Intro to AI,
Autumn, 2025

Group 8’s Presentation:
The Pac-Man Game

Content

1. Introduction to PacMan

Pac-Man, first launched in 1980, is one of the most iconic arcade games in history.

Artificial intelligence (AI) plays an important role in creating an engaging gaming experience. A well-designed AI system helps the game maintain a balance between challenge and entertainment.

The A* (A-star) algorithm was chosen in this study for its efficiency and ability to find optimal paths. A* uses the evaluation function $f(n)=g(n)+h(n)$ to balance the actual cost and the estimated cost to the goal.

2. Pac-Man Problem

2.1 Problem Definition

- Pac-Man needs to move within a **5x5 grid** to eat all the "dots" (food pellets).

- **Objective:** Find the **shortest** path that helps Pac-Man **collect all dots** with the **minimum** movement cost.

- The environment includes:
    - Empty cells (Pac-Man can pass through)
    - Pac-Man's starting position
    - Dots to be eaten
    - Obstacles (#) that Pac-Man cannot cross
- Movement directions: Up/Down/Right/Left (cannot move diagonally)
- Each step has a cost = 1

2. Pac-Man Problem

2.1 Problem Definition

2.1 Problem Definition
A* Search Algorithm

- Combination of:
  - g(n): Actual cost already traveled
  - h(n): Estimated remaining cost
- Heuristic: Total Manhattan distance to uncollected dots
- Suitable for pathfinding in a 4-direction grid

2. Pac-Man Problem

2.1 Problem Definition

Method:

A* Search Algorithm

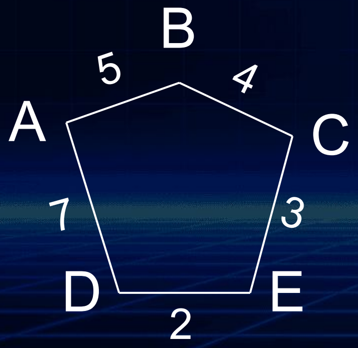
- **Applications:** Google Maps, Grab, in-car GPS systems.
- **Objective:** Find the shortest path from point A → B on a real map.
- **Operation:**
  - `g(n)`: actual distance traveled (based on traffic data).
  - `h(n)`: estimated distance to destination (usually Euclidean or Manhattan straight line).
- **Advantages:** A* helps find paths faster than Dijkstra by ignoring unnecessary directions.
- 🗺️ *Example: When you open Google Maps and select "Fastest route," the system is actually using A* or a variant of it (like A* with geographical heuristic).*

2. Pac-Man Problem

2.1 Problem Definition

Example of A*:

Task: Find the optimal route from position A (start) → position C (destination).



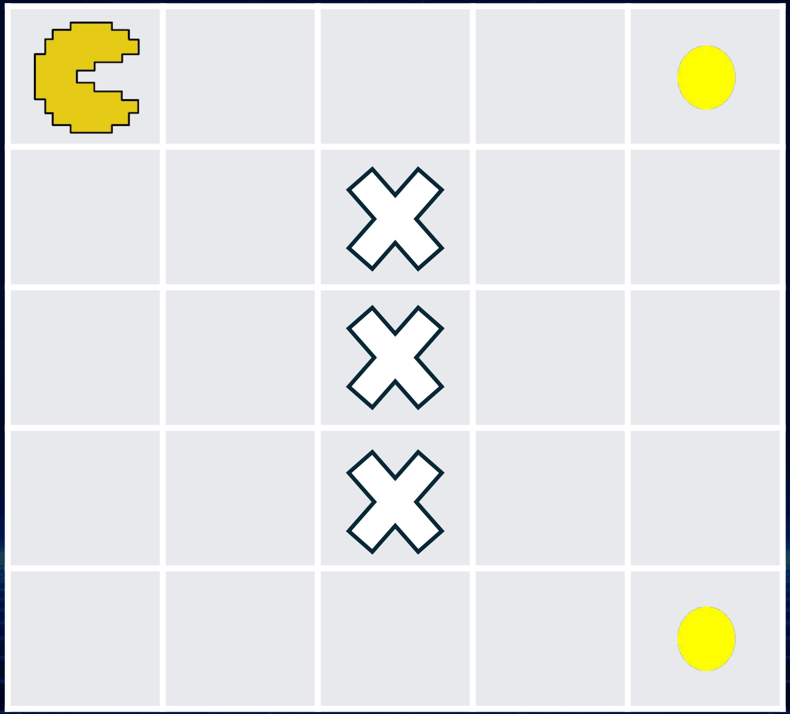
Heuristic:
Assume we use "straight line" as an estimate:
h(A)=7, h(B)=4, h(D)=5, h(E)=2, h(C)=0.

Interpretation:
Open A:
A (f=0) → move A to Closed, check neighbors B,D
Check B: g(B) = 0 + 5 = 5, h(B) = 0, f(B) = 5 → add B to Open
Check D: g(D) = 0 + 7 = 7, h(D) = 0, f(D) = 7 → add D to Open
→ Choose B first because f(B) is smaller.

B's neighbor is C:
Check C: g(C)=g(B) +4 = 5 + 4 = 9, h(C) = 0, f(C)=9=9 → Add C to Open → end at C (goal)
A* ends at C with path A → B → C, total cost = 9 (shortest).

