ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Let's chase Pacman

Project 1: Search

Môn học: Cơ sở Trí tuệ nhân tạo

Sinh viên thực hiện:

Nguyễn Thị Lan Anh (21120198)

Trịnh Nguyên Lương (22120198)

Hoàng Lê Nam (22120217)

Trần Thảo Ngân (22120225)

Giáo viên hướng dẫn:

ThS. Nguyễn Thanh Tình

Ngày 19 tháng 4 năm 2025

Mục lục

1 Overview

1.1 Project Planning

Dự án được chia thành ba giai đoạn chính như sau:

1. **Giai đoạn 1 - Tìm hiểu và nghiên cứu**: Các thành viên trong nhóm cùng nhau tìm hiểu kiến thức liên quan, nghiên cứu yêu cầu dự án và thống nhất hướng phát triển.
2. **Giai đoạn 2 - Phát triển trò chơi**: Tiến hành lập trình game theo từng phần được phân công, đồng thời cập nhật tiến độ và hỗ trợ lẫn nhau trong quá trình triển khai.
3. **Giai đoạn 3 - Kiểm thử và hoàn thiện sản phẩm**: Thực hiện kiểm thử toàn bộ trò chơi, sửa lỗi nếu có, hoàn thiện báo cáo và thực hiện quay video sản phẩm.

1.2 Task Distribution

Nhóm gồm 4 thành viên, mỗi người phụ trách một phần công việc cụ thể như sau:

- **Nguyễn Thị Lan Anh (21120198)**: Phụ trách lập trình trò chơi và viết báo cáo cho Level 1 và Level 2. **Tỷ lệ hoàn thành công việc: 100%**
- **Trịnh Nguyên Lương (22120198)**: Phụ trách xây dựng mã nguồn tổng thể, lập trình trò chơi cho Level 5 và Level 6. **Tỷ lệ hoàn thành công việc: 100%**
- **Hoàng Lê Nam (22120217)**: Phụ trách lập trình trò chơi và viết báo cáo cho Level 3 và Level 4. **Tỷ lệ hoàn thành công việc: 100%**
- **Trần Thảo Ngân (22120225)**: Phụ trách lập trình giao diện trò chơi, quay video demo, tổng hợp và hoàn thiện các phần báo cáo còn lại. **Tỷ lệ hoàn thành công việc: 100%**

Việc đánh giá được nhóm thống nhất dựa trên mức độ đóng góp thực tế, thái độ làm việc và tinh thần trách nhiệm của từng thành viên trong suốt quá trình thực hiện dự án.

1.3 Demo video

Let's chase Pacman - Youtube

2 Search Algorithm Description

2.1 BFS (Breadth-First Search) - Blue Ghost Implementation

2.1.1 Giải thích thuật toán

Thuật toán Breadth-First Search (BFS) [2] – hay còn gọi là tìm kiếm theo chiều rộng – là một trong những thuật toán đơn giản và dễ hiểu nhất mà nhóm bọn em đã triển khai. Ý tưởng chính của BFS là duyệt từng lớp (level) của bản đồ: đi hết tất cả các vị trí gần nhất rồi mới xét đến những vị trí xa hơn.

BFS hoạt động bằng cách sử dụng một hàng đợi (queue) – chỗ này giống như việc chúng ta xếp hàng vậy, ai đến trước thì được xử lý trước. Điều đặc biệt là BFS đảm bảo tìm được đường đi ngắn nhất về số bước từ điểm bắt đầu (nơi ma xanh đang đứng) đến điểm đích (Pac-Man). Điều này rất phù hợp với đặc tính của Blue Ghost – chuyên đuổi Pac-Man một cách trực diện và nhanh chóng.

Algorithm 1 Breadth-First Search

```
1:  function BFS(problem)
2:      node ← a node with STATE = problem.INITIAL-STATE
3:      frontier ← a FIFO queue with node as the element
4:      explored ← an empty set
5:      loop
6:          if frontier is empty then
7:              return failure
8:          end if
9:          node ← POP-FRONT(frontier)
10:         if problem.GOAL-TEST(node.STATE) then
11:             return SOLUTION(node)
12:         end if
13:         add node.STATE to explored
14:         for all action ∈ problem.ACTIONS(node.STATE) do
15:             child ← Child-Node(problem, node, action)
16:             if child.STATE ∉ explored and child.STATE ∉ frontier then
17:                 append child to the end of frontier
18:             end if
19:         end for
20:     end loop
21: end function
```

Một số đặc điểm nổi bật của BFS:

