Intro to AI,
Autumn, 2025

Group 8's Presentation:

The Pac-Man Game

Nội dung

1. Giới Thiệu
2. Bài Toán Pac-Man
   - 2.1. Xác định vấn đề
   - 2.2. Phương Pháp
   - 2.3. Đánh Giá
3. Kết Luận
4. Thảo Luận

1. Giới thiệu về Pac-Man

Pac-Man, ra mắt lần đầu vào năm 1980, là một trong những trò chơi arcade **mang tính biểu tượng** nhất trong lịch sử.

Trí tuệ nhân tạo (AI) đóng vai trò quan trọng trong việc tạo nên trải nghiệm chơi game hấp dẫn. Một hệ thống AI được thiết kế tốt giúp trò chơi duy trì sự cân bằng giữa thử thách và giải trí.

Thuật toán **A*** (A-star) được lựa chọn trong nghiên cứu này vì tính hiệu quả và khả năng tìm đường tối ưu. A* sử dụng hàm đánh giá f(n)=g(n)+h(n) để cân bằng giữa chi phí thực tế và chi phí ước lượng đến mục tiêu.

2. Bài toán Pac-Man

2.1 Xác định vấn đề

- Pac-Man cần **di chuyển trong lưới 5x5** để ăn hết các "dot" (viên thức ăn).
- **Mục tiêu:** Tìm **đường đi ngắn nhất** giúp Pac-Man thu thập toàn bộ dot với **chi phí di chuyển nhỏ nhất**.
- Môi trường bao gồm:
    - Ô trống (Pac-Man có thể đi qua)
    - Vị trí bắt đầu của Pac-Man
    - Các dot cần ăn
    - Chướng ngại vật (#) mà Pac-Man **không thể vượt qua**
- Hướng di chuyển: Lên/Xuống/Phải/Trái (Không thể đi chéo)
- Mỗi bước có **chi phí = 1**

2. Bài toán Pac-Man

2.1 Xác định vấn đề

Phương pháp:
Thuật toán A* Search

- Kết hợp giữa:
  - g(n): Chi phí thực tế đã đi
  - h(n): ước lượng chi phí còn lại
- Heuristic: Tổng khoảng cách Manhattan đến các dot chưa ăn
- Phù hợp cho tìm đường trong lưới 4 hướng di chuyển

Công thức:

$$f(n)=g(n)+h(n)$$

2. Bài toán Pac-Man

2.1 Xác định vấn đề

Phương pháp:
Thuật toán A* Search

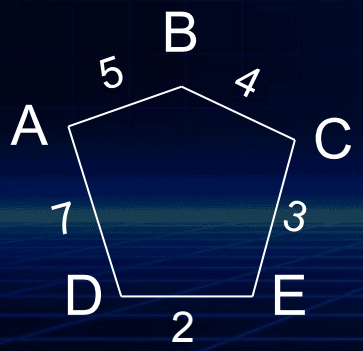
- **Ứng dụng:** Google Maps, Grab, hệ thống GPS trong xe hơi.
- **Mục tiêu:** Tìm đường đi ngắn nhất từ điểm A → B trên bản đồ thật.
- **Hoạt động:**
  - `g(n)`: quãng đường thực đã đi (dựa theo dữ liệu giao thông).
  - `h(n)`: khoảng cách ước lượng đến đích (thường là **đường thẳng Euclidean** hoặc **Manhattan**).
- **Ưu điểm:** A* giúp **tìm đường nhanh hơn Dijkstra** vì bỏ qua hướng không cần thiết.
- 🗺️ *Ví dụ:* Khi bạn mở Google Maps và chọn "Đường đi nhanh nhất", hệ thống thật ra đang dùng **A*** hoặc biến thể của nó (như *A với heuristic địa lý**).

2. Bài toán Pac-Man

2.1 Xác định vấn đề

Ví dụ về A*:

Nhiệm vụ: Tìm tuyến đường tối ưu từ vị trí A (xuất phát) → vị trí C (đích).

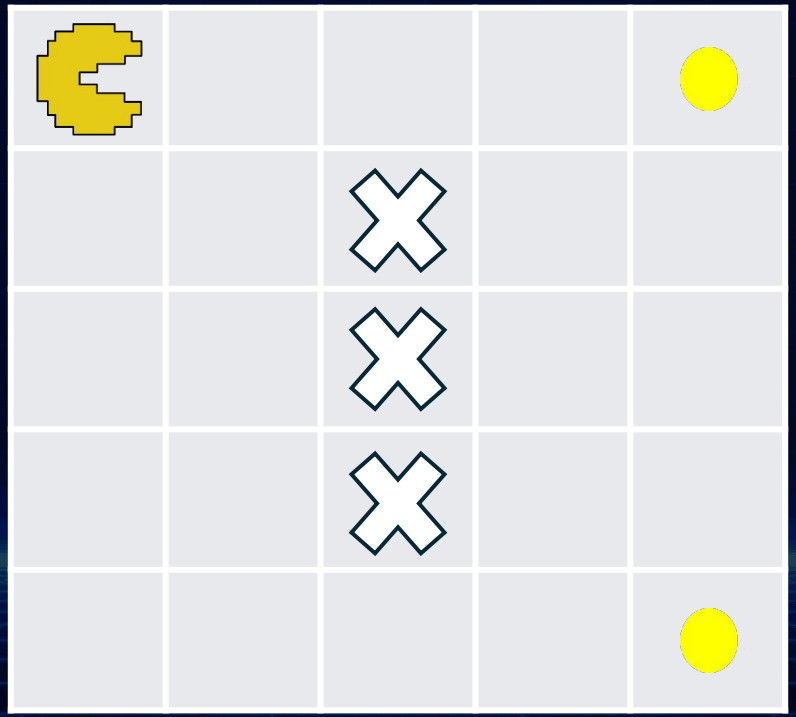


Heuristic:
Giả sử ta dùng "đường thẳng" làm ước lượng:
h(A)=7, h(B)=4, h(D)=5, h(E)=2, h(C)=0.

Diễn giải:
Mở A:
A (f=0) → chuyển A sang Closed, check các láng giềng B,D
Xem B: g(B) = 0 + 5 = 5,h(B) = 0,f(B) = 5 → thêm B vào Open
Xem D: g(D) = 0 + 7 = 7,h(D) = 0,f(D) = 7 → thêm D vào Open
→ Chọn B trước vì f(B) nhỏ hơn.
Ta có láng giềng của B là C:
Xem C: g(C)=g(B) +4 = 5 + 4 = 9, h(C) = 0, f(C)=9=9 → Thêm C vào Open → kết thúc tại C (goal)
A* kết thúc ở C với đường đi A → B → C, tổng chi phí = 9 (ngắn nhất).