2. Pac-Man Problem

Example of a 5x5 grid map:



2. Pac-Man Problem

2.2 Method

Method:

- A* is an optimal pathfinding algorithm with guidance (heuristic), often used for shortest path problems.

- The algorithm maintains an open set of cells to explore, sorted by f(n) value in a priority queue.

2. Pac-Man Problem

Calculate the functions f(n), g(n), h(n) for the Pac-Man problem.

g(n): The actual cost from the starting position -> to the current cell (node)

-> Each movement step has a cost = 1

-> Example: $(0,0) -> (1,0) => g = 1$

h(n): Estimated remaining cost to the Dot (Goal) -> Using Manhattan distance:

$$h = |x1 - x2| + |y1 - y2|$$

-> Heuristic admissible (does not overestimate the actual cost)
f**(n) = g(n) + h(n):**

-> Total predicted cost

-> A* always expands the cell (node) with the smallest f first

Example:

Pac-Man at (1,0), Dot at (2,3):

-> g = 1, h = 4 => f = 5

2. Pac-Man Problem

Extending A* for Multiple Goals (A* for Multiple Goals)

Method 1: Brute Force / Dynamic Programming

Iterate through all possible orders of visiting dots (permutations).

- Use A* or BFS/UCS to calculate the total cost of each path.
- Select the shortest path → absolutely optimal result.
- Disadvantage: Complexity O(m!), only suitable for a small number of dots (e.g., 5x5).

Method 2: Extended A*

- State = (Pac-Man Position, Set of Remaining Dots).
- When Pac-Man eats a dot → remove that dot from the remaining set.
- Goal: The set of remaining dots is empty (all eaten).
- A* finds the optimal path in the extended state space.
- Requires building a suitable heuristic function for multiple goals.

2. Pac-Man Problem

Obstacles and handling in the algorithm

- Pac-Man cannot pass through wall cells
  → limits the state space.
- When expanding steps, ignore cells:
  - Outside the map
  - Are walls (no valid neighbors)
- Walls make the actual path longer, but the Manhattan heuristic remains admissible (does not underestimate the cost).
- If a Dot is enclosed by walls, A* cannot find the target state → the algorithm concludes failure.
- Need to handle and report errors when not all Dots can be collected.

2. Pac-Man Problem

Technical details of the algorithm's operation

Map & state representation:

- 5x5 maze represented as a matrix/grid of cells
- Each cell: empty, contains a Dot, or a wall (#)
- A* state includes:
- Current Pacman position (x, y)
- Set of uncollected Dots (as a set or bitmask)
- Example:
- Initial state: Pacman at (0,0), all Dots remaining
- Goal state: no Dots remaining

2. Pac-Man Problem

Technical details of the algorithm's operation

Operational process:

1.Initialize:

- Add the initial state to the Open list (g=0, f=g+h)

2.Loop:

- Get the state with the smallest f from Open
- If all Dots have been eaten → ✅ Success
- Otherwise: expand valid adjacent cells (4 directions)
  - Ignore walls/outside map
  - If moving into a cell with a Dot → update the set of remaining Dots
- Recalculate g, h, f and update the Open list

3.End:

- If Open is empty → ❌ No **valid path**

2. Pac-Man Problem

Technical details of the algorithm's operation

Differences from a single-goal problem:

- **State:** Includes the set of Dots → larger search space
- **Stopping condition:** All Dots eaten, not just reaching 1 destination cell
- **Heuristic:**
  - Can use total Manhattan distance to remaining Dots
  - Simple but not always optimal
- **Optimality:** Maintained if heuristic is admissible

