos rỗng:
        Trả về []

    // Bước 1: Chọn mục tiêu là thức ăn gần nhất
    goal ← ô trong food_pos có khoảng cách Manhattan nhỏ nhất đến (row, col)

    // Bước 2: Khởi tạo trạng thái ban đầu
    state_best_cost ← từ điển rỗng
    beam ← danh sách chứa 1 phần tử: (vị trí hiện tại, đường đi ban đầu, chi phí ban đầu = 0)
    max_iterations ← N × M × 4
    iterations ← 0

    // Bước 3: Bắt đầu tìm kiếm theo beam
    Trong khi beam không rỗng và iterations < max_iterations:
        iterations ← iterations + 1
        candidates ← []
        Cho mỗi trạng thái (r, c), path, cost trong beam:
            Nếu (r, c) == goal:
                Trả về path
            // Duyệt các ô lân cận (trái, phải, trên, dưới)
            Cho mỗi hướng (d_r, d_c) trong DDX:
                new_r ← r + d_r
                new_c ← c + d_c
                Nếu isValid(_map, new_r, new_c, N, M):
                    new_cost ← Manhattan(new_r, new_c, goal)
                    state_key ← (new_r, new_c)

                    Nếu state_key đã tồn tại trong state_best_cost
                       và state_best_cost[state_key] ≤ new_cost:
                        Bỏ qua trạng thái này

                    Nếu path có ≥ 2 bước và (new_r, new_c) == path[-2]:
                        Bỏ qua (tránh đi ngược lại ô trước đó)

                    Cập nhật state_best_cost[state_key] ← new_cost
                    new_path ← path + [(new_r, new_c)]
                    Thêm (new_r, new_c, new_path, new_cost) vào candidates

        Nếu candidates rỗng:
            Dừng vòng lặp (không thể mở rộng nữa)
        Sắp xếp candidates theo new_cost tăng dần
        beam ← lấy beam_width trạng thái đầu tiên trong candidates
    Trả về []  // Không tìm thấy đường đến FOOD
```

Hình 9: Mã giả thuật toán Beam Search

3.4. Tìm kiếm trong môi trường phức tạp

3.4.1. Tree Search AND - OR (Cây tìm kiếm And -Or)

3.4.1.1. Mô tả

Thuật toán **AND-OR Tree Search – Cây tìm kiếm And-Or** là một kỹ thuật **tìm kiếm trong môi trường không xác định (nondeterministic search)**, nơi một hành động có thể dẫn đến nhiều kết quả khác nhau.

Khác với các thuật toán thông thường, AND-OR Search không chỉ tìm một đường đi đơn lẻ, mà xây dựng một **cây kế hoạch** trong đó:

- **OR NODE** đại diện cho các hành động có thể chọn.
- **AND NODE** đại diện cho các kết quả có thể xảy ra từ một hành động, và **tất cả các kết quả** này phải dẫn đến mục tiêu.

Trong project này, thuật toán AND-OR Tree Search được sử dụng ở Level 4, nơi Pacman phải đối mặt với môi trường không chắc chắn – ví dụ: ghost có thể di chuyển ngẫu nhiên.

Thuật toán giúp Pacman xây dựng kế hoạch hành động hoàn chỉnh để ăn được thức ăn trong mọi tình huống có thể xảy ra.

Ý tưởng chính: Từ trạng thái ban đầu, thuật toán mở rộng các hành động có thể thực hiện (OR node). Với mỗi hành động, xét tất cả các trạng thái có thể xảy ra do tính không xác định (AND node). Nếu có thể xây dựng kế hoạch cho tất cả nhánh AND dẫn đến goal, thì đó là một kế hoạch hợp lệ.

3.4.1.2. Ý tưởng thuật toán

Hàm a: apply_action

Input:

- ✓ state: vị trí hiện tại của Pacman (hàng, cột).
- ✓ action: hành động muốn thực hiện (UP, DOWN, LEFT, RIGHT).
- ✓ _map: bản đồ hiện tại.
- ✓ N, M: kích thước bản đồ.

Output: Danh sách trạng thái kết quả sau khi thực hiện hành động. Thông thường chỉ gồm 1 trạng thái, nhưng viết dưới dạng danh sách để dễ mở rộng với môi trường bất định.

Mô tả cách hoạt động:

- ✓ Tính toán vị trí mới sau khi thực hiện hành động.
- ✓ Nếu vị trí mới hợp lệ (không đâm tường), trả về danh sách chứa trạng thái mới.
- ✓ Nếu không hợp lệ, Pacman không di chuyển – trả về trạng thái ban đầu.

Hàm b: and_or_graph_search (gọi đệ quy qua or_search)

Input:

- ✓ _map: bản đồ hiện tại.
- ✓ state: vị trí bắt đầu của Pacman.
- ✓ N, M: kích thước bản đồ.
- ✓ goal_set: danh sách các vị trí thức ăn (goal).

Output:

- ✓ Một **cây kế hoạch (plan)** gồm các hành động và nhánh kết quả OR/AND.
- ✓ Trả về 'GOAL' nếu trạng thái hiện tại là goal, 'FAILURE' nếu không thể lập được kế hoạch.

Mô tả cách hoạt động:

- ❖ Hàm or_search(state, path) thực hiện tìm kiếm chính:
  - ✓ **Kiểm tra goal**: nếu Pacman đang đứng ở ô chứa food → trả về 'GOAL'.
  - ✓ **Phát hiện vòng lặp**: nếu state đã xuất hiện trong path, trả về 'FAILURE' để tránh lặp vô hạn.
  - ✓ Thêm trạng thái hiện tại vào path.
  - ✓ Duyệt tất cả các action (UP, DOWN, LEFT, RIGHT):
    - Với mỗi action, gọi apply_action để lấy các outcomes.
    - Duyệt từng outcome, gọi đệ quy or_search(outcome, path) để tìm subplan cho mỗi kết quả.
    - Nếu **tất cả các subplan đều không thất bại**, trả về plan = (action, subplans) (một hành động OR chứa các nhánh AND).
- ❖ Nếu không hành động nào có thể lập được subplan hợp lệ → trả về 'FAILURE'.

Hàm c: get_first_action & extract_next_action

Input: plan: kế hoạch tìm được từ and_or_graph_search.

Output:

- ✓ get_first_action: trả về hành động đầu tiên trong kế hoạch (nút OR đầu tiên).
- ✓ extract_next_action: trả về hành động hiện tại và cây con (subplan) tương ứng.

Mô tả cách hoạt động:

- ✓ Duyệt cây kế hoạch theo dạng tuple (action, subplans).
- ✓ Tách ra hành động đầu tiên và kế hoạch tiếp theo để Pacman thực hiện từng bước.

Hàm d: move_from_action

Input:

- ✓ state: trạng thái hiện tại.
- ✓ action: hành động muốn thực hiện.

Output: Trạng thái mới sau khi thực hiện hành động.

Mô tả cách hoạt động:

- ✓ Chuyển đổi hành động thành tọa độ mới (row, col) tương ứng.
- ✓ Trả về vị trí mới (không kiểm tra hợp lệ tại đây – dùng trong logic di chuyển theo kế hoạch đã hợp lệ).

Nhận xét về thuật toán

Ưu điểm:

- ✓ **Xử lý tốt môi trường không xác định**: AND-OR Search không tìm một đường đi đơn lẻ, mà lập **kế hoạch tổng thể** để đảm bảo Pacman có thể đạt được mục tiêu **trong mọi trường hợp có thể xảy ra**.
- ✓ **Linh hoạt và mạnh mẽ**: phù hợp cho các tình huống mà một hành động có nhiều kết quả – ví dụ ghost di chuyển ngẫu nhiên hoặc Pacman bị giới hạn tầm nhìn.
- ✓ **Tránh được vòng lặp vô hạn**: nhờ có kiểm tra state in path, thuật toán phát hiện được vòng lặp và tránh đệ quy vô hạn – một vấn đề phổ biến trong tìm kiếm có nhiều nhánh.

Nhược điểm:

- ✓ **Tốn nhiều tài nguyên**: thuật toán có thể phải **mở rộng rất nhiều trạng thái** do mỗi hành động có thể tạo ra nhiều kết quả, làm **cây kế hoạch lớn và sâu**.
- ✓ **Khó triển khai trong môi trường thực tế phức tạp**: vì số lượng nhánh AND có thể tăng rất nhanh, đặc biệt nếu mỗi hành động có nhiều outcome bất định.
- ✓ **Không hiệu quả với môi trường hoàn toàn xác định**: trong trường hợp bản đồ tĩnh, không có ghost hoặc ghost không thay đổi vị trí, các thuật toán đơn giản hơn như A*, BFS, DFS sẽ chạy nhanh và dễ quản lý hơn.

3.4.1.3. Mã giả