2. Bài toán Pac-Man

Ví dụ bản đồ lưới 5x5:



2. Bài toán Pac-Man

2.2 Phương pháp

Phương pháp:

- A* là thuật toán tìm đường tối ưu có hướng dẫn (heuristic), thường dùng cho bài toán đường đi ngắn nhất.
- Thuật toán duy trì một tập mở (open set) các ô cần khám phá, sắp xếp theo giá trị f(n) trong hàng đợi ưu tiên (priority queue).

Công thức:

$$f(n)=g(n)+h(n)$$

- **g(n):** Chi phí đã đi từ điểm xuất phát → n
- **h(n):** Ước lượng chi phí còn lại từ n → đích (heuristic)
- **f(n):** Tổng chi phí dự đoán → dùng để chọn ô tiếp theo

2. Bài toán Pac-Man

Tính toán các hàm f(n), g(n), h(n) cho bài toán Pacman

g(n): Chi phí thực tế từ vị trí bắt đầu → ô hiện tại
→ Mỗi bước di chuyển có **cost = 1**
→ Ví dụ: (0,0) → (1,0) ⇒ g = 1
h(n): Ước lượng chi phí còn lại đến Dot
→ Dùng **khoảng cách Manhattan**:

$$h = | x_1 - x_2 | + | y_1 - y_2 |$$

→Heuristic admissible (không vượt chi phí thực)
f(n) = g(n) + h(n):
→ Tổng chi phí dự đoán
→ A* luôn mở rộng **ô có f nhỏ nhất trước**

Ví dụ:
Pac-Man tại (1,0), Dot ở (2,3):
→ g = 1, h = 4 ⇒ f = 5

2. Bài toán Pac-Man

Mở rộng A* cho Bài toán Nhiều Mục Tiêu (A* for Multiple Goals)

Cách 1: Brute Force / Quy Hoạch Động
Duyệt **mọi thứ tự ghé thăm các Dot** (hoán vị).

- Dùng **A*** hoặc **BFS/UCS** để tính tổng chi phí từng hành trình.
- Chọn hành trình ngắn nhất → **kết quả tối ưu tuyệt đối**.
- Nhược điểm: **Độ phức tạp O(m!)**, chỉ phù hợp với **ít Dot (như 5x5)**.

Cách 2: A* mở rộng

- **Trạng thái = (Vị trí Pac-Man, Tập Dot còn lại).**
- Khi Pac-Man ăn 1 Dot → loại Dot đó khỏi tập còn lại.
- **Mục tiêu:** tập Dot còn lại **rỗng** (ăn hết).
- A* tìm đường tối ưu trong **không gian trạng thái mở rộng**.
- Cần xây dựng **hàm heuristic phù hợp** cho nhiều mục tiêu.

2. Bài toán Pac-Man

Chướng ngại vật và xử lý trong thuật toán

- Pac-Man **không thể đi qua ô tường** → giới hạn không gian trạng thái.
- Khi mở rộng bước đi, **bỏ qua** các ô:
  - Nằm ngoài bản đồ
  - Là tường (không có neighbor hợp lệ)
- Tường khiến **đường đi thực tế dài hơn**, nhưng **heuristic Manhattan** vẫn **admissible** (không đánh giá thấp chi phí).
- Nếu **Dot bị tường bao kín**, A* **không tìm được trạng thái mục tiêu** → thuật toán **kết luận failure**.
- Cần xử lý và **thông báo lỗi** khi không thể thu thập đủ Dot.

2. Bài toán Pac-Man

Chi tiết kỹ thuật hoạt động của thuật toán

Biểu diễn bản đồ & trạng thái:

- Mê cung 5x5 biểu diễn dưới dạng **ma trận/lưới ô**
- Mỗi ô: **trống**, **có Dot**, hoặc **tường (#)**
- **Trạng thái A*** gồm:
- Vị trí Pacman hiện tại (x, y)
- Tập các Dot **chưa thu thập** (dạng tập hợp hoặc bitmask)
- Ví dụ:
- Trạng thái đầu: Pacman tại (0,0), còn tất cả Dot
- Trạng thái đích: không còn Dot nào

2. Bài toán Pac-Man

Chi tiết kỹ thuật hoạt động của thuật toán

Cấu trúc dữ liệu tìm kiếm:

- **Open list**: Hàng đợi ưu tiên theo:

  f = g + h

  → Lưu các trạng thái "biên" đang chờ mở rộng

- **Closed list**: Lưu các trạng thái đã duyệt
  → Tránh lặp lại, mỗi trạng thái gồm **vị trí + tập Dot còn lại**

2. Bài toán Pac-Man

Chi tiết kỹ thuật hoạt động của thuật toán

Quy trình hoạt động

1.Khởi tạo

- Đưa trạng thái đầu vào Open list (g=0, f=g+h)

2.Lặp

- Lấy trạng thái có **f nhỏ nhất** từ Open
- Nếu tất cả Dot đã ăn hết → ✅ **Thành công**
- Ngược lại: mở rộng các **ô kề hợp lệ** (4 hướng)
  - Bỏ qua tường/ngoài bản đồ
  - Nếu đi vào ô có Dot → cập nhật tập Dot còn lại
- Tính lại `g, h, f` và cập nhật vào Open list

3.Kết thúc

- Nếu Open rỗng → ❌ **Không có đường đi hợp lệ**

2. Bài toán Pac-Man

Chi tiết kỹ thuật hoạt động của thuật toán

Khác biệt so với bài toán một mục tiêu

- **Trạng thái**: Gồm cả tập Dot → không gian tìm kiếm lớn hơn
- **Điều kiện dừng**: Ăn hết tất cả Dot, không chỉ tới 1 ô đích
- **Heuristic**:
  - Có thể dùng **tổng Manhattan** đến các Dot còn lại
  - Đơn giản nhưng không luôn tối ưu
- **Tính tối ưu**: Giữ được nếu heuristic **admissible**