Code

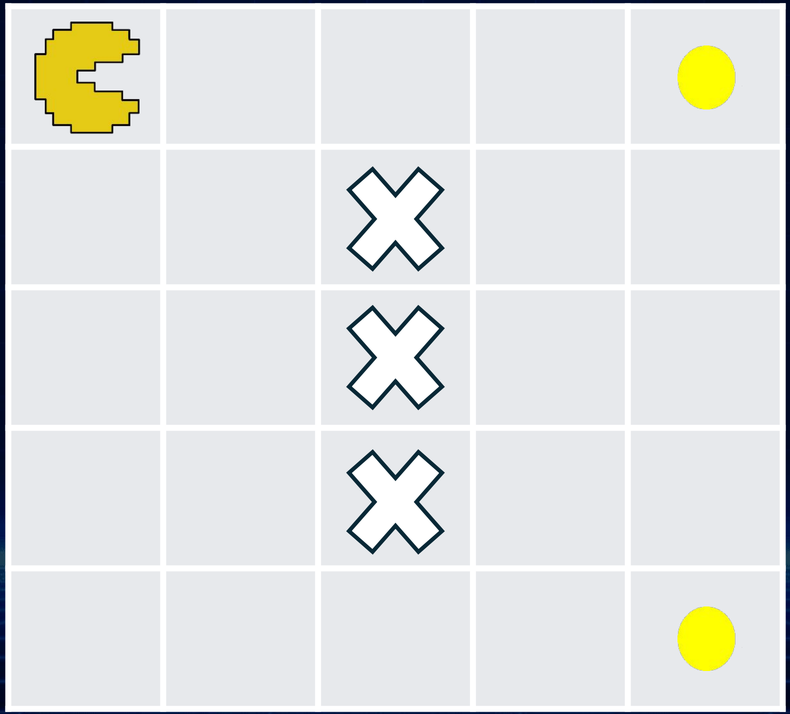
```
function AStar_MultiGoal(start_position, goal_set, grid):
    # Khởi tạo Open list (ưu tiên theo f) và Closed set
    Open := priority_queue()
    Closed := empty set


    # 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

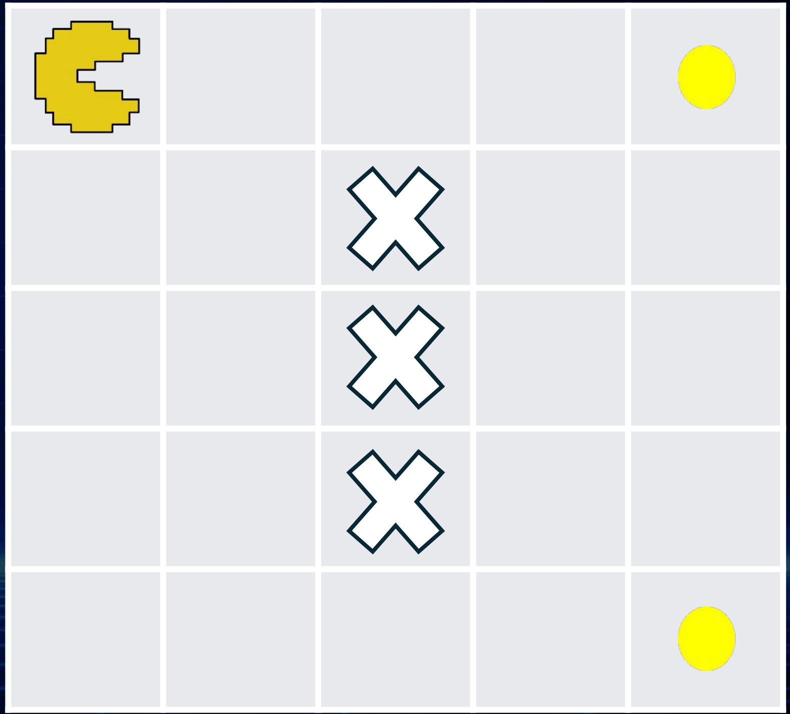
```
(current_pos, remainingGoals) := current_state
        # Kiểm tra mục tiêu: đã ăn hết tất cả Dot?
        if remainingGoals is empty:
            return reconstruct_path(current_state, parent)  # thành công, trả về đường đi


        # 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. Pac-Man Problem

Example of a
5x5 grid map:



2. Pac-Man Problem



Step 0 – Initialization

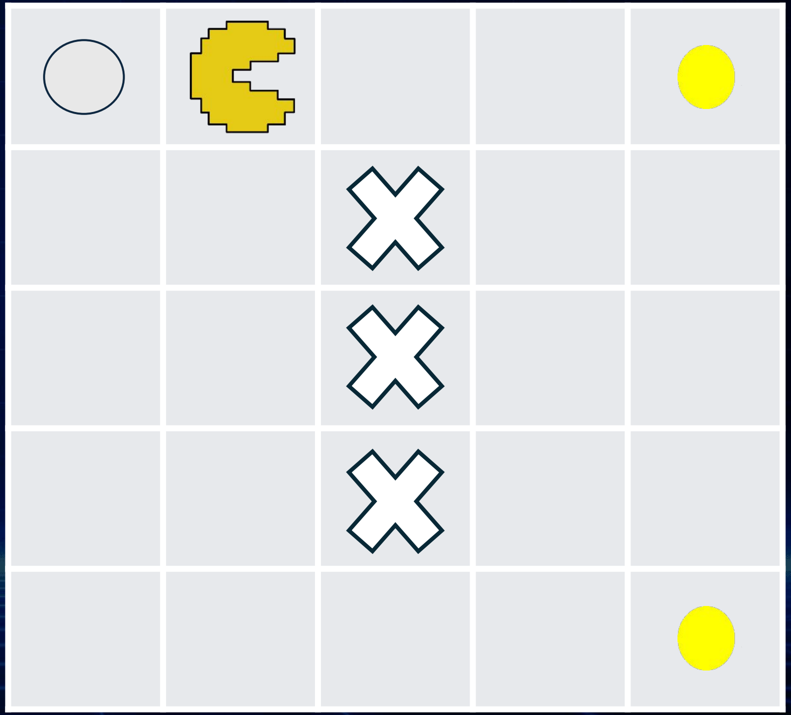
- Start: (0,0), goals {D1, D2}
- g=0, h=4+8=12, f=12
- **Open = {(0,0;{D1,D2}), f=12}, Closed = ∅**

2. Pac-Man Problem

Step 1 – Expand (0,0)