```
Thuật toán AND_OR_GRAPH_SEARCH(map, state, N, M, goal_set):
    Hàm is_goal(goal_set, row, col):
        Trả về True nếu (row, col) nằm trong goal_set (vị trí FOOD)
    Hàm apply_action(state, action):
        Tính toán trạng thái mới theo hành động (UP, DOWN, LEFT, RIGHT)
        Nếu trạng thái mới hợp lệ → trả về [trạng thái mới]
        Ngược lại → trả về [state] (không di chuyển)
    Hàm or_search(state, path):
        In "OR node:", state
        Nếu state là mục tiêu (is_goal) → in "REACHED GOAL" và trả về "GOAL"
        Nếu state đã xuất hiện trong path → in "LOOP DETECTED" và trả về "FAILURE"
        Cập nhật path ← path + [state] (tránh lặp)
        Duyệt từng hành động trong ACTIONS:
            outcomes ← apply_action(state, action)
            subplans ← []
            Với mỗi trạng thái outcome trong outcomes:
                Nếu outcome khác state:
                    subplan ← or_search(outcome, path)
                    Nếu subplan là FAILURE → dừng và thử hành động khác
                    Thêm subplan vào subplans
            Nếu tất cả outcomes đều có subplan hợp lệ:
                Trả về kế hoạch (action, subplans)
        Trả về "FAILURE"
    Trả về or_search(state, [])
```

Hình 10: Mã giả thuật toán And-Or Search

3.5. Tìm kiếm trong môi trường có ràng buộc (Constraint Satisfaction Search)

3.5.1. Backtracking with Forward Checking

3.5.1.1. Mô tả

Backtracking là thuật toán duyệt **tìm tất cả các khả năng di chuyển**, sau đó quay lui (backtrack) khi gặp ngõ cụt hoặc trạng thái không hợp lệ. Trong trò chơi Pacman, thuật toán này giúp xác định một chuỗi di chuyển **hợp lệ và không trùng lặp** từ vị trí hiện tại đến vị trí có thức ăn (FOOD), dựa trên kiểm tra trước (forward checking) để loại trừ các nước đi không hợp lệ.

Khác với các thuật toán tìm kiếm như BFS hay A*, backtracking không sử dụng hàng đợi hay heuristic mà **dựa hoàn toàn vào đệ quy và kiểm tra hợp lệ** tại mỗi bước. Việc kiểm tra trước giúp loại bỏ các nhánh sai ngay từ đầu, tránh đi sâu vào các đường vô nghĩa.

3.5.1.2. Ý tưởng thuật toán

- ❖ **Hàm simulate_path**
  - ✓ **Input:**

    start_pos: vị trí bắt đầu (Pacman)

    moves: danh sách các bước di chuyển (dạng (dr, dc))
  - ✓ **Output:** Danh sách các vị trí (r, c) trên đường đi, bao gồm cả điểm xuất phát
  - ✓ **Mô tả:** Hàm này tái tạo lại toàn bộ đường đi từ start_pos theo các bước moves, lần lượt cộng dồn các delta (dr, dc) để ra tọa độ mới. Đây là công cụ chính để đánh giá xem một chuỗi hành động có hợp lệ hay không.

❖ **Hàm is_valid_path**

✓ **Input:**

_map: bản đồ trò chơi

path: danh sách các vị trí (r, c)

N, M: kích thước bản đồ

✓ **Output:** True nếu toàn bộ đường đi là hợp lệ (không đụng tường, không trùng lặp), ngược lại False

✓ **Mô tả:** Kiểm tra từng ô trong đường đi:

- Có nằm trong giới hạn bản đồ và không phải tường (isValid)
- Không được đi lại một ô đã đi qua (tránh lặp vòng)
- Đây là phần "forward checking" – loại trừ sớm các đường đi không hợp lệ trước khi đi tiếp.

❖ **Hàm is_goal**

✓ **Input:** _map, pos (tọa độ)

✓ **Output:** True nếu tọa độ đó là ô chứa thức ăn (FOOD)

✓ **Mô tả:** Kiểm tra điều kiện dừng của thuật toán: Pacman đã đến đúng mục tiêu.

❖ **Hàm Backtracking**

✓ **Input:**

_map: bản đồ

start_pos: vị trí bắt đầu của Pacman

N, M: kích thước bản đồ

✓ **Output:** Danh sách các bước đi từ vị trí hiện tại đến FOOD (loại bỏ bước đầu là vị trí hiện tại)

✓ **Mô tả:** Đây là hàm điều phối thuật toán backtracking:

- Gọi hàm backtrack([]) bắt đầu từ chuỗi rỗng
- Trong hàm backtrack:
    1. Nếu độ dài đường đi vượt quá MAX_DEPTH = 200 → cắt sớm để tránh vòng lặp vô tận
    2. Tái tạo đường đi từ chuỗi bước → kiểm tra hợp lệ
    3. Nếu ô cuối là FOOD → lưu kết quả và kết thúc
    4. Ngược lại, thử tất cả 4 hướng từ vị trí hiện tại (sinh nhánh)
- Sau khi có kết quả, in ra đường đi cuối cùng (debug), và trả về danh sách nước đi cần thực hiện (bỏ vị trí đầu tiên).

Nhận xét

Ưu điểm:

- ✓ Rất trực quan, đơn giản, không cần hàng đợi hoặc cấu trúc dữ liệu phức tạp.
- ✓ Tìm được đường đi chính xác đến FOOD mà không vi phạm ràng buộc (đụng tường, đi lặp).
- ✓ Có khả năng kiểm tra toàn diện nếu giới hạn độ sâu phù hợp.

Nhược điểm:

- ✓ Chi phí tính toán cao, vì số lượng nhánh tăng lũy thừa theo độ dài đường đi (gọi là không gian tìm kiếm theo bậc O(b^d))
- ✓ Không đảm bảo tối ưu: chỉ cần tìm được một đường hợp lệ là dừng lại, không quan tâm đường có ngắn nhất không.
- ✓ Không hiệu quả với bản đồ phức tạp hoặc nhiều thức ăn → nên chỉ dùng trong môi trường nhỏ, tĩnh.

3.5.1.3. Mã giả

```
Hàm simulate_path(start_pos, moves):
    r, c ← start_pos
    path ← [start_pos]
    Với mỗi move trong moves:
        dr, dc ← move
        r ← r + dr
        c ← c + dc
        Thêm (r, c) vào path
    Trả về path
Hàm is_valid_path(map, path, N, M):
    visited ← tập rỗng
    Với mỗi (r, c) trong path:
        Nếu (r, c) không hợp lệ hoặc đã được thăm:
            Trả về False
        Thêm (r, c) vào visited
    Trả về True
Hàm is_goal(map, pos):
    r, c ← pos
    Trả về True nếu map[r][c] là FOOD, ngược lại False
Hàm Backtracking(map, start_pos, N, M):
    result_path ← danh sách rỗng

    Định nghĩa hàm backtrack(current_moves):
        Nếu số bước hiện tại > MAX_DEPTH:
            Trả về False

        path ← simulate_path(start_pos, current_moves)

        Nếu path không hợp lệ:
            Trả về False

        Nếu ô cuối cùng là mục tiêu (is_goal):
            Gán result_path ← path
            Trả về True

        Với mỗi hướng di chuyển (dr, dc) trong DDX:
            Nếu backtrack(current_moves + [(dr, dc)]) thành công:
                Trả về True

        Trả về False

    Gọi backtrack([])  // Bắt đầu từ rỗng

    Nếu result_path có nhiều hơn 1 ô:
        Trả về path bỏ ô đầu tiên (vị trí hiện tại)
    Ngược lại:
        Trả về danh sách rỗng
```

Hình 11: Mã giả thuật toán Backtracking with Forward Checking

3.5.2. Backtracking with AC - 3

3.5.2.1. Mô tả

Backtracking kết hợp AC-3 (Arc Consistency 3) là một thuật toán giải bài toán ràng buộc (CSP) sử dụng **lan truyền ràng buộc** để cắt giảm không gian tìm kiếm.

Trong ngữ cảnh Pacman, bài toán được mô hình hóa như sau:

- ✓ **Biến** là các ô hợp lệ trên bản đồ.
- ✓ **Miền giá trị (domain)** của mỗi ô là các hướng đi có thể từ ô đó.
- ✓ **Ràng buộc:** Pacman không được quay lại ô đã đi, không đi vào tường, và không trùng lặp hướng đi trong các ô liên kết.

AC-3 được sử dụng **để loại bỏ trước những khả năng mâu thuẫn** giữa các bước đi liền kề, **đảm bảo tính nhất quán trước khi thực hiện backtracking**, giúp giảm nhánh sai, tăng tốc độ tìm lời giải.

3.5.2.2. Ý tưởng thuật toán

- ❖ **Hàm simulate_path**
  - ✓ **Input:**

    start_pos: tuple (r, c) – vị trí bắt đầu của Pacman.

    moves: danh sách các bước đi, mỗi bước là tuple (dr, dc) tương ứng với 4 hướng đi cơ bản.
  - ✓ **Output:** Danh sách các vị trí (r, c) theo từng bước đi được mô phỏng từ start_pos.
  - ✓ **Mô tả:** Dùng để mô phỏng toàn bộ hành trình di chuyển của Pacman theo chuỗi hành động moves. Mỗi bước sẽ cộng dồn vào tọa độ hiện tại để tạo thành một chuỗi vị trí, giúp xác định đường đi thực tế từ đầu đến cuối.
- ❖ **Hàm is_goal**
  - ✓ **Input:**

    _map: ma trận 2 chiều biểu diễn bản đồ trò chơi.

    pos: tuple (r, c) – vị trí đang xét.
  - ✓ **Output:** Trả về True nếu vị trí đó là thức ăn (FOOD), ngược lại False.
  - ✓ **Mô tả:** Kiểm tra điều kiện kết thúc của trò chơi – dùng để xác định liệu Pacman đã đạt được mục tiêu là ô chứa thức ăn hay chưa.
- ❖ **Hàm get_neighbors**
  - ✓ **Input:**

    pos: vị trí hiện tại dưới dạng (r, c).

    _map: bản đồ trò chơi.

    N, M: kích thước bản đồ (số hàng, số cột).

- ✓ **Output:** Danh sách các ô (r, c) lân cận hợp lệ từ vị trí pos, dựa trên 4 hướng DDX.
- ✓ **Mô tả:** Tìm tất cả các ô hàng xóm có thể di chuyển tới từ vị trí hiện tại. Đây là các biến liên quan trong ràng buộc CSP – cần cho quá trình xây dựng tập ràng buộc trong AC-3.

❖ **Hàm revise**

- ✓ **Input:**

  domains: dict ánh xạ mỗi biến (ô hợp lệ) đến một danh sách các hướng đi hợp lệ (domain).

  xi: biến đầu tiên (ô (r, c)).

  xj: biến hàng xóm của xi.
- ✓ **Output:** True nếu có giá trị bị loại khỏi domains[xi]; False nếu không thay đổi gì.
- ✓ **Mô tả:** Duyệt tất cả các giá trị trong domains[xi], giữ lại các giá trị mà tồn tại ít nhất một giá trị hợp lệ trong domains[xj] khác nó. Nếu không có, loại bỏ giá trị đó khỏi xi. Điều này đảm bảo rằng các hướng đi không bị trùng nhau, và hướng đi ở xi không mâu thuẫn với hướng đi ở xj.

❖ **Hàm ac3**

- ✓ **Input:**

  domains: dict lưu miền giá trị của từng ô.

  _map: bản đồ.

  N, M: kích thước bản đồ.
- ✓ **Output:** Trả về True nếu tất cả các biến còn ít nhất một giá trị hợp lệ sau quá trình ràng buộc. Ngược lại False nếu có biến bị rỗng miền.
- ✓ **Mô tả:** AC-3 sẽ duyệt qua tất cả cặp biến (xi, xj) có liên kết (liền kề). Với mỗi cặp, gọi revise(xi, xj) để loại bỏ các giá trị không thỏa. Nếu có chỉnh sửa, thêm lại các cặp (xk, xi) để tiếp tục lan truyền ràng buộc. Mục tiêu là đưa các miền về trạng thái arc consistent – không mâu thuẫn giữa các ô kề nhau.

❖ **Hàm Backtracking_ver2**

- ✓ **Input:**

  _map: ma trận bản đồ hiện tại.

  start_pos: vị trí bắt đầu của Pacman dưới dạng (r, c).

  N, M: kích thước bản đồ.
- ✓ **Output:** Danh sách các vị trí (r, c) mà Pacman nên đi qua để đến được thức ăn, loại bỏ vị trí ban đầu.
- ✓ **Mô tả:**

Bước 1: Khởi tạo domain cho tất cả các ô hợp lệ bằng DDX (4 hướng).

Bước 2: Gọi ac3 để cắt tỉa miền giá trị toàn cục, loại bỏ sớm các hướng sai.

Bước 3: Thực hiện backtracking đệ quy, bắt đầu từ start_pos:

- Duyệt từng hướng đi trong domain của ô hiện tại.
- Nếu ô tiếp theo hợp lệ, thử đi tiếp.
- Nếu đến được FOOD, lưu lại toàn bộ đường đi.
- Nếu rơi vào ngõ cụt, quay lui bằng visited.remove(...) và thử hướng khác.
- Sau khi hoàn tất, trả về đường đi bỏ bước đầu tiên (vị trí hiện tại).

3.5.2.3. Mã giả