Code

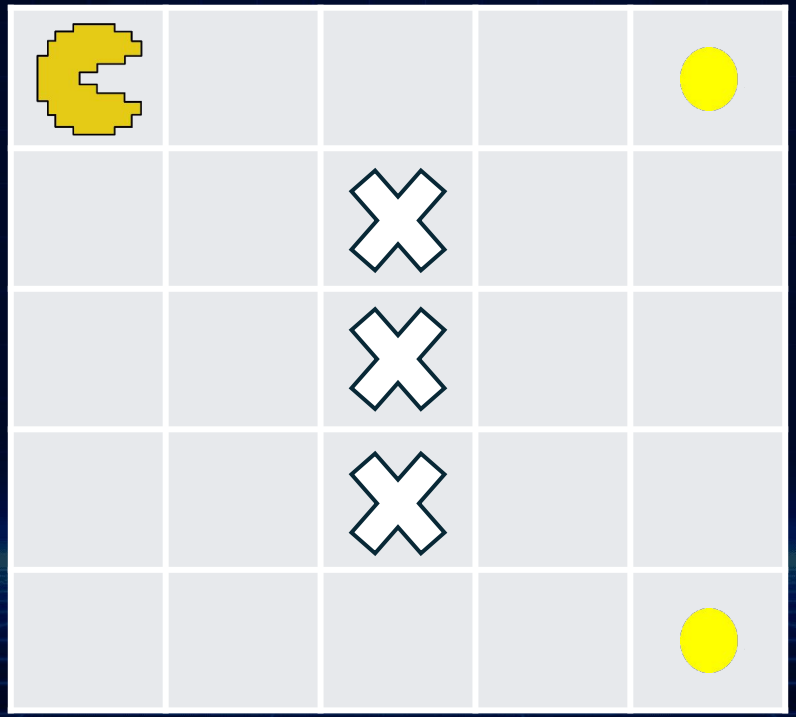
```
function AStar_MultiGoal(start_position, goal_set, grid):
    # Khởi tạo Open list (ưu tiên theo f) và Closed set
    Open := priority_queue()
    Closed := empty set


    # Hàm heuristic h(u) ước lượng chi phí từ trạng thái u đến khi ăn hết goal
còn lại
    function heuristic(state):
        (pos, remainingGoals) := state
        # Ví dụ: dùng heuristic Manhattan tổng đến các mục tiêu còn lại
        h_sum = 0
        for each goal in remainingGoals:
            h_sum += ManhattanDistance(pos, goal)
        return h_sum


    # Tạo trạng thái bắt đầu
    start_state := (start_position, goal_set)
    g[start_state] := 0
    h[start_state] := heuristic(start_state)
    f[start_state] := g[start_state] + h[start_state]
    parent[start_state] := NULL


    Open.push(start_state, f[start_state])


    # Vòng lặp tìm kiếm chính
    while Open is not empty:
        current_state := Open.pop()  # lấy trạng thái có f nhỏ nhất
        if current_state in Closed:
            continue  # bỏ qua nếu đã xét (tránh trùng lặp)
        add Closed <- current_state
```

Code

```
(current_pos, remainingGoals) := current_state
        # Kiểm tra mục tiêu: đã ăn hết tất cả Dot?
        if remainingGoals is empty:
            return reconstruct_path(current_state, parent)  # thành công, trả về đường đi


        # Mở rộng các trạng thái kế tiếp từ current_state
        for each neighbor_pos in get_neighbors(current_pos) do:
            if grid[neighbor_pos] == WALL:
                continue  # bỏ qua hướng đi vào tường


            # Xác định tập mục tiêu còn lại khi đi đến neighbor
            newRemaining := remainingGoals
            if neighbor_pos in remainingGoals:
                # nếu ô hàng xóm có một Dot chưa ăn
                newRemaining := remainingGoals \ { neighbor_pos }  # bỏ nó khỏi tập mục tiêu


            new_state := (neighbor_pos, newRemaining)
            tentative_g := g[current_state] + 1  # chi phí đến neighbor = chi phí hiện tại + 1 bước


            # Nếu tìm được đường đi mới đến new_state ngắn hơn đường đã biết trước đó (nếu có)
            if (new_state not in Closed) and ( (new_state not in g) or (tentative_g < g[new_state]) ):
                parent[new_state] := current_state
                g[new_state] := tentative_g
                h[new_state] := heuristic(new_state)
                f[new_state] := g[new_state] + h[new_state]
                Open.push(new_state, f[new_state])
                # (Nếu new_state đã có trong Open với chi phí cao hơn, cần cập nhật lại priority -
                # tùy cấu trúc hàng đợi, có thể cần thủ tục decrease-key)


    # Nếu Open rỗng mà không tìm được mục tiêu
    return failure  # Không tồn tại đường đi ăn hết các Dot
```

2. Bài toán Pac-Man

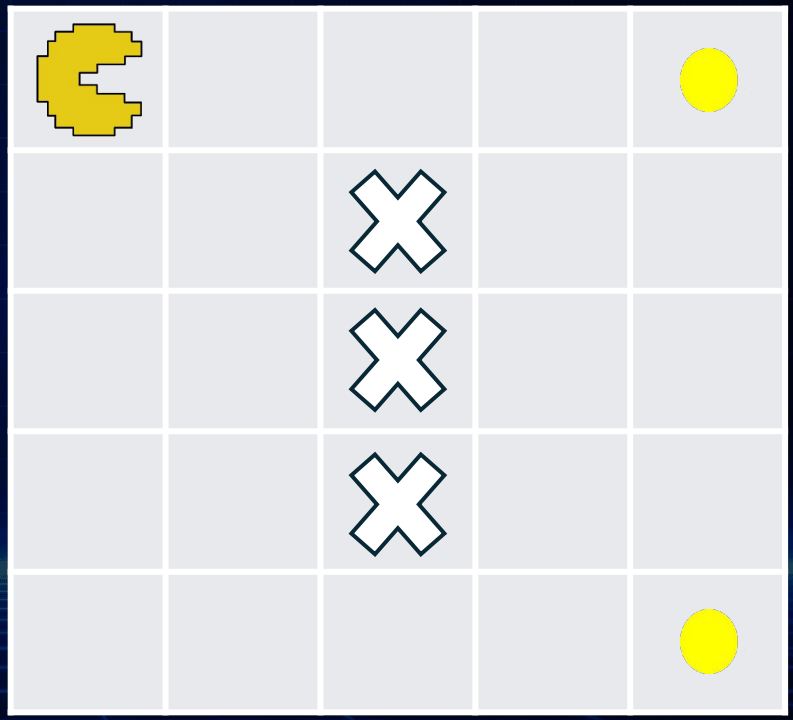
Ví dụ bản đồ lưới 5x5:



2. Bài toán Pac-Man

Bước 0 – Khởi tạo

- Start: (0,0), mục tiêu {D1, D2}
- g=0, h=4+8=12, f=12
- **Open = {(0,0;{D1,D2}), f=12}, Closed = ∅**

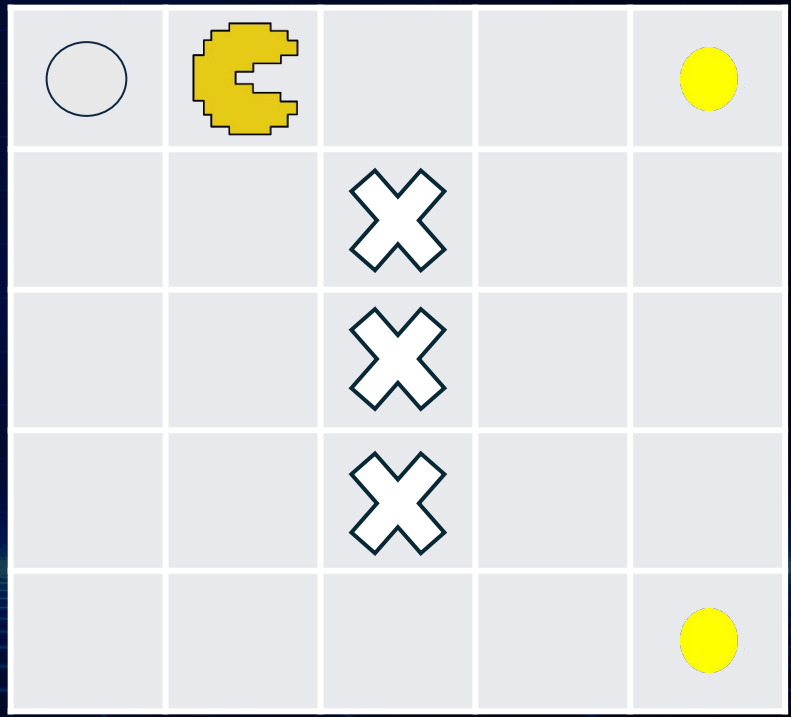


2. Bài toán Pac-Man

Bước 1 – Mở rộng (0,0)

- Neighbor hợp lệ: (0,1), (1,0)
- (0,1): f=11 → tốt hơn
- (1,0): f=13

  → **Open = {(0,1):11, (1,0):13}**

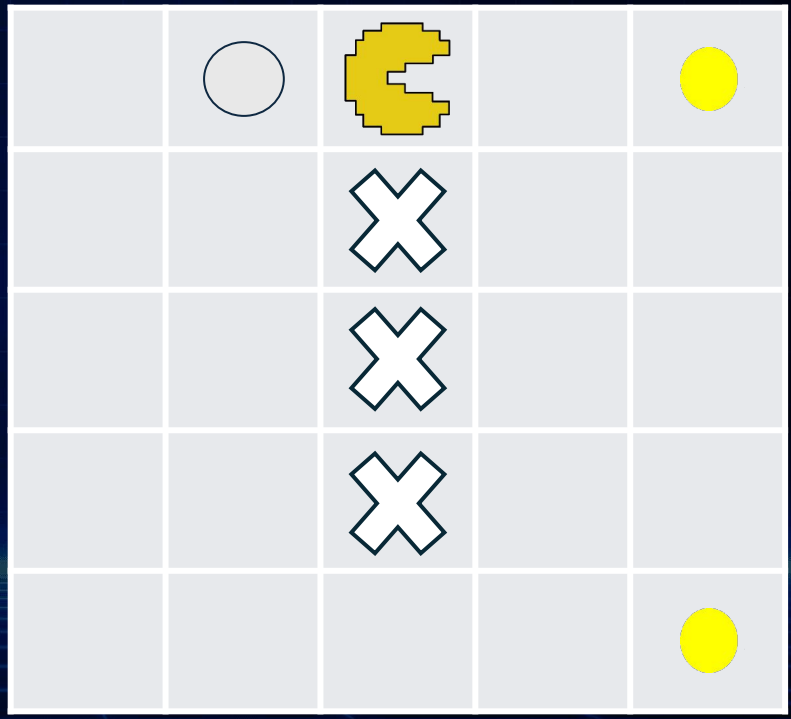


2. Bài toán Pac-Man

Bước 2 – Mở rộng (0,1)

- Neighbor: (0,2), (1,1)
- (0,2): f=10
- (1,1): f=12

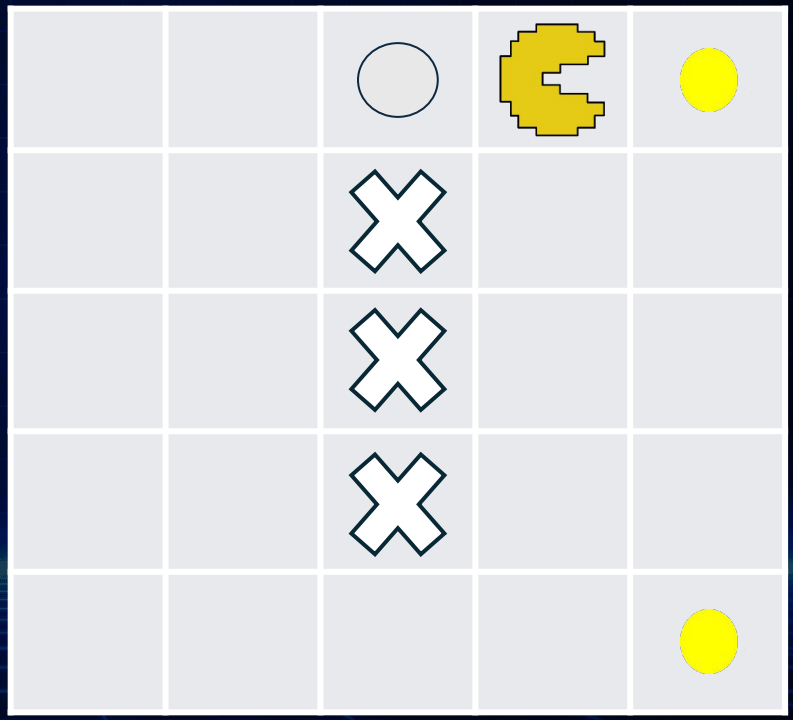
  → **Open = {(0,2):10, (1,1):12, (1,0):13}**