- Valid Neighbors: (0,1), (1,0)
- (0,1): f=11 → better
- (1,0): f=13

  → **Open = {(0,1):11, (1,0):13}**

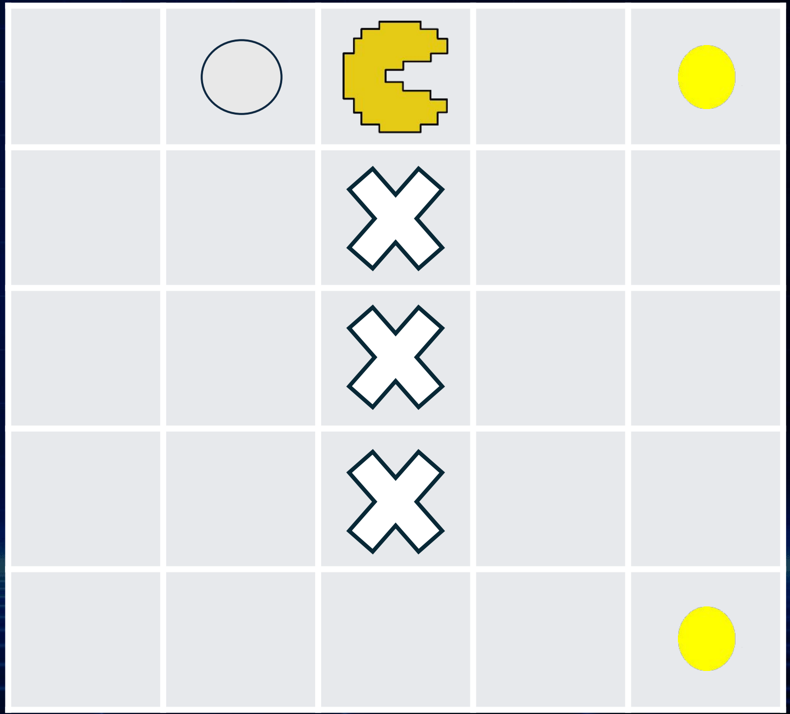


2. Pac-Man Problem

Step 2 – Expand (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}

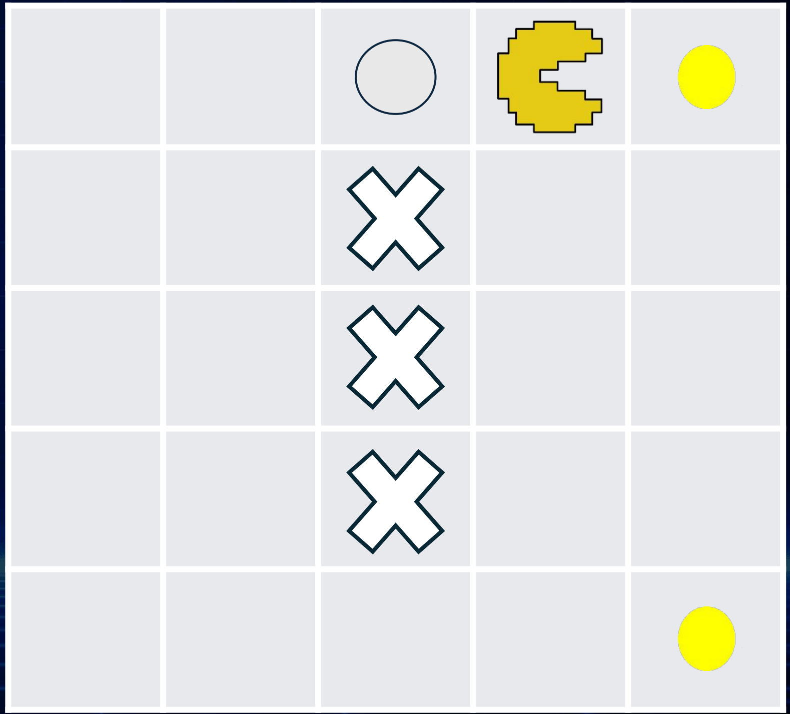


2. Pac-Man Problem

Bước 3 – Expand (0,2)

- Neighbor: (0,3) (because (1,2) is wall)
- (0,3): f=9

  → **Open = {(0,3):9, (1,1):12, (1,0):13}**

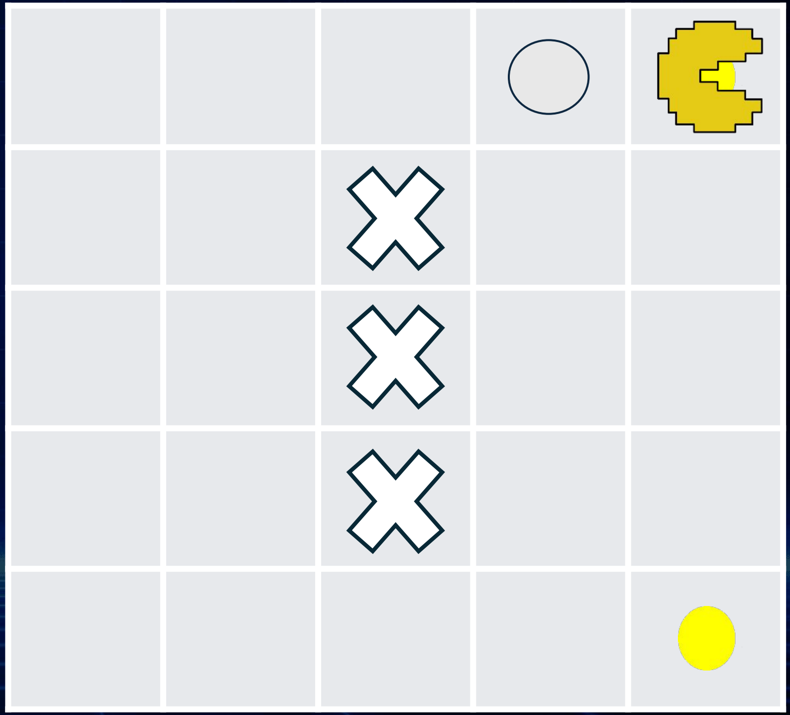


2. Pac-Man Problem

Step 4 – Expand (0,3)