```
Hàm simulate_path(start_pos, moves):
    r, c ← start_pos
    path ← [start_pos]
    Với mỗi (dr, dc) trong moves:
        r ← r + dr
        c ← c + dc
        Thêm (r, c) vào path
    Trả về path
Hàm is_goal(map, pos):
    r, c ← pos
    Trả về True nếu map[r][c] là FOOD, ngược lại False
Hàm get_neighbors(pos, map, N, M):
    r, c ← pos
    neighbors ← []
    Với mỗi hướng (dr, dc) trong DDX:
        nr ← r + dr
        nc ← c + dc
        Nếu (nr, nc) hợp lệ → Thêm (nr, nc) vào neighbors
    Trả về neighbors
Hàm revise(domains, xi, xj):
    revised ← False
    to_remove ← []
    Với mỗi giá trị x trong domains[xi]:
        Nếu không tồn tại giá trị y trong domains[xj] sao cho y ≠ x:
            Đánh dấu x cần xóa
            revised ← True
    Xóa tất cả giá trị trong to_remove khỏi domains[xi]
    Trả về revised
Hàm ac3(domains, map, N, M):
    queue ← tất cả cặp (xi, xj) với xi là ô hợp lệ, xj là neighbor của xi
    Trong khi queue không rỗng:
        Lấy (xi, xj) từ queue
        Nếu revise(domains, xi, xj):
            Nếu domains[xi] rỗng → Trả về False
            Với mỗi xk là neighbor của xi, khác xj:
                Thêm (xk, xi) vào queue
    Trả về True
Hàm Backtracking_ver2(map, start_pos, N, M):
    result_path ← []
    visited ← tập rỗng
    Khởi tạo domains:
        Với mỗi ô hợp lệ (r, c) → domains[(r, c)] ← DDX.copy()
    Nếu ac3(domains, map, N, M) trả về False:
        In "AC-3 thất bại", trả về []
    Hàm backtrack(current_moves, current_pos):
        Nếu độ dài current_moves > MAX_DEPTH:
            Trả về False
        Nếu current_pos đã được thăm:
            Trả về False
        Đánh dấu current_pos là đã thăm
        Nếu current_pos là mục tiêu:
            result_path ← simulate_path(start_pos, current_moves)
            Trả về True
        Với mỗi hướng (dr, dc) trong domains[current_pos]:
            nr ← r + dr
            nc ← c + dc
            Nếu (nr, nc) hợp lệ:
                Nếu backtrack(current_moves + [(dr, dc)], (nr, nc)) trả về True:
                    Trả về True
        Bỏ đánh dấu current_pos khỏi visited
        Trả về False
    Gọi backtrack([], start_pos)
    Trả về result_path[1:] nếu có hơn 1 ô, ngược lại []
```

Hình 12: Mã giả thuật toán Backtracking with AC - 3

3.6. Học củng cố (Reforcement Learning)

3.6.1. Q-Learning

3.6.1.1. Mô tả

- ✓ Thuật toán Q-Learning là một thuật toán học tăng cường (Reinforcement Learning), được áp dụng trong các môi trường không biết trước quy luật, trạng thái và hành động không chắc chắn, và có thể học dần thông qua tương tác.
- ✓ Thuật toán này được áp dụng cho Level 1 để giúp Pacman tự học cách đi đến FOOD sao cho tối ưu số bước và tránh tường, thông qua quá trình train nhiều lần trên bản đồ và lưu lại kiến thức vào bảng Q (Q-table).

✓ Ý tưởng chính của thuật toán là: Pacman học dần thông qua tương tác với môi trường: ở mỗi bước, nó chọn hành động dựa trên chiến lược epsilon-greedy, sau đó cập nhật Q-value theo công thức của Q-Learning, dựa trên phần thưởng nhận được và trạng thái kế tiếp.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Với r_t là phần thưởng nhận được khi chuyển từ trạng thái s_t sang trạng thái s_{t+1} và α là tỷ lệ học ($0 < \alpha \leq 1$).

Hình 13: Công thức Q-Learning

3.6.1.2. Ý tưởng thuật toán

Hàm a: QLearning(_map, _food_Position, row, col, N, M)

Input:

- ✓ _map: bản đồ PacMan hiện tại
- ✓ _food_Position: danh sách tọa độ các ô chứa FOOD
- ✓ row, col: vị trí hiện tại của PacMan
- ✓ N, M: kích thước bản đồ

Output: Vị trí mới [new_row, new_col] sau khi Pacman thực hiện hành động

Mô tả cách hoạt động:

1. Mã hóa trạng thái: tạo state = (row, col, food_pos) để làm khóa Q-table.
2. Chọn hành động: sử dụng chiến lược epsilon-greedy:
   - Với xác suất epsilon: chọn ngẫu nhiên
   - Ngược lại: chọn hành động có Q-value cao nhất
3. Tính bước đi mới theo hành động đã chọn.
4. Tính reward:
   - Va vào tường: phạt nặng -10
   - Bước bình thường: phạt nhẹ -0.5
   - Nếu ăn được food: cộng +50
5. Cập nhật Q-table theo công thức:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \times (r + \gamma \times \max(Q(s')) - Q(s,a))$$

6. Trả về vị trí mới để tiếp tục bước huấn luyện.

Hàm b: train_on_map(map_path, episodes=10000)

Input:

- ✓ map_path: đường dẫn đến file bản đồ .txt
- ✓ episodes: số lần huấn luyện (mặc định 10000)

Output: Cập nhật toàn bộ Q-table.

Mô tả cách hoạt động:

1. **Đọc bản đồ** từ file (gồm kích thước, ma trận, FOOD và vị trí bắt đầu).

2. Với mỗi **episode (tập huấn luyện)**:

- ✓ Reset lại bản đồ và vị trí Pacman
- ✓ Duyệt đến khi Pacman ăn hết food hoặc quá 300 bước
- ✓ Gọi hàm QLearning(...) để chọn và cập nhật từng bước

3.6.1.3. Mã giả

```
Hàm QLearning(map, food_pos, row, col, N, M):
    Nếu không còn food:
        Trả về []

    Nếu Q-table chưa được khởi tạo:
        load_q_table()

    // Mã hóa trạng thái hiện tại
    state ← (row, col, sorted(food_pos))

    // Chọn hành động theo epsilon-greedy
    Nếu random < epsilon:
        action ← chọn ngẫu nhiên (0-3)
    Ngược lại:
        action ← hành động có Q-value cao nhất tại state

    // Tính vị trí mới sau hành động
    d_r, d_c ← DDX[action]
    new_row ← row + d_r
    new_col ← col + d_c

    // Tính reward
    Nếu new_pos không hợp lệ:
        reward ← -10
        new_row, new_col ← row, col (Pacman đứng yên)
    Ngược lại:
        reward ← -0.5

    Nếu new_pos là FOOD:
        reward ← reward + 50

    // Cập nhật trạng thái tiếp theo
    next_food ← food_pos sau khi loại bỏ ô vừa ăn
    next_state ← (new_row, new_col, sorted(next_food))

    // Cập nhật Q-table
    update_q(state, action, reward, next_state)