2. Bài toán Pac-Man

Bước 3 – Mở rộng (0,2)

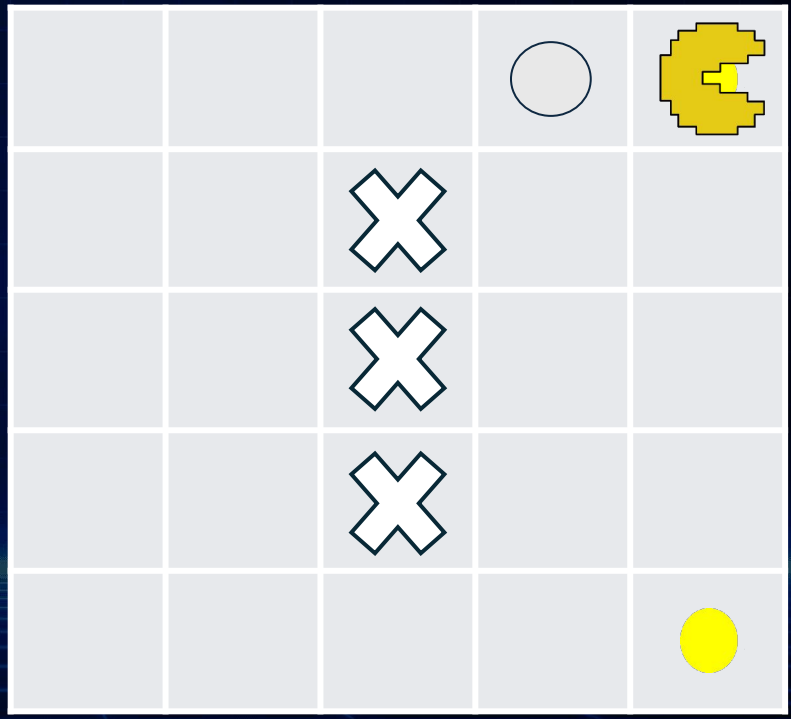
- Neighbor: (0,3) (vì (1,2) là tường)
- (0,3): f=9
  → **Open = {(0,3):9, (1,1):12, (1,0):13}**



2. Bài toán Pac-Man

Bước 4 – Mở rộng (0,3)

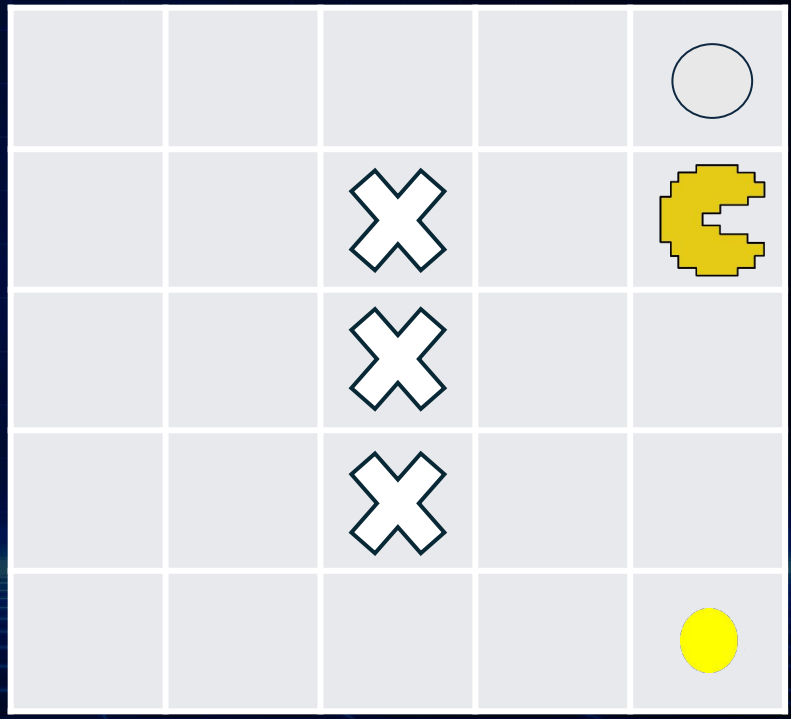
- Neighbor: (0,4) (Dot D1), (1,3)
- (0,4; còn D2): f=8
- (1,3; chưa ăn D1): f=10
  → **Open = {(0,4;D2):8, (1,3):10, (1,1):12, (1,0):13}**
-



2. Bài toán Pac-Man

Bước 5 – Mở rộng (0,4; còn D2)

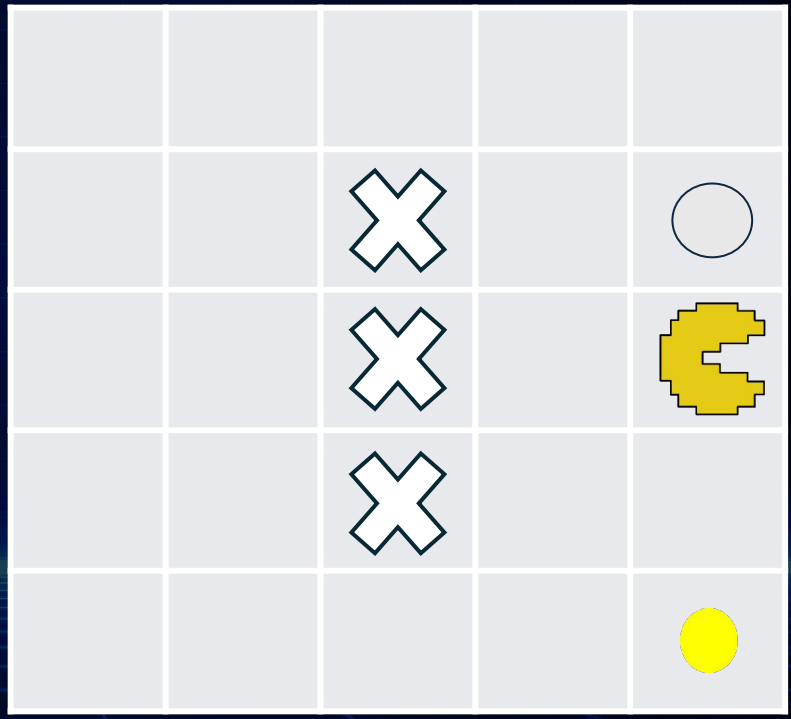
- Neighbor: (1,4): f=8
  → **Open = {(1,4;D2):8, (1,3):10, (1,1):12, (1,0):13}**