- Neighbor: (0,4) (Dot D1), (1,3)
- ((0,4; remaining D2): f=8
- (1,3; D1 not yet eaten): f=10

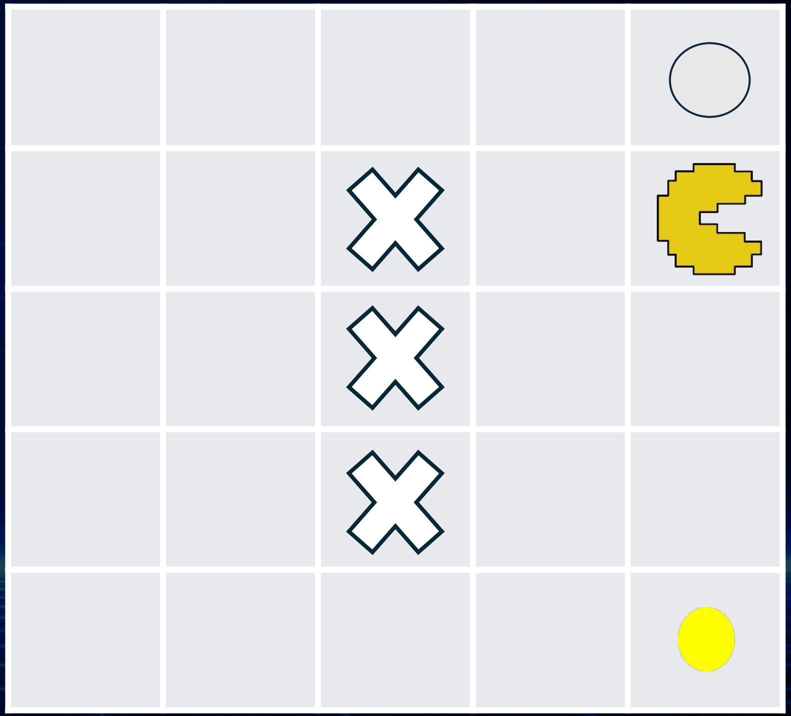
  → **Open = {(0,4;D2):8, (1,3):10, (1,1):12, (1,0):13}**



2. Pac-Man Problem

Step 5 – Expand (0,4; remaining D2)

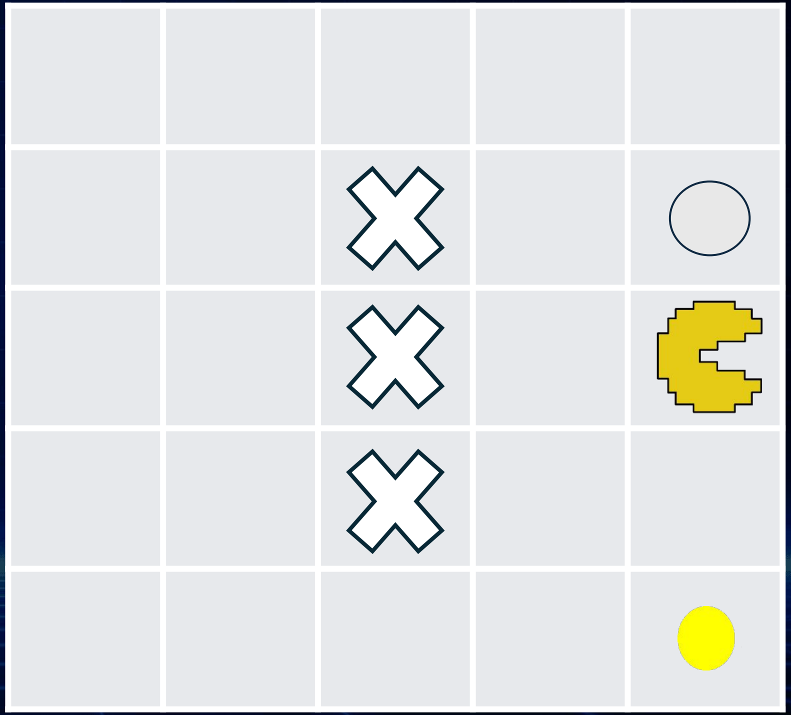
- Neighbor: (1,4): f=8
  → **Open = {(1,4;D2):8, (1,3):10, (1,1):12, (1,0):13}**



2. Pac-Man Problem

Step 6 – Expand (1,4;D2)