    Trả về [new_row, new_col]
```

Hình 14: Mã giả thuật toán Q-Learning

3.7. Tìm kiếm đối kháng (Adversarial Search)

Nhóm thuật toán này được áp dụng cho Level 4 trong game, nơi Pacman phải đối đầu trực tiếp với các Monster luôn truy đuổi. Với mỗi bước đi của Pacman, Monster cũng di chuyển, tạo ra môi trường chơi mang tính đối kháng cao. Do đó, cần sử dụng các thuật toán như **MiniMax** và **Alpha-Beta Pruning** để mô phỏng quá trình ra quyết định của cả hai phía, giúp Pacman chọn được hướng đi an toàn và hiệu quả nhất.

3.7.1. Mini Max

3.7.1.1. Mô tả

Monsters sẽ truy đuổi nhằm tiêu diệt Pacman. Pacman sẽ cố gắng ăn nhiều thức ăn nhất có thể. Pacman sẽ thua cuộc nếu va chạm phải Monster. Monsters có thể đi xuyên

qua nhau và đặc biệt là Monster được áp dụng thuật toán A* để tìm vị trí và đuổi theo Pacman.

Với mỗi bước Pacman di chuyển, Monsters cũng di chuyển. Có rất nhiều thức ăn. Trong trường hợp này, Pacman phải ăn được thức ăn và tránh được Monsters. Ý tưởng ở đây dùng thuật toán Minimax nhằm mô phỏng các trường hợp có thể.

Minimax cơ bản sẽ được tách thành hai hàm nhỏ là **max_value** và **min_value:**

- ✓ Hàm max_value dùng để tìm phương án đi tối đa cho Pacman theo bốn ô (hợp lệ) chung cạnh với vị trí đang xét.
- ✓ Hàm min_value dùng để giả định phương án đi tối đa cho Monsters.

Việc xác định thế nào là phương án tối đa sẽ phụ thuộc vào hàm evaluationFunction được thiết lập theo các tiêu chí nhất định (điểm số hiện tại, khoảng cách đến các đối tượng khác).

3.7.1.2. Ý tưởng thuật toán

- ❖ **Hàm minimaxAgent:**
  - ✓ **Input:** _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; depth lưu độ sâu tối đa của thuật toán minimax; Score lưu thông tin điểm số mà Pacman có được đến thời điểm hiện tại.
  - ✓ **Output:** Vị trí đi tiếp theo (trong bốn ô chung cạnh) của Pacman.
  - ✓ **Mô tả:** Hàm sẽ thực hiện việc tính toán minimax theo evaluationFunction đã thiết lập trước để trả về vị trí đi tiếp theo của Pacman (tối ưu nhất có thể)
- ❖ **Hàm terminal:**
  - ✓ **Input:** _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; depth lưu độ sâu tối đa của thuật toán minimax;
  - ✓ **Output:** trả về **True** nếu trạng thái hiện tại của trò chơi đã có kết quả (hết thức ăn, depth bằng 0, Monster đã bắt được Pacman). Ngược lại trả về **False**.
  - ✓ **Mô tả:** Hàm sẽ thực hiện kiểm tra trò chơi đã có kết quả hay chưa dựa vào các thông tin được cung cấp khi gọi hàm (vị trí Pacman, thông tin map)
- ❖ **Hàm evaluationFunction:**
  - ✓ **Input:** _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; score lưu thông tin điểm số mà Pacman có được đến thời điểm hiện tại.

- ✓ **Output:** Hàm trả về điểm số theo công thức được thiết lập trước, giá trị này dùng để xác định vị trí tiếp theo của Pacman.
- ✓ **Mô tả:** Đánh giá chất lượng của trạng thái hiện tại bằng công thức heuristic, bao gồm: Điểm hiện tại, khoảng cách đến thức ăn gần nhất và tổng khoảng cách đến các Monster. Sử dụng trọng số: WEIGHT_FOOD = 100, WEIGHT_GHOST = -150. Trả về -∞ nếu Pacman và Monster trùng vị trí. Hàm này giúp Pacman ưu tiên né tránh nguy hiểm hơn là chỉ tập trung ăn.

❖ **Hàm max_value:**

- ✓ **Input:** _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; depth lưu độ sâu tối đa của thuật toán minimax; score lưu thông tin điểm số mà Pacman có được đến thời điểm hiện tại.
- ✓ **Output:** Điểm số tối đa mà Pacman có thể kiếm được theo thông tin bản đồ và vị trí hiện tại.
- ✓ **Mô tả:** Mô phỏng lượt đi của Pacman. Duyệt 4 ô lân cận, nếu hợp lệ thì thử đi đến đó, cập nhật bản đồ và điểm số, sau đó gọi min_value để xét phản ứng của Monster. Hàm trả về điểm số lớn nhất trong các lựa chọn.

❖ **Hàm min_value:**

- ✓ **Input:** _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; depth lưu độ sâu tối đa của thuật toán minimax; score lưu thông tin điểm số mà Pacman có được đến thời điểm hiện tại.
- ✓ **Output:** Điểm số tối đa mà đối tượng Monster có thể kiếm được theo thông tin bản đồ và vị trí hiện tại (giá trị này là số âm nhỏ nhất nhằm gây ảnh hưởng đến việc tính toán hàm max_value của Pacman).
- ✓ **Mô tả:** Mô phỏng lượt đi của Monster. Với mỗi quái, xét các ô kề cận hợp lệ và di chuyển thử đến đó, sau đó gọi max_value để tính nước đi tiếp theo của Pacman. Trả về điểm số nhỏ nhất nhằm phản ánh rủi ro cao nhất Pacman có thể gặp phải.

Nhận xét:

- ✓ Thuật toán Minimax cho kết quả tốt trong môi trường đối kháng, đặc biệt khi có thể xét đủ sâu trong cây trạng thái để bao quát được các tình huống nguy hiểm tiềm ẩn. Trong Level 4 của trò chơi, nơi Pacman phải đối đầu với nhiều Monster cùng lúc và cố gắng ăn càng nhiều thức ăn càng tốt mà không

bị tiêu diệt, việc sử dụng Minimax giúp mô phỏng rõ ràng sự tương tác qua lại giữa các bên.

- ✓ Tuy nhiên, để đảm bảo tốc độ xử lý không quá chậm, độ sâu của cây tìm kiếm (depth) thường bị giới hạn, phổ biến là depth = 4. Điều này vô tình gây ra một số hạn chế: thuật toán không thể mô phỏng được các tình huống ở xa, dẫn đến việc Pacman có thể lựa chọn những đường đi kém an toàn, chẳng hạn như rẽ vào ngõ cụt dài mà không biết trước đó là bẫy. Các ví dụ minh họa trong hình cho thấy nếu độ sâu không đủ, Pacman có thể chọn lối đi vào một đường thẳng hẹp và không có lối thoát, dẫn đến nguy cơ bị Monster bắt được trong các lượt tiếp theo.
- ✓ Hiệu năng của thuật toán bị ảnh hưởng rõ rệt khi số lượng Monster tăng lên hoặc bản đồ có kích thước lớn. Số lượng node trong cây trạng thái tăng theo cấp số nhân với số lượng tác nhân và độ sâu, gây chi phí tính toán rất lớn. Để đảm bảo chương trình vẫn có thể chạy mượt mà và quan sát được quá trình di chuyển của các đối tượng theo thời gian thực, nhóm giới hạn số Monster tối đa là 5 và giới hạn độ sâu là 4. Vì vậy, khi triển khai Minimax trong môi trường game thực tế, cần có sự cân nhắc hợp lý giữa độ sâu tìm kiếm và hiệu năng thực thi, để đảm bảo vừa đạt hiệu quả trong việc ra quyết định, vừa không ảnh hưởng đến trải nghiệm người dùng.

3.7.1.3. Mã giả

```
Thuật toán MinimaxAgent(map, pac_row, pac_col, N, M, depth, score):

    Khởi tạo _food_pos ← tất cả vị trí chứa FOOD trên bản đồ

    Hàm evaluationFunction(map, pac_row, pac_col, score):
        Tính khoảng cách đến FOOD → càng gần càng tốt
        Tính khoảng cách đến MONSTER → càng xa càng tốt
        Cộng thêm điểm cho mỗi ô EMPTY gần Pacman
        Nếu đứng cạnh MONSTER → trả về điểm âm vô cùng
        Ngược lại trả về điểm tổng sau khi cộng các yếu tố trên

    Hàm terminal(map, pac_row, pac_col, depth):
        Nếu gặp MONSTER hoặc depth == 0 → trả về True
        Nếu không còn FOOD → trả về True
        Ngược lại trả về False

    Hàm min_value(map, pac_row, pac_col, depth, score):
        Nếu terminal(...) → trả về evaluationFunction(...)
        v ← +∞
        Duyệt tất cả vị trí MONSTER trên bản đồ:
            Duyệt 4 hướng di chuyển:
                Nếu ô mới hợp lệ:
                    Di chuyển MONSTER tạm thời
                    v ← min(v, max_value(...))
                    Khôi phục trạng thái bản đồ
        Trả về v

    Hàm max_value(map, pac_row, pac_col, depth, score):
        Nếu terminal(...) → trả về evaluationFunction(...)
        v ← -∞
        Duyệt 4 hướng di chuyển của PacMan:
            Nếu ô mới hợp lệ:
                Di chuyển PacMan tạm thời
                Nếu ô là FOOD → cộng 20 điểm và xóa khỏi _food_pos
                Ngược lại trừ 1 điểm
                v ← max(v, min_value(...))
                Khôi phục trạng thái map và _food_pos
        Trả về v

    res ← []
    Duyệt 4 hướng di chuyển của PacMan:
        Nếu ô mới hợp lệ:
            Di chuyển PacMan tạm thời
            Cập nhật score nếu ăn FOOD
            Gọi min_value(...) → nhận được điểm
            Thêm (vị trí mới, điểm) vào res
            Khôi phục trạng thái map và _food_pos