2. Bài toán Pac-Man

Bước 6 – Mở rộng (1,4;D2)

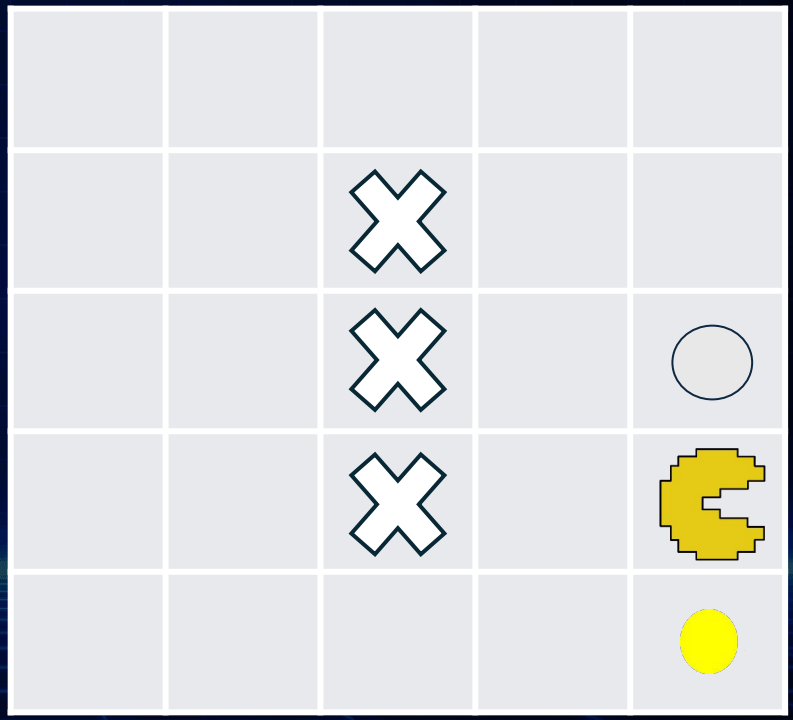
- Neighbor: (2,4): f=8, (1,3): f=10
  → **Open = {(2,4;D2):8, (1,3):10, (1,1):12, (1,0):13}**



2. Bài toán Pac-Man

Bước 7 – Mở rộng (2,4;D2)

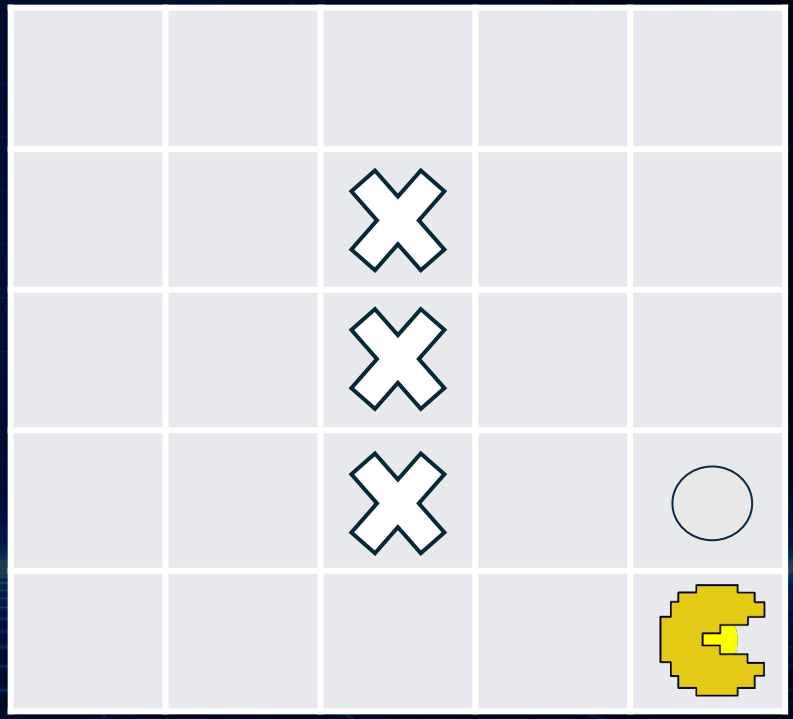
- Neighbor: (3,4): f=8
  → **Open = {(3,4;D2):8, (1,3):10, (1,1):12, (1,0):13}**



2. Bài toán Pac-Man

Bước 8 – Mở rộng (3,4;D2)

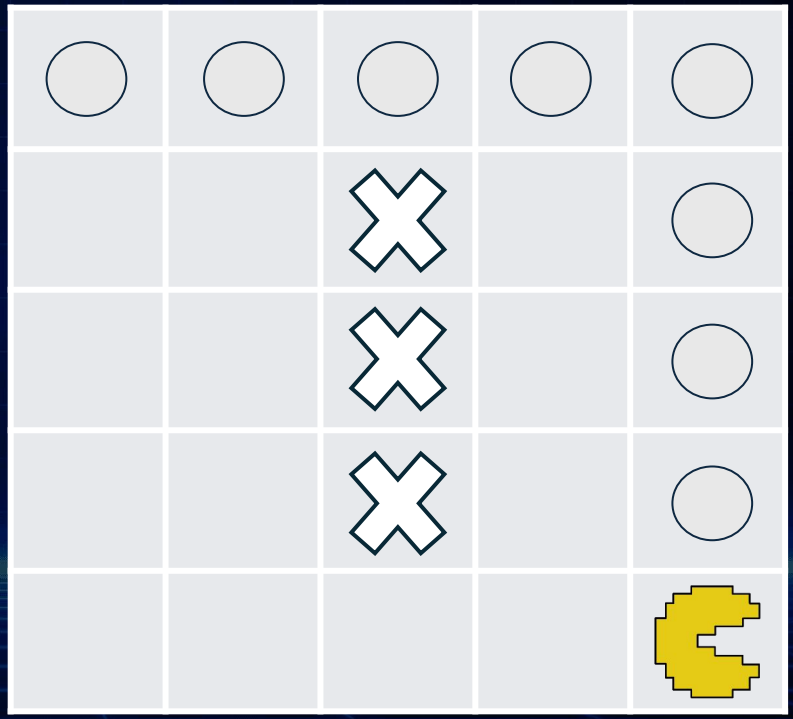
- Neighbor: (4,4) là **Dot D2** → đạt đích
- g=8, h=0, f=8
  Thuật toán kết thúc: Pacman đã ăn hết Dots.