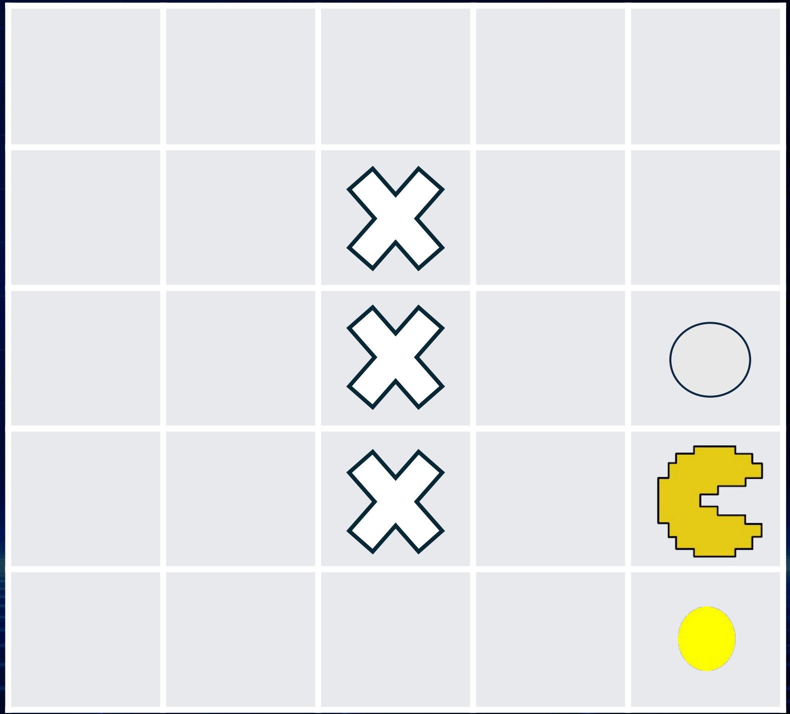
- Neighbor: (2,4): f=8, (1,3): f=10
  → Open = {(2,4;D2):8, (1,3):10, (1,1):12, (1,0):13}



2. Pac-Man Problem

Step 7 – Expand (2,4;D2)

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

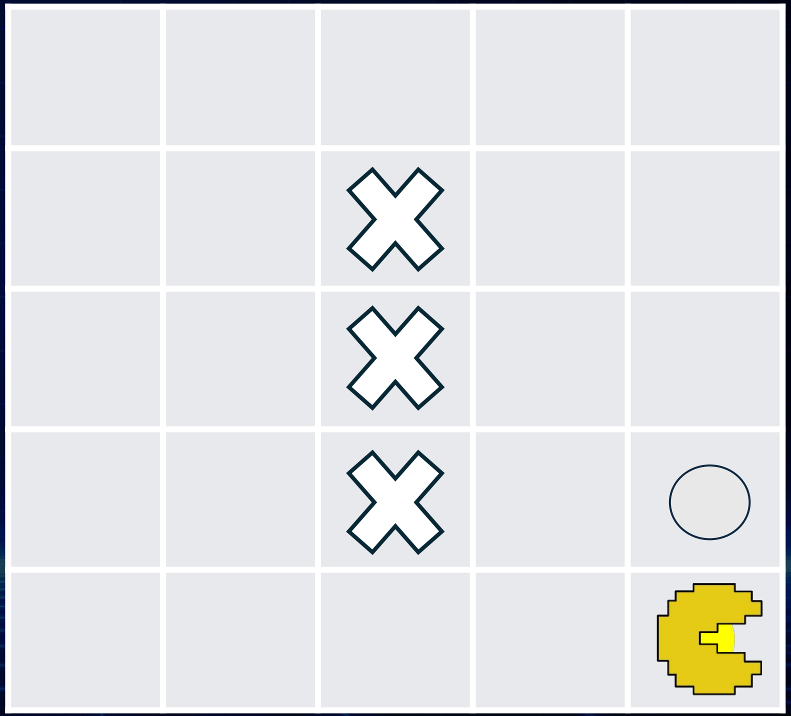


2. Pac-Man Problem

Step 8 – Expand (3,4;D2)

- Neighbor: (4,4) is Dot D2 → goal reached
- g=8, h=0, f=8

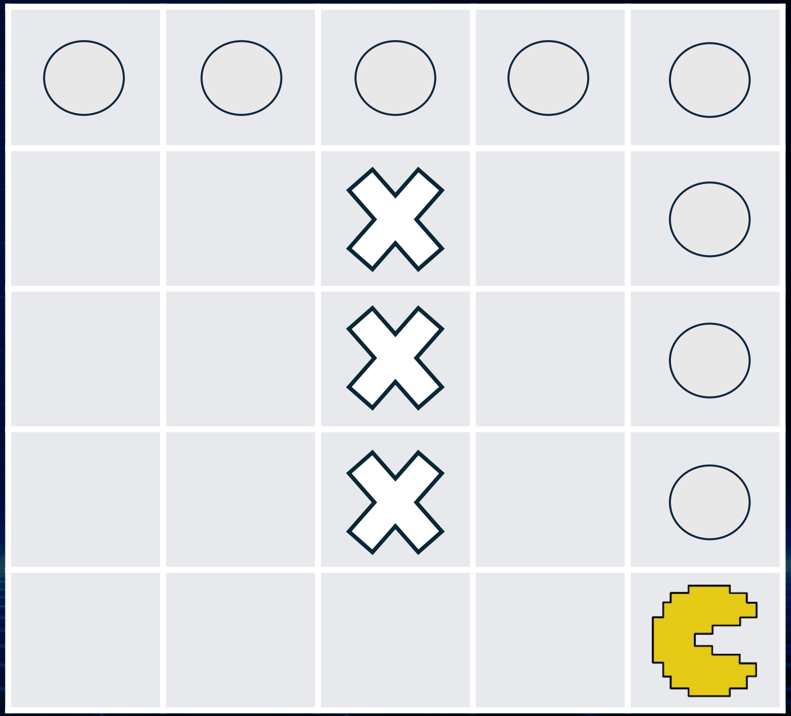
  Algorithm ends: Pacman has eaten all Dots.