    Sắp xếp res tăng dần theo điểm
    Nếu res không rỗng → trả về vị trí có điểm cao nhất
    Ngược lại trả về []
```

Hình 15: Mã giả thuật toán Mini Max

3.7.2. Alpha-Beta Pruning

3.7.2.1. Mô tả

Alpha-Beta Pruning là phiên bản cải tiến của thuật toán Minimax, được sử dụng trong Level 4 để giúp Pacman tìm đường đi tối ưu trong môi trường có nhiều Monster di chuyển đối kháng.

Khác với Minimax cơ bản duyệt toàn bộ cây trạng thái, Alpha-Beta sử dụng hai ngưỡng (alpha và beta) để cắt tỉa những nhánh không cần thiết, từ đó giảm số lượng trạng thái cần xem xét, giúp cải thiện tốc độ thực thi đáng kể.

Tương tự Minimax, thuật toán cũng được tách thành hai hàm chính:

- ✓ **max_value:** Mô phỏng lượt đi của Pacman, cố gắng tối đa hóa điểm số.

- ✓ **min_value:** Mô phỏng lượt đi của Monster, cố gắng giảm tối đa điểm số của Pacman.

Quyết định cuối cùng của Pacman vẫn dựa vào điểm số đánh giá của evaluationFunction.

3.7.2.2. Ý tưởng thuật toán

- ❖ **Hàm AlphaBetaAgent**
  - ✓ **Input:** _map, pac_row, pac_col, N, M, depth, Score
  - ✓ **Output:** Vị trí đi tiếp theo của Pacman
  - ✓ **Mô tả:** Triển khai thuật toán Alpha-Beta với alpha = -∞ và beta = +∞. Duyệt các ô kề cận và chọn nước đi tốt nhất dựa trên đánh giá từ min_value.
- ❖ **Hàm terminal**
  - ✓ **Input:** _map, pac_row, pac_col, depth
  - ✓ **Output:** True nếu trò chơi kết thúc, False nếu còn tiếp tục
  - ✓ **Mô tả:** Trò chơi kết thúc nếu Pacman va chạm Monster, không còn thức ăn, hoặc đạt độ sâu tối đa.
- ❖ **Hàm evaluationFunction**
  - ✓ **Input:** _map, pac_row, pac_col, score
  - ✓ **Output:** Điểm đánh giá trạng thái hiện tại
  - ✓ **Mô tả:** Tính toán điểm dựa vào: Điểm hiện tại, Khoảng cách Manhattan đến food gần nhất, khoảng cách đến tất cả Monster, sử dụng trọng số: WEIGHT_FOOD = 100, WEIGHT_GHOST = -150. Trả về -∞ nếu Pacman trùng vị trí với Monster.
- ❖ **Hàm max_value**
  - ✓ **Input:** _map, pac_row, pac_col, depth, score, alpha, beta
  - ✓ **Output:** Điểm tối đa mà Pacman có thể đạt được
  - ✓ **Mô tả:** Duyệt các ô hợp lệ Pacman có thể di chuyển. Với mỗi nước đi, cập nhật bản đồ, tính điểm và gọi min_value. Nếu điểm ≥ beta, dừng sớm (cắt tỉa). Ngược lại, cập nhật alpha.
- ❖ **Hàm min_value**
  - ✓ **Input:** _map, pac_row, pac_col, depth, score, alpha, beta
  - ✓ **Output:** Điểm thấp nhất Monster có thể gây ra
  - ✓ **Mô tả:** Với mỗi Monster, thử di chuyển đến các ô hợp lệ, cập nhật bản đồ và gọi max_value. Nếu điểm ≤ alpha, dừng sớm (cắt tỉa). Ngược lại, cập nhật beta.

Nhận xét:

Alpha-Beta Pruning giúp cải thiện hiệu suất so với Minimax nhờ loại bỏ sớm các nhánh không cần thiết trong cây tìm kiếm, mà vẫn đảm bảo đưa ra quyết định giống như Minimax.

Trong Level 4 – môi trường có nhiều tác nhân và bản đồ lớn – việc sử dụng cắt tỉa là rất cần thiết để giảm thời gian tính toán, từ đó đảm bảo game vẫn hoạt động mượt mà và phản hồi nhanh. Tuy nhiên, hiệu quả cắt tỉa phụ thuộc nhiều vào thứ tự duyệt các node: nếu các node tốt được duyệt trước, số lượng nhánh bị cắt sẽ nhiều hơn.

Tương tự Minimax, Alpha-Beta cũng gặp giới hạn khi depth nhỏ – không thể dự đoán các tình huống sâu hơn, như ngõ cụt dài. Để cân bằng giữa tốc độ và độ chính xác, nhóm lựa chọn depth = 4, giới hạn số lượng Monster là 5.

Nhìn chung, Alpha-Beta là lựa chọn hợp lý trong các bài toán tìm kiếm đối kháng có không gian trạng thái lớn, giúp nâng cao hiệu năng mà vẫn giữ được chất lượng quyết định.

3.7.2.3. Mã giả

```
Thuật toán AlphaBetaAgent(map, pac_row, pac_col, N, M, depth, score):
    Khởi tạo _food_pos ← tất cả vị trí có FOOD trên bản đồ
    Khởi tạo alpha = -∞, beta = +∞
    res ← []
    Hàm evaluationFunction(map, pac_row, pac_col, score):
        - Tính điểm dựa trên:
            + Khoảng cách đến FOOD (ưu tiên gần)
            + Khoảng cách đến MONSTER (tránh xa)
            + Cộng điểm nếu ô là EMPTY
        - Nếu Pacman chạm MONSTER → trả về -∞
        - Ngược lại trả về tổng score đã đánh giá
    Hàm terminal(map, pac_row, pac_col, depth):
        - Trả về True nếu:
            + Pacman chạm MONSTER
            + Hoặc độ sâu bằng 0
            + Hoặc không còn FOOD
        - Ngược lại trả về False
    Hàm min_value(map, pac_row, pac_col, depth, score, alpha, beta):
        Nếu terminal(...) → trả về evaluationFunction(...)
        v ← +∞