2. Bài toán Pac-Man

Kết quả

- **Đường đi:** (0,0) → (0,1) → (0,2) → (0,3) → (0,4) → (1,4) → (2,4) → (3,4) → (4,4)
- **Tổng chi phí (g)** = 8
  A* tìm ra đường đi tối ưu, tránh tường, thỏa điều kiện admissible heuristic.



2. Bài toán Pac-Man

Thuật toán mở rộng (BFS)

Cách hoạt động của BFS trong bài toán Pac-Man

- **Mỗi ô** trên bản đồ là **một đỉnh** trong đồ thị.
- **Các cạnh** nối giữa hai ô **liền kề nhau theo 4 hướng** (trên, dưới, trái, phải), **trừ khi gặp tường** .
- BFS bắt đầu từ **vị trí Pac-Man** .
- Nó dùng **hàng đợi** để lưu các ô sẽ được khám phá kế tiếp.
  Mỗi khi duyệt một ô, BFS:
  1. Đánh dấu ô đó là **đã thăm** .
  2. Thêm **các ô kề hợp lệ (không phải tường, chưa thăm)** vào hàng đợi.
  3. Nếu gặp **một viên thức ăn** thì dừng lại — vì BFS đảm bảo đây là **đường đi ngắn nhất**.

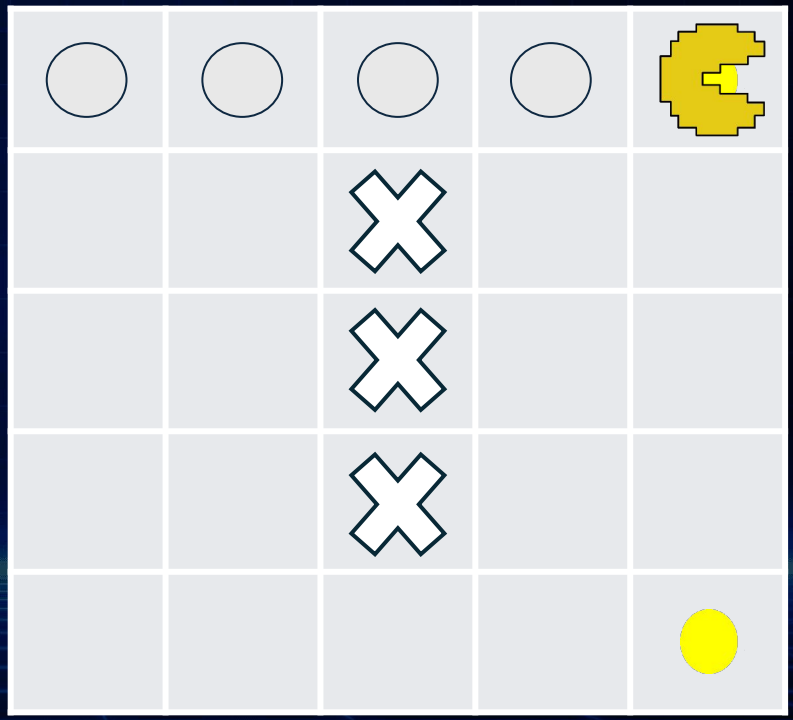
2. Bài toán Pac-Man

Thuật toán mở rộng (BFS)

Áp dụng BFS để tìm đường đi ngắn nhất ăn cả hai viên

Bước 1: Tìm viên thức ăn gần nhất bằng BFS

Từ (0,0) đến (0,4):
(0,0) → (0,1) → (0,2) → (0,3) → (0,4)
→ Pac-Man ăn **viên thứ nhất**



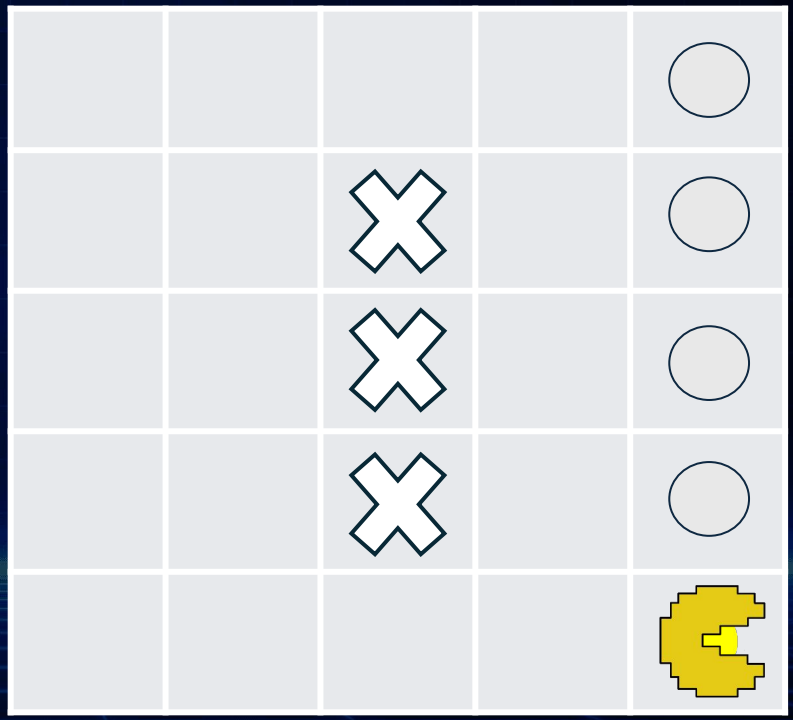
2. Bài toán Pac-Man

Thuật toán mở rộng (BFS)

Áp dụng BFS để tìm đường đi ngắn nhất ăn cả hai viên

Bước 2: Từ vị trí hiện tại (0,4), tìm đường ngắn nhất đến viên còn lại (4,4)

Do cột tường chắn giữa bản đồ, Pac-Man phải **đi vòng xuống dưới qua cột phải**:
(0,4) → (1,4) → (2,4) → (3,4) → (4,4)
→ Pac-Man ăn **viên thứ hai**