2. Pac-Man Problem

Result

- **Path:** (0,0) → (0,1) → (0,2) → (0,3) → (0,4) → (1,4) → (2,4) → (3,4) → (4,4)
- **Total cost (g) = 8**
  A* found the optimal path, avoiding walls, satisfying the admissible heuristic condition.



2. Pac-Man Problem

Extended Algorithm (BFS)

How BFS works in the Pac-Man problem:

- Each cell on the map is a node in the graph.
- Edges connect two adjacent cells in 4 directions (up, down, left, right), unless there is a wall.
- BFS starts from Pac-Man's position.
- It uses a queue to store cells to be explored next.

  Whenever a cell is traversed, BFS:

  1. Marks that cell as visited.
  2. Adds valid adjacent cells (not walls, not yet visited) to the queue.
  3. If a food pellet is encountered, it stops — because BFS guarantees this is the shortest path.

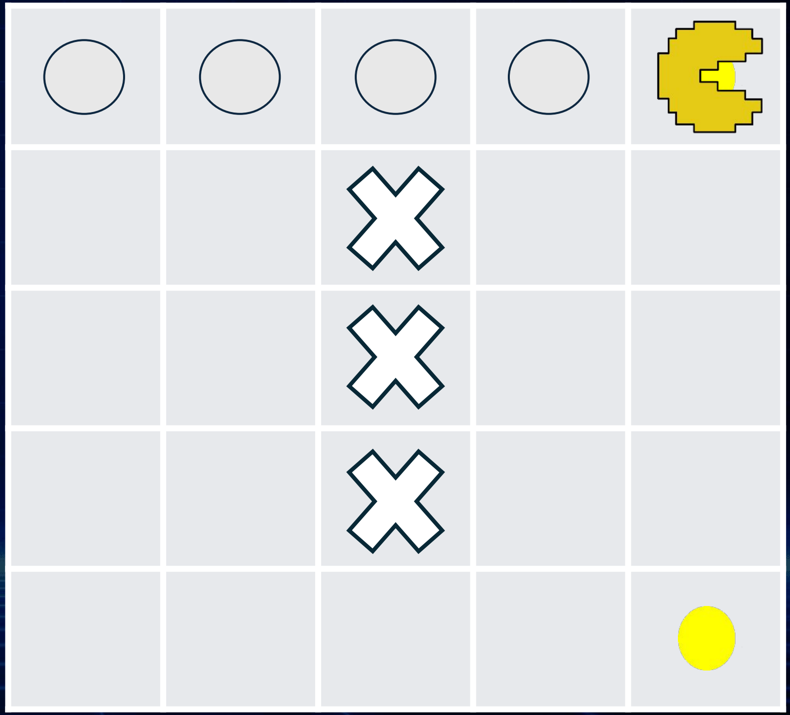
2. Pac-Man Problem

Extended Algorithm (BFS)

Applying BFS to find the shortest path to eat both pellets

Step 1: Find the nearest food pellet using BFS

From (0,0) to (0,4):
(0,0) → (0,1) → (0,2) → (0,3) → (0,4)
→ Pac-Man eats the first pellet



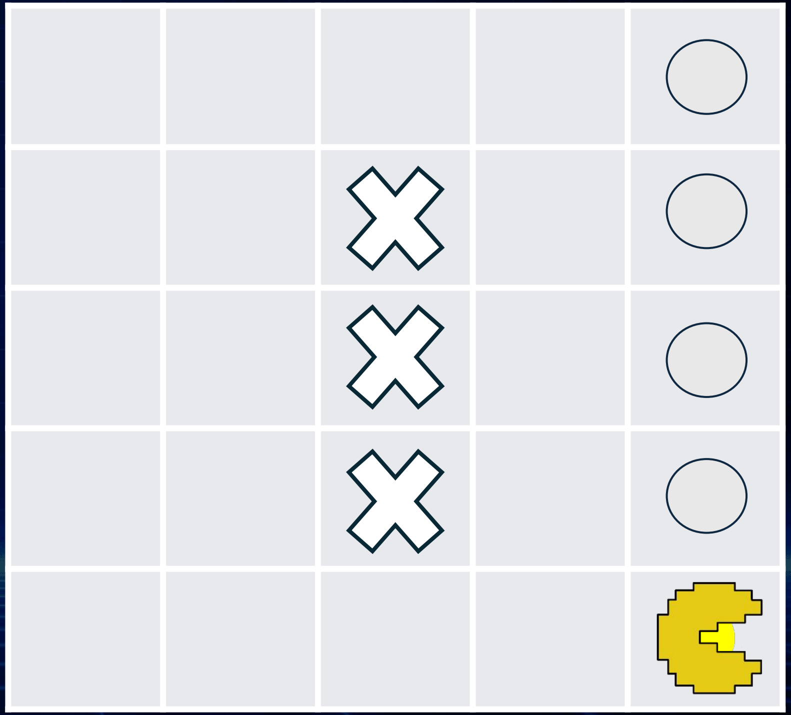
2. Pac-Man Problem

Extended Algorithm (BFS)