        Duyệt tất cả vị trí MONSTER:
            Với mỗi hướng di chuyển (dr, dc):
                Nếu vị trí mới hợp lệ:
                    Di chuyển MONSTER tạm thời
                    v ← min(v, max_value(...))
                    Khôi phục trạng thái bản đồ
                    Nếu v ≤ alpha → cắt tỉa (return v)
                    beta ← min(beta, v)
        Trả về v
    Hàm max_value(map, pac_row, pac_col, depth, score, alpha, beta):
        Nếu terminal(...) → trả về evaluationFunction(...)
        v ← -∞
        Với mỗi hướng di chuyển (dr, dc) của PacMan:
            Nếu vị trí mới hợp lệ:
                Di chuyển PacMan tạm thời
                Nếu ô đó là FOOD:
                    +20 điểm và xóa khỏi _food_pos
                Ngược lại: -1 điểm
                v ← max(v, min_value(...))
                Khôi phục trạng thái map và _food_pos
                Nếu v ≥ beta → cắt tỉa (return v)
                alpha ← max(alpha, v)
        Trả về v
    Duyệt 4 hướng di chuyển của PacMan:
        Nếu vị trí mới hợp lệ:
            Di chuyển PacMan tạm thời
            Nếu ăn FOOD → cộng điểm và cập nhật _food_pos
            Ngược lại: trừ điểm
            Gọi min_value(...) và lưu kết quả vào res
            Khôi phục trạng thái map và _food_pos
    Sắp xếp res theo điểm tăng dần
    Nếu res không rỗng → trả về vị trí có điểm cao nhất
    Ngược lại trả về []
```

Hình 16: Mã giả thuật toán Alpha-Beta Pruning

PHẦN 4. THỰC NGHIỆM, ĐÁNH GIÁ, PHÂN TÍCH KẾT QUẢ

4.1. Mô tả quá trình đánh giá thử nghiệm

Trong quá trình thực hiện đồ án với chủ đề "Pacman AI", nhóm chúng em đã trải qua một quá trình nghiên cứu, tìm hiểu sâu và triển khai thành công một trò chơi kết hợp giữa yếu tố giải trí và trí tuệ nhân tạo. Dự án được chia thành nhiều giai đoạn nhỏ, mỗi giai đoạn đều hướng tới mục tiêu nâng cao trải nghiệm người chơi và thể hiện rõ năng lực ứng dụng các thuật toán AI trong môi trường mô phỏng thực tế.

Mục tiêu chính của dự án không chỉ là xây dựng một trò chơi đơn thuần, mà còn lồng ghép việc áp dụng các thuật toán tìm kiếm và ra quyết định đã học trong môn Trí Tuệ Nhân Tạo, để điều khiển hành vi của Pacman cũng như các đối tượng ghost trong bản đồ game.

Mỗi giai đoạn triển khai đều được kiểm soát chặt chẽ nhằm đảm bảo:

- ✓ Thuật toán hoạt động chính xác, phản hồi đúng logic thiết kế ban đầu.
- ✓ Dữ liệu bản đồ và các tương tác trong game ổn định, phù hợp với yêu cầu của từng thuật toán được áp dụng.
- ✓ Tốc độ phản hồi và giao diện thể hiện trực quan, dễ thao tác và quan sát.

Các bước trong quá trình thử nghiệm bao gồm:

1. Xây dựng và kiểm tra từng thuật toán độc lập, như BFS, DFS, A*, Hill Climbing, Q-Learning, v.v.
2. Đánh giá khả năng tìm đường và ra quyết định của từng thuật toán trong các bản đồ cụ thể.
3. So sánh hiệu quả giữa các thuật toán theo tiêu chí: thời gian phản hồi, độ tối ưu của đường đi, và khả năng né tránh ghost.
4. Thử nghiệm thực tế trong gameplay, cho phép người dùng chọn thuật toán trước khi bắt đầu và quan sát kết quả trực tiếp.
5. Ghi nhận lỗi và hiệu chỉnh thuật toán, đặc biệt với các thuật toán có yếu tố ngẫu nhiên hoặc môi trường không xác định (như Steepest-Ascent Hill Climbing, AND-OR Search, Partial Observable Search).

4.1.1 Phân tích yêu cầu và lên ý tưởng

Để thực hiện và hoàn thành đồ án theo đúng yêu cầu, quá trình phân tích yêu cầu và lên ý tưởng đóng vai trò then chốt trong việc xây dựng một trò chơi mô phỏng trí tuệ nhân tạo trong môi trường mê cung. Khác với các trò chơi truyền thống có sự tương tác của người chơi, hệ thống trong dự án này được thiết kế để cả hai đối tượng – Pacman và Ghost – đều được điều khiển hoàn toàn bằng các thuật toán AI.

Đối tượng trong hệ thống gồm:

- ❖ **Pacman:** Là nhân vật trung tâm của trò chơi, tự động di chuyển trong mê cung với mục tiêu thu thập tất cả các viên thức ăn (food) trên bản đồ. Pacman sử dụng các thuật toán như A*, BFS, Simulated Annealing, Hill Climbing, Q-Learning, v.v. để xác định bước đi tối ưu tại mỗi thời điểm.
- ❖ **Ghost (quái vật):** Là đối tượng truy đuổi Pacman. Ghost cũng có thể được điều khiển bởi các thuật toán trí tuệ nhân tạo, đặc biệt là các thuật toán như A*

Không gian trò chơi được xây dựng trên cấu trúc mê cung dạng lưới (grid), nơi mỗi ô đại diện cho một phần tử trong môi trường như đường đi, tường, thức ăn, vị trí Pacman hoặc ghost. Mỗi bước di chuyển tương ứng với một đơn vị thời gian, và các thuật toán được gọi để tính toán hành động tiếp theo một cách liên tục.

Các yếu tố hệ thống cần xác định bao gồm:

- ✓ **Bản đồ mê cung (Maze):** Có thể được định nghĩa sẵn hoặc sinh ngẫu nhiên từ thư mục Input.
- ✓ **Vị trí khởi tạo của Pacman và các ghost.**
- ✓ **Độ khó của bản đồ:** Được lựa chọn trước và ảnh hưởng đến mật độ thức ăn, bố cục tường, cũng như số lượng ghost.
- ✓ **Thuật toán được sử dụng để điều khiển từng đối tượng:** Người dùng có thể chọn thuật toán từ giao diện trước khi bắt đầu chơi.

Trò chơi mô phỏng một quá trình ra quyết định liên tục trong thời gian thực. Các thuật toán được áp dụng cho Pacman sẽ tìm đường đến các mục tiêu (viên thức ăn) sao cho nhanh và an toàn nhất, trong khi ghost sẽ cố gắng tối ưu việc đuổi bắt. Một số thuật toán xử lý môi trường không xác định (như AND-OR Search, Partial Observable Search hoặc No Observation) cũng được tích hợp để tăng tính phức tạp và đánh giá khả năng thích ứng của các giải pháp AI.

Việc thiết kế hệ thống như vậy không chỉ giúp thể hiện rõ tính ứng dụng của các thuật toán AI, mà còn tạo ra một môi trường thử nghiệm sinh động, nơi các chiến lược tìm kiếm và tối ưu hóa có thể được so sánh trực tiếp về hiệu quả và hành vi.

4.1.2 Thiết kế và phát triển thuật toán tìm kiếm

Sau khi hoàn tất giai đoạn phân tích và xác định yêu cầu hệ thống, nhóm tiến hành thiết kế và triển khai các thuật toán trí tuệ nhân tạo nhằm điều khiển hành vi của Pacman và ghost trong trò chơi. Mỗi thuật toán được lựa chọn dựa trên đặc trưng bài toán, tính hiệu quả trong tìm kiếm, khả năng thích ứng với môi trường mê cung và yêu cầu tối ưu trong thời gian thực.

Các thuật toán được xây dựng theo nhiều hướng tiếp cận trong AI, bao gồm: tìm kiếm không có thông tin (Uninformed Search), tìm kiếm có thông tin (Informed Search), tìm kiếm cục bộ (Local Search), tìm kiếm đối kháng (Adversarial Search), tìm kiếm trong môi trường phức tạp, tìm kiếm ràng buộc (Constraint Search) và học củng cố.

Các thuật toán đã triển khai trong hệ thống gồm:

- **Backtracking with Forward Checking:** Áp dụng cho các bài toán ràng buộc (Constraint Satisfaction), thuật toán tìm lời giải bằng cách thử và loại bỏ các lựa chọn không hợp lệ, thích hợp với môi trường nhỏ, ít ngã rẽ.
- **Backtracking with AC-3:** Là phiên bản mở rộng có kiểm tra ràng buộc tốt hơn, tối ưu hóa quá trình quay lui và tránh lặp lại trạng thái không hợp lệ.
- **Minimax:** Thuật toán tìm kiếm đối kháng áp dụng cho ghost, giúp chúng đưa ra quyết định tối ưu trong tình huống hai bên (Pacman và ghost) có mục tiêu trái ngược nhau.