2. Bài toán Pac-Man

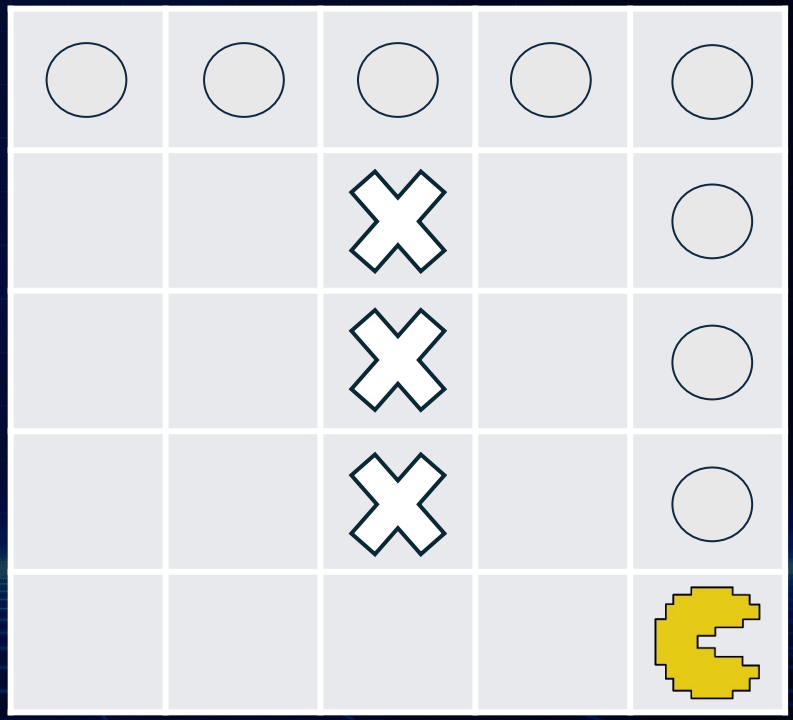
Thuật toán mở rộng (BFS)

Áp dụng BFS để tìm đường đi ngắn nhất ăn cả hai viên

Tổng đường đi ngắn nhất để ăn cả hai viên:

(0,0) → (0,1) → (0,2) → (0,3) → (0,4) → (1,4) → (2,4) → (3,4) → (4,4)

Tổng cộng 8 bước di chuyển.



2. Bài toán Pac-Man

2.3 Đánh giá

So sánh BFS và A* (Manhattan heuristic)

Tiêu chí	BFS	A* (Manhattan)
Độ dài hành trình	9 ô (8 bước) – tối ưu	9 ô (8 bước) – tối ưu
Tối ưu đường đi	Tối ưu theo từng đoạn (shortest path)	Tối ưu nếu heuristic admissible
Độ phức tạp thời gian	O(b^d) – mở rộng toàn bộ	O(b^d') – ít hơn nhờ heuristic
Bộ nhớ sử dụng	Cao (queue + visited)	Thấp hơn, có định hướng
Số nút mở rộng (5×5)	~23 ô	~10 ô
Tốc độ thực thi	Chậm hơn (~2.3× nhiều nút)	Nhanh hơn, tiết kiệm tài nguyên
Ưu điểm	Đơn giản, chính xác	Hiệu quả, có định hướng
Nhược điểm	Tốn thời gian/bộ nhớ	Cần chọn heuristic phù hợp

2. Bài toán Pac-Man

2.3 Đánh giá

Heuristic trong A*

- **Heuristic (h)**: Ước lượng chi phí còn lại đến mục tiêu.
  → Giúp **A*** ưu tiên hướng "hứa hẹn nhất", không duyệt toàn bộ bản đồ.

- **Công thức:**

$$f(n) = g(n) + h(n)$$

  - g(n): chi phí thật đã đi
  - h(n): chi phí ước lượng còn lại
  - f(n): tổng chi phí dùng để sắp hàng ưu tiên

2. Bài toán Pac-Man

2.3 Đánh giá

Heuristic được sử dụng:

- **Tổng khoảng cách Manhattan** từ Pac-Man đến **tất cả các Dot chưa ăn**
- **Ý nghĩa:** Ước lượng tổng số bước cần đi để ăn hết Dot
- **Ưu điểm:**
  - Dễ tính toán
  - Có định hướng tốt, giúp giảm trạng thái cần duyệt
- **Nhược điểm:**
  - Có thể **đánh giá cao hơn thực tế**
  - Không hoàn toàn admissible, nhưng **vẫn hiệu quả trên bản đồ nhỏ**

2. Bài toán Pac-Man

2.3 Đánh giá

Thử nghiệm & Đánh giá

Thiết lập:

- Bản đồ 5×5, không có tường
- Start: (0,0)
- Dots: (4,0), (2,2), (4,4)
- Heuristic: tổng Manhattan
- Baseline: BFS (đường đi tối ưu tuyệt đối)

Kết quả:

- A* tìm đúng đường đi ăn hết các Dot
- Chi phí trùng BFS → thuật toán đúng đắn
- Thời gian xử lý: **0.0005s – 0.0011s**
  → Nhanh hơn BFS, do heuristic giảm số trạng thái mở rộng

Bản đồ	Start	Dots	Số bước (A*)	Số bước (BFS)	Thời gian A* (s)	Nhận xét
Test 1	(0,0)	{(4,0), (2,2), (4,4)}	12	12	0.0009	A* tìm đường chính xác, nhanh hơn BFS
Test 2	(2,2)	{(0,0), (4,4)}	8	8	0.0005	Kết quả trùng khớp BFS
Test 3	(0,4)	{(1,1), (3,3), (4,0)}	13	13	0.0011	A* hoạt động ổn định

3. Kết Luận

Việc áp dụng thuật toán A* trong trò chơi Pac-Man cho thấy tính hiệu quả của nó trong việc tìm đường tối ưu và điều khiển di chuyển thông minh. A* duy trì được sự cân bằng giữa tốc độ xử lý và độ chính xác, giúp các nhân vật tìm được lộ trình ngắn nhất trong thời gian thực. Điều này chứng minh rằng A* là lựa chọn phù hợp cho các hệ thống AI trong trò chơi, góp phần nâng cao yếu tố chiến lược, độ khó và trải nghiệm tổng thể của người chơi.

4. Thảo Luận

Hạn chế:
- Map nhỏ (5×5) → ít dữ liệu thử nghiệm
- Heuristic đơn giản (tổng Manhattan) → chưa tối ưu
- Môi trường tĩnh, chưa có **Ghost**

Hướng phát triển:
- Mở rộng bản đồ, thêm vật cản
- Thêm nhiều tác tử (Ghost, nhiều Pac-Man)
- Dùng heuristic nâng cao (MST, học máy)

Thank you!