Applying BFS to find the shortest path to eat both pellets

Step 2: From the current position (0,4), find the shortest path to the remaining pellet (4,4)

Due to the wall column blocking the middle of the map, Pac-Man must go around through the right column:
(0,4) → (1,4) → (2,4) → (3,4) → (4,4)
→ Pac-Man eats the second pellet



2. Pac-Man Problem

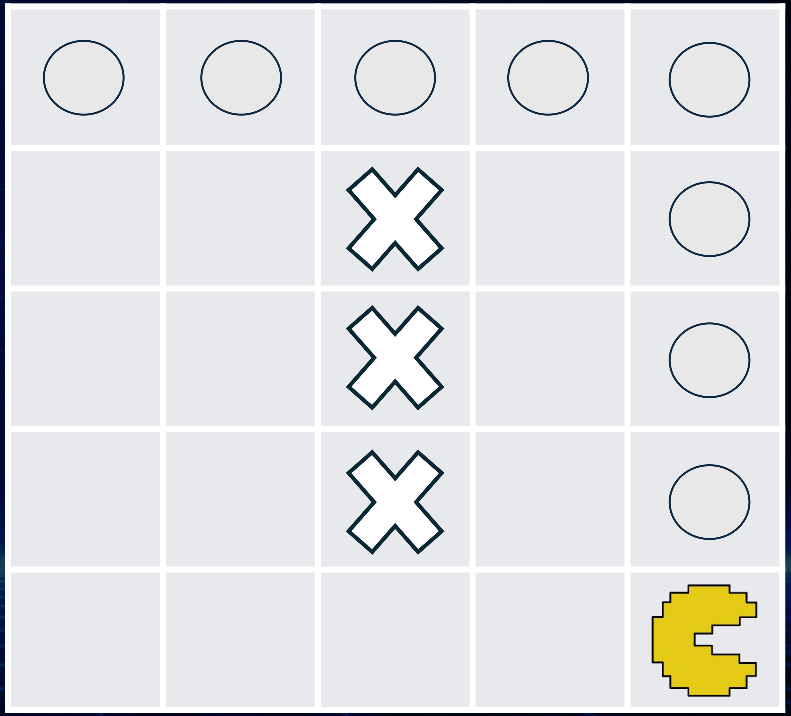
Extended Algorithm (BFS)

Applying BFS to find the shortest path to eat both pellets

Total shortest path to eat both pellets:

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

Total of 8 movement steps.



2. Pac-Man Problem

2.3 Evaluation

Compare BFS and A* (Manhattan heuristic)

Criterion	BFS	A* (Manhattan)
Path Length	9 cells (8 steps) – Optimal	9 ô (8 bước) – tối ưu
Path Optimality	Optimal in segments (shortest path)	Tối ưu nếu heuristic admissible
Time Complexity	O(b^d) – expands all nodes	O(b^d′) – ít hơn nhờ heuristic
Memory Usage	High (queue + visited)	Thấp hơn, có định hướng
Number of Expanded Nodes (5x5)	~23 cells	~10 cells
Execution Speed	Slower (~2.3× more nodes)	Faster, saves resources
Advantages	Simple, accurate	Efficient, goal-directed (guided)
Disadvantages	Consumes time/memory	Requires choosing an appropriate heuristic

2. Pac-Man Problem

2.3 Evaluation

Heuristic used:

- Total Manhattan distance from Pac-Man to all uncollected Dots
- **Meaning:** Estimates the total number of steps needed to eat all Dots
- **Advantages:**
    - Easy to calculate
    - Provides good guidance, helping to reduce the number of states to explore
- **Disadvantages:**
    - May overestimate the actual cost
    - Not entirely admissible, but still effective on small maps

2. Pac-Man Problem

2.3 Evaluation

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

Experiment & Evaluation

Setup:
- 5x5 map, no walls
- Start: (0,0)
- Dots: (4,0), (2,2), (4,4)
- Heuristic: total Manhattan
- Baseline: BFS (absolutely optimal path)

Results:
- A* found the correct path to eat all Dots
- Cost matches BFS → algorithm is correct
- Processing time: 0.0005s – 0.0011s

  → Faster than BFS, as heuristic reduces the number of expanded states

3. Thảo Luận

Limitations:

- Small map (5x5) → limited test data
- Simple heuristic (total Manhattan) → not yet optimal
- Static environment, no Ghosts yet

Future development:

- Expand map, add obstacles
- Add more agents (Ghosts, multiple Pac-Man)
- Use advanced heuristics (MST, machine learning)

4. Kết Luận

The application of the A* algorithm in the Pac-Man game demonstrates its effectiveness in finding optimal paths and intelligent movement control. A* maintains a balance between processing speed and accuracy, helping characters find the shortest routes in real-time. This proves that A* is a suitable choice for AI systems in games, contributing to enhancing strategic elements, difficulty, and the overall player experience.

Thank you!