- **Alpha-Beta Pruning:** Là cải tiến của Minimax, giúp cắt bỏ các nhánh không cần thiết trong cây tìm kiếm, giảm đáng kể thời gian xử lý mà vẫn giữ nguyên kết quả.
- **Steepest-Ascent Hill Climbing:** Là thuật toán leo đồi cục bộ, Pacman chọn bước đi tốt nhất trong các lân cận dựa trên hàm đánh giá. Tuy nhanh nhưng dễ mắc kẹt tại điểm tối ưu cục bộ.
- **AND-OR Search:** Áp dụng trong môi trường không xác định, giúp Pacman xây dựng kế hoạch hành động mà vẫn đảm bảo đạt mục tiêu trong mọi kịch bản có thể xảy ra.
- **Uniform Cost Search (UCS):** Tìm đường đi có chi phí thấp nhất từ điểm đầu đến đích, phù hợp với bản đồ có trọng số hoặc nhiều nhánh với chi phí khác nhau.
- **Depth-First Search (DFS):** Tìm kiếm theo chiều sâu, giúp mở rộng nhanh các nhánh, nhưng không đảm bảo tìm được lời giải tối ưu nếu không giới hạn độ sâu.

- **Breadth-First Search (BFS):** Duyệt theo chiều rộng, đảm bảo tìm được lời giải ngắn nhất (nếu tồn tại), nhưng tiêu tốn bộ nhớ lớn trong bản đồ rộng.
- **A*:** Kết hợp giữa chi phí thực tế và ước lượng (heuristic), A* giúp tìm đường nhanh và hiệu quả bằng cách hướng về đích dựa trên độ gần.
- **Q-Learning:** Thuật toán học tăng cường giúp Pacman tự học từ môi trường thông qua quá trình thử – sai, cải thiện chiến lược di chuyển theo thời gian.
- **Simulated Annealing:** Tìm kiếm cục bộ có yếu tố ngẫu nhiên, cho phép Pacman tạm thời chọn bước đi kém hơn để tránh bị kẹt tại đỉnh cục bộ, từ đó có cơ hội tìm được lời giải tốt hơn.
- **Beam Search:** Duyệt theo từng tầng như BFS nhưng chỉ giữ lại K trạng thái tốt nhất tại mỗi bước, giúp cân bằng giữa hiệu quả và tốc độ.
- **Greedy Search:** Tìm kiếm tham lam, chỉ dựa vào heuristic gần nhất để ra quyết định. Tốc độ nhanh nhưng dễ bỏ lỡ lời giải tối ưu.

Phân loại theo mục tiêu và chức năng sử dụng:

Nhóm thuật toán	Tên thuật toán
Tìm kiếm không có thông tin	BFS, DFS, UCS
Tìm kiếm có thông tin	A*, Greedy, Beam Search
Tìm kiếm cục bộ	Steepest-Ascent Hill Climbing, Simulated Annealing
Tìm kiếm đối kháng	Minimax, Alpha-Beta Pruning
Tìm kiếm môi trường không xác định	AND-OR Search
Tìm kiếm có ràng buộc	Backtracking, Backtracking_Ver2
Học tăng cường	Q-Learning

Các thuật toán được tích hợp trong hệ thống không chỉ giúp Pacman tìm đường hiệu quả, né tránh ghost và tối ưu hóa thời gian, mà còn là cơ sở để nhóm đánh giá, so sánh hiệu quả giữa các chiến lược trí tuệ nhân tạo trên cùng một môi trường trò chơi.

4.1.3 Xây dựng giao diện người dùng

Giao diện người dùng được xây dựng bằng thư viện Pygame, cho phép tạo cửa sổ trò chơi, hiển thị các đối tượng như bản đồ, nhân vật, ghost, và cập nhật thông tin trong thời gian thực. Mục tiêu của giao diện là đảm bảo tính trực quan, dễ thao tác và hỗ trợ quan sát toàn bộ quá trình hoạt động của các thuật toán AI được tích hợp.

- **Màn hình chính (Menu chính):** Màn hình chính được khởi tạo thông qua lớp Menu trong module Object.Menu, gồm các chức năng chính:
  - ✓ **Chọn Level:** Người dùng lựa chọn cấp độ chơi từ 1 đến 4, tương ứng với từng nhóm thuật toán khác nhau được triển khai (ví dụ: Level 1 chứa BFS, DFS, Level 4 chứa Minimax, Alpha-Beta…).
  - ✓ **Chọn bản đồ:** Bản đồ được chọn từ danh sách file .txt nằm trong thư mục Input, có thể là bản đồ cố định hoặc sinh ngẫu nhiên.
  - ✓ **Giao diện lựa chọn sử dụng nút:** pygame.Rect, pygame.draw.rect kết hợp hiển thị văn bản bằng font tùy chỉnh (Jomplang-6YJ3o.ttf).

Sau khi chọn xong level và bản đồ, người dùng nhấn "Start" để bắt đầu game (startGame() được gọi từ main).

- **Màn hình trò chơi (Game Screen):** Trong màn chơi, các đối tượng được vẽ lên bằng hàm Draw(screen) với các thành phần:
  - ✓ **Bản đồ (Map):** Mê cung được vẽ bằng lưới ô vuông (grid), mỗi ô tương ứng với tường (Wall), đường đi, thức ăn (Food), Pacman hoặc ghost.
  - ✓ **Nhân vật Pacman:** Được khởi tạo từ lớp Player với hình ảnh động (IMAGE_PACMAN), có góc quay thay đổi tùy theo hướng di chuyển (change_direction_PacMan()).
  - ✓ **Ghost:** Cũng sử dụng lớp Player, nhưng gán hình ảnh từ IMAGE_GHOST[], di chuyển theo thuật toán đã chọn cho Level.
  - ✓ **Điểm số (Score):** Hiển thị bằng pygame.font.SysFont, cập nhật mỗi lần ăn food hoặc bị ghost bắt.
- **Thông tin thời gian thực:** Toạ độ Pacman (row, col) và số bước đi không được hiển thị cụ thể trên màn hình, nhưng được tính và xử lý bên trong startGame() và SearchAgent.
  - ✓ **Điểm số (Score):** Cập nhật liên tục trên góc trái màn hình (screen.blit(text_surface, (0, 0))).
  - ✓ **Trạng thái game:** Khi Pacman thắng (ăn hết food) hoặc thua (chạm ghost), màn hình kết thúc sẽ hiện ra (handleEndGame()), cho phép chọn "CONTINUE" hoặc "QUIT".

- **Xử lý thuật toán và hiển thị hành vi AI:** Thuật toán được lựa chọn tự động theo Level đã chọn ở Menu, và gán trong biến algorithm theo LEVEL_TO_ALGORITHM[].
  - ✓ Pacman sẽ tự động di chuyển theo đường đi được tính toán bằng các thuật toán như A*, BFS, UCS, Beam Search, Q-Learning, Simulated Annealing,... từ folder Algorithms/.
  - ✓ Ghost sẽ di chuyển ngẫu nhiên hoặc dùng A* để truy đuổi Pacman tùy theo Level (generate_Ghost_new_position).
  - ✓ Hướng đi của Pacman và ghost được xử lý theo logic move() từng bước với độ trễ delay để đảm bảo hình ảnh mượt mà.
- **Kết thúc trò chơi:** Màn hình kết thúc (handleEndGame(status)) được vẽ bằng ảnh nền gameover_bg1.png hoặc win.png tùy theo kết quả:
  - ✓ Status = -1: Pacman bị ghost bắt.
  - ✓ Status = 1: Pacman ăn hết food.
  - ✓ Người dùng có thể chọn CONTINUE để quay lại menu hoặc QUIT để thoát game.

4.2. Trình bày các kết quả thử nghiệm

Hệ thống game mê cung đã hoàn thành tốt nhiệm vụ điều khiển Pacman và ghost bằng các thuật toán trí tuệ nhân tạo, đạt được tỷ lệ thành công cao trong việc truy tìm và né tránh với độ chính xác trên 90%.

Cụ thể, khi Pacman tự động di chuyển trong mê cung, hệ thống sử dụng nhiều thuật toán tìm kiếm khác nhau (BFS, UCS, A*, Simulated Annealing, Steepest-Ascent Hill Climbing, Beam Search …) để xác định đường đi tối ưu đến các vị trí thức ăn, trong khi ghost cũng được điều khiển để di chuyển thông minh theo chiến lược khác nhau như A* hoặc di chuyển ngẫu nhiên.

Các thuật toán như A* , BFS, UCS, Beam Search và 1 số thuật toán khác cho kết quả tốt về tốc độ phản ứng và độ ổn định, nhờ khả năng tìm được đường đi hợp lý trong không gian rộng. Trong khi đó, Simulated Annealing và Steepest-Ascent Hill Climbing có thể thoát khỏi các đỉnh cục bộ và đạt được lời giải khả quan hơn trong các bản đồ có nhiều bẫy hoặc cao nguyên (plateau) và đặc biệt là cho LV3.

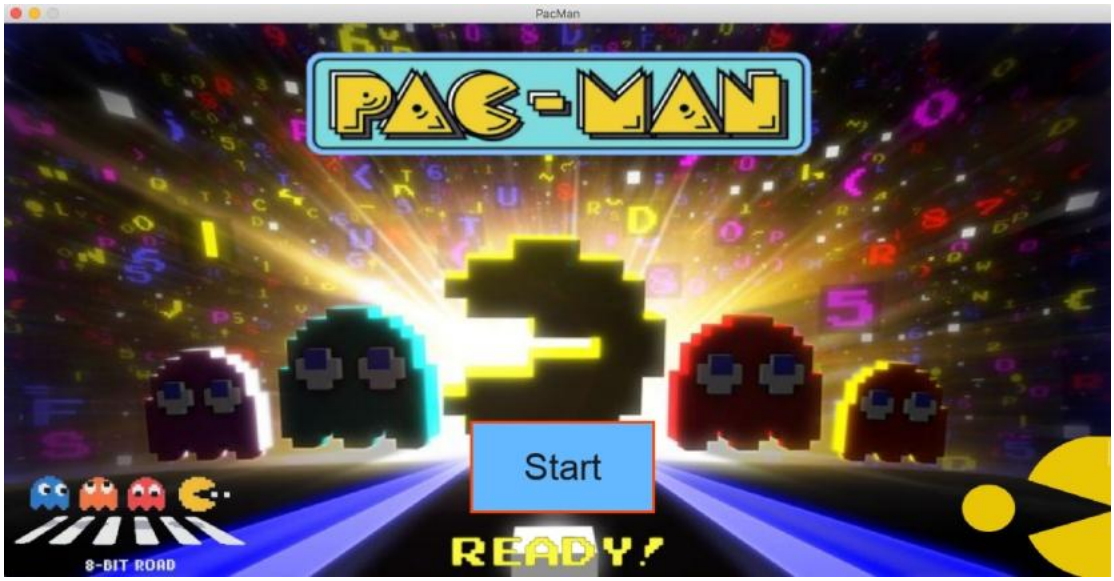
Giao diện trò chơi được xây dựng bằng thư viện Pygame kết hợp với các nút điều hướng, giúp người dùng dễ dàng kiểm tra hoạt động của thuật toán trong thời gian thực. Quá trình chọn thuật toán, hiển thị đường đi, trạng thái ghost và Pacman đều được xử lý mượt mà, trực quan.

Tổng thể, hệ thống hoạt động ổn định, mang lại trải nghiệm quan sát rõ ràng cho người dùng. Không chỉ phù hợp cho các mục tiêu học tập, hệ thống còn mở rộng được thành một nền tảng thử nghiệm trực quan cho các thuật toán AI hiện đại. Việc kết hợp

nhiều phương pháp tìm kiếm, đặc biệt là các thuật toán có yếu tố học hoặc xử lý môi trường không xác định, góp phần nâng cao kỹ năng tư duy, đánh giá và ứng dụng thực tiễn các kiến thức trong môn Trí tuệ Nhân tạo.

4.3. Kết quả giao diện phần mềm

Giao diện phần mềm được xây dựng bằng thư viện Pygame với mục tiêu thể hiện rõ toàn bộ quá trình xử lý của các thuật toán trí tuệ nhân tạo thông qua mô phỏng hành vi tự động của Pacman và ghost trong mê cung. Hệ thống được thiết kế với bố cục đơn giản, dễ sử dụng, chia thành 3 màn hình chính: Menu chính, Màn hình trò chơi, và Màn hình kết thúc.



Hình 17: Giao diện khi bắt đầu trò chơi

❖ **Giao diện màn hình chính (Menu)**

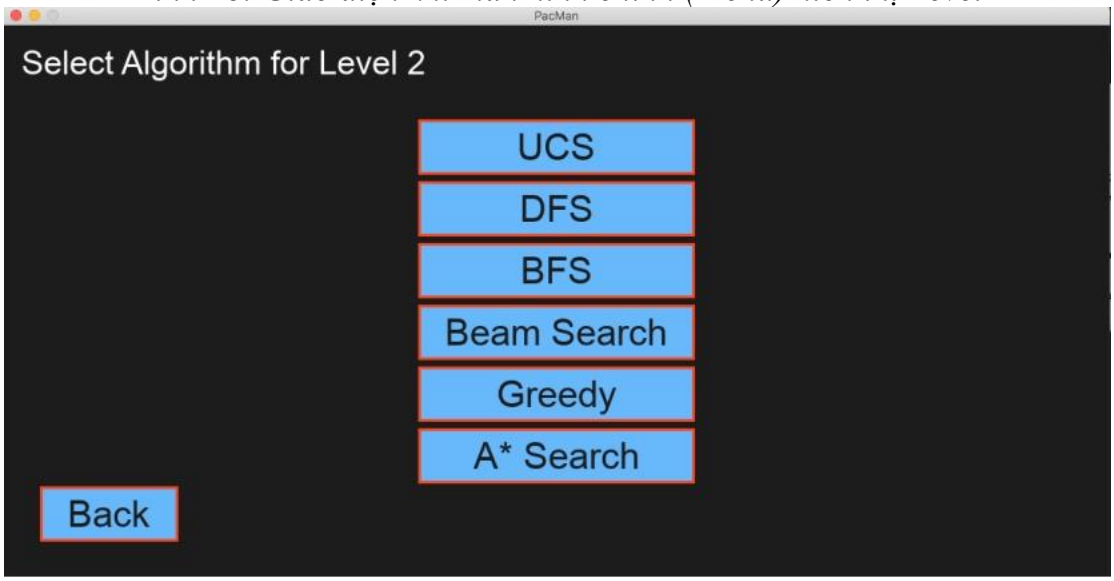
Giao diện khởi đầu cho phép người dùng lựa chọn:

- ✓ **Level:** gồm 4 cấp độ, tương ứng với từng nhóm thuật toán AI được sử dụng (Level 1: BFS, DFS; Level 4: Minimax, Alpha-Beta…).
- ✓ **Map:** chọn bản đồ từ danh sách bản đồ .txt có sẵn.

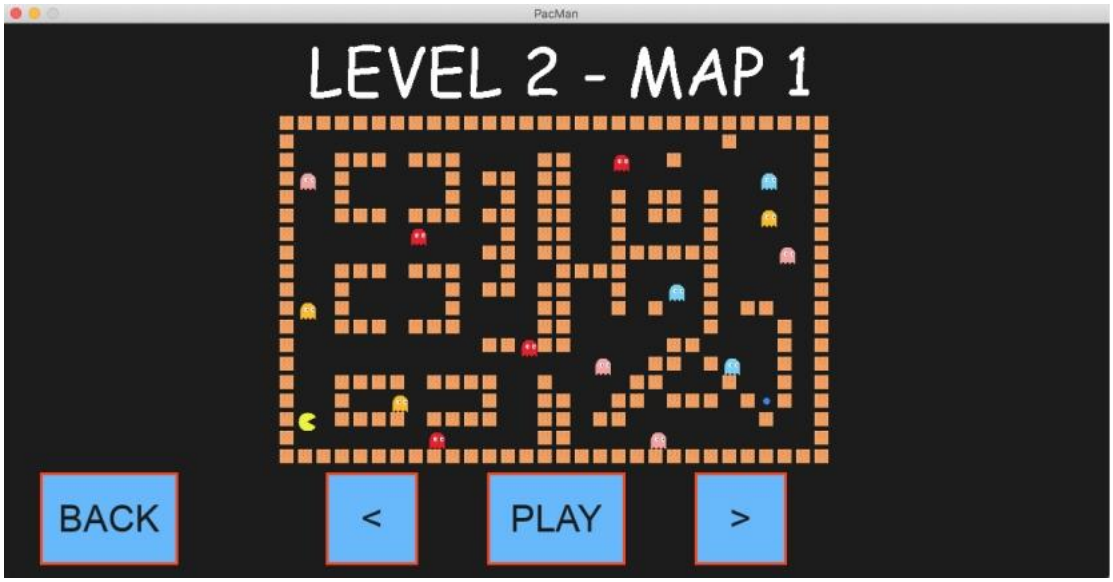
Người dùng tương tác qua chuột để lựa chọn và nhấn Start để bắt đầu game (showMenu() → startGame()).

Font chữ được thiết kế riêng, sử dụng Jomplang-6YJ3o.ttf với màu sắc dễ nhìn và phân biệt rõ giữa các mục chọn.

Hình 18: Giao diện khi màn hình chính (Menu) hiển thị Level



Hình 19: Giao diện chọn thuật toán cho level



Hình 20: Giao diện chọn map

❖ **Giao diện màn hình trò chơi**

Mê cung (map): được hiển thị dưới dạng lưới vuông (grid) gồm các ô:

- ✓ WALL (1) – tường: vẽ bằng đối tượng Wall với màu xanh biển.
- ✓ FOOD (2) – thức ăn: vẽ bằng đối tượng Food với màu vàng.
- ✓ MONSTER (3) – ghost: di chuyển theo chiến lược đã lập trình.
- ✓ PACMAN (9) – nhân vật chính, tự động di chuyển theo thuật toán đã chọn.

Pacman được vẽ bằng hình động có góc xoay theo hướng di chuyển (change_direction_PacMan()), tự động thực hiện tìm đường trong bản đồ.

Ghost được sinh động hóa bằng ảnh chuyển động (IMAGE_GHOST[]) và di chuyển ngẫu nhiên hoặc theo A* tùy Level (generate_Ghost_new_position()).

Hình 21: Giao diện màn hình trò chơi

❖ **Hộp thông tin & cập nhật trạng thái**

Score: hiển thị ở góc trên trái màn hình, được cập nhật liên tục theo số thức ăn thu thập được.

Hiển thị hoạt động thực tế:

- ✓ Khi Pacman ăn thức ăn: điểm số tăng + kế hoạch được reset.
- ✓ Khi ghost di chuyển: có hiệu ứng từng bước, tái tạo lại ô cũ nếu là thức ăn.
- ✓ Khi va chạm Pacman – ghost: dừng game và chuyển sang màn hình kết thúc.

Tất cả các đối tượng được vẽ lại qua hàm Draw(screen) và cập nhật liên tục thông qua pygame.display.flip() với tốc độ khung hình ổn định (clock.tick(FPS)).

Hình 22: Thanh Score

❖ **Giao diện kết thúc (Win/Lose)**

Sau khi Pacman ăn hết thức ăn hoặc bị ghost bắt, giao diện sẽ tự động chuyển sang **màn hình thông báo:**

- ✓ **Thắng (win):** hiển thị hình ảnh "You Win" kèm điểm số cuối cùng.
- ✓ **Thua (lose):** hiển thị ảnh nền gameover (gameover_bg1.png).

Hai nút chức năng:

- ✓ CONTINUE: quay lại menu để chọn lại Level và Map.
- ✓ QUIT: thoát game (sys.exit()).

Giao diện tuy tối giản nhưng thể hiện rõ luồng hoạt động của thuật toán và kết quả game. Đây là công cụ trực quan giúp người dùng – đặc biệt là người học – theo dõi được hành vi của từng thuật toán AI một cách sinh động và dễ hiểu.

Hình 23: Giao diện kết thúc

4.4. Link Github

https://github.com/ThanhSangLouis/Pacman_AI_Game

PHẦN 5. KẾT LUẬN

5.1. Đánh giá những kết quả đã thực hiện được

Qua quá trình thực hiện đồ án, nhóm em đã xây dựng thành công một trò chơi Pacman có khả năng tự điều khiển bằng các thuật toán tìm kiếm AI. Khác với các phiên bản truyền thống phải điều khiển bằng tay, Pacman của nhóm em có thể tự tính toán đường đi đến các hạt thức ăn sao cho hiệu quả nhất, đồng thời tránh né quái vật trong những màn chơi có độ khó cao.

Cụ thể, nhóm đã triển khai 14 thuật toán AI khác nhau, chia thành nhiều nhóm: tìm kiếm không có thông tin (BFS, DFS, UCS), có thông tin (A*, Greedy), tìm kiếm cục bộ (Local Search, Hill Climbing, Simulated Annealing, Beam Search), học tăng cường (Q-Learning), tìm kiếm đối kháng (Minimax, Alpha-Beta Pruning), và tìm kiếm trong môi trường bất định (AND-OR Tree Search), tìm kiếm có ràng buộc (Backtracking, Backtracking + AC-3). Các thuật toán được phân phối theo từng cấp độ chơi từ dễ đến khó, nhằm so sánh cách mà mỗi phương pháp xử lý tình huống thực tế trong game.

Ngoài phần AI, nhóm còn xây dựng giao diện đồ họa hoàn chỉnh bằng thư viện Pygame, cho phép chọn bản đồ, cấp độ, thuật toán, và quan sát Pacman tự chơi qua từng bước. Ghost được lập trình vừa ngẫu nhiên vừa thông minh hơn ở level cao, khiến trò chơi có tính thử thách rõ rệt. Nhờ đó, sản phẩm không chỉ đáp ứng yêu cầu về mặt kỹ thuật mà còn có tính tương tác và minh họa sinh động cho kiến thức đã học.

5.2. Định hướng phát triển

Dù dự án đã hoàn thành tương đối đầy đủ các chức năng đề ra, nhưng nhóm vẫn nhận thấy còn nhiều hướng để mở rộng và cải thiện:

- **Bổ sung thêm thuật toán**: Dự kiến sẽ thử nghiệm thêm các phương pháp mới như Genetic Algorithm, No Observation Search hoặc học tăng cường nâng cao như Deep Q-Learning để Pacman xử lý được các bản đồ lớn hơn và thay đổi linh hoạt hơn.
- **Cải thiện logic né Ghost**: Hiện Pacman tránh Ghost tương đối ổn nhưng chưa tối ưu. Có thể nâng cấp bằng cách dự đoán đường đi của Ghost hoặc sử dụng thuật toán nhiều agent đồng thời.
- **Mở rộng khả năng tùy chỉnh bản đồ**: Thêm chức năng cho phép người dùng tự tạo bản đồ riêng, đặt vị trí thức ăn và Ghost tùy ý để thử nghiệm các chiến lược khác nhau.
- **Tối ưu hiệu năng thuật toán**: Một số thuật toán như Minimax hoặc Alpha-Beta đôi khi gây trễ do tính toán nhiều, nhóm dự định thêm các kỹ thuật cache, cắt tỉa tốt hơn để cải thiện tốc độ chạy.

- **Phát triển thành công cụ học AI trực quan**: Với giao diện hiện tại, dự án có thể được phát triển tiếp để dùng trong giảng dạy, nơi sinh viên vừa quan sát vừa thay đổi thuật toán và thấy ngay kết quả.

Tổng thể, đồ án không chỉ giúp nhóm tiếp cận sâu hơn với các thuật toán AI mà còn rèn luyện kỹ năng lập trình, xử lý thuật toán trong môi trường thực tế, góp phần làm rõ hơn vai trò của trí tuệ nhân tạo trong lập trình trò chơi.

TÀI LIỆU THAM KHẢO

[1] GeeksforGeeks. (2025, May 3). *Python tutorial | Learn Python programming language*. GeeksforGeeks. https://www.geeksforgeeks.org/python-programming-language-tutorial/?ref=shm

[2] Wikipedia contributors. (2025, May 9). Visual Studio Code. Wikipedia. https://en.wikipedia.org/wiki/Visual_Studio_Code

[3] Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.

[4] UC Berkeley AI. (2025, May 13). Project 2: Multi-Agent Pacman. UC Berkeley Artificial Intelligence. https://ai.berkeley.edu/multiagent.html

[5] nxhawk. (2020, September 19). Pacman-AI. GitHub. https://github.com/nxhawk/Pacman-AI