- **Tìm được đường đi ngắn nhất (số bước):** Vì BFS mở rộng các vị trí gần nhất trước nên nếu mọi bước đi có chi phí bằng nhau, BFS luôn cho kết quả tối ưu.
- **Đảm bảo tìm ra lời giải (nếu tồn tại):** Miễn là có đường đến đích thì BFS chắc chắn sẽ tìm được.
- **Đơn giản, dễ hiểu, dễ cài đặt**

2.1.2 Cài đặt thuật toán trong project

```
def BFS(start, goal, graph):
    nodes_expanded = 0
    tracemalloc.start()
    start_time = time.perf_counter()
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        current_node, path = queue.popleft()
        if current_node == goal:
            end_time = time.perf_counter()
            current_mem, peak_mem = tracemalloc.get_traced_memory()
            tracemalloc.stop()
            return {
                'path': path,
                'nodes_expanded': nodes_expanded,
                'time_ms': (end_time - start_time) * 1000,
                'memory_kb': peak_mem / 1024
            }
        nodes_expanded += 1
        visited.add(current_node)
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))
    end_time = time.perf_counter()
    tracemalloc.stop()
```

```
28      return None
```

Listing 1: Hàm tìm đường cho Ma Xanh

Giải thích cách hoạt động:

Thuật toán sử dụng một hàng đợi queue để lưu các trạng thái sẽ xét tiếp. Mỗi phần tử trong queue là một tuple gồm: trạng thái hiện tại (current_node) và đường đi từ điểm xuất phát đến đó (path). Một tập visited được dùng để đánh dấu các nút đã được mở rộng, giúp tránh việc lặp lại.
Thuật toán sẽ duyệt lần lượt từng trạng thái trong hàng đợi:

- Nếu tìm thấy trạng thái hiện tại trùng với đích (goal), thì trả về kết quả bao gồm: đường đi, số nút đã mở rộng, thời gian chạy (ms), và bộ nhớ sử dụng (KB).
- Nếu chưa tới đích, thì mở rộng tất cả các nút kề chưa duyệt và thêm vào hàng đợi.

Trong quá trình chạy, nhóm sử dụng thư viện tracemalloc để đo lượng bộ nhớ sử dụng và time.perf_counter() để đo thời gian thực thi, từ đó có thể đánh giá hiệu năng thực tế của thuật toán.

2.2 DFS (Depth-First Search) - Pink Ghost Implementation

2.2.1 Giải thích thuật toán

Depth-First Search (DFS) [3] là một trong những thuật toán tìm kiếm đơn giản và phổ biến nhất. Thay vì mở rộng theo chiều rộng như DFS, DFS đi theo chiều sâu – tức là nó luôn cố gắng đi sâu nhất có thể vào một nhánh trước khi quay lại để thử nhánh khác.
DFS thường sử dụng ngăn xếp (stack) để quản lý các trạng thái đang chờ duyệt, hoặc được cài đặt thông qua đệ quy. Trong đồ thị không có chu trình, DFS có thể rất hiệu quả trong việc tìm kiếm nhanh đến đích nếu nó nằm gần nhánh đầu tiên được mở rộng. Tuy nhiên, DFS không đảm bảo tìm được đường đi ngắn nhất.
Trong game Pac-Man, dự án sử dụng thuật toán DFS cho Ma Hồng (Pink Ghost) – vốn có tính cách "bốc đồng" và hay đi theo những hướng bất ngờ. DFS phù hợp với hành vi này vì nó không tìm đường tối ưu mà đi sâu vào một nhánh ngẫu nhiên nào đó.

Algorithm 2 Thuật toán DFS (Depth-First Search)

```
1: function DFS(graph, start, goal)
2:     stack ← một ngăn xếp chứa (start, [start])
3:     visited ← ∅
4:     while stack không rỗng do
5:         (current, path) ← Pop(stack)
6:         if current ∈ visited then
7:             continue
8:         end if
9:         visited ← visited ∪ {current}
10:        if current = goal then
11:            return (path, nodes_expanded)
12:        end if
13:        for all neighbor ∈ graph[current] do
14:            if neighbor ∉ visited then
15:                Push((neighbor, path + [neighbor]) vào stack)
16:            end if
17:        end for
18:    end while
19:    return failure
20: end function
```

Đặc điểm và ưu điểm:

- **Đơn giản và dễ triển khai:** DFS có cấu trúc rõ ràng và dễ hiện thực trong cả ngôn ngữ thủ tục lẫn hướng đối tượng.
- **Tiết kiệm bộ nhớ hơn DFS:** Vì DFS không lưu tất cả các nút ở cùng mức, nên với đồ thị rộng, DFS sử dụng ít bộ nhớ hơn.

Quá trình thực hiện:

1. **Khởi tạo:**
   - Khởi tạo ngăn xếp (stack) với trạng thái bắt đầu.
   - Tạo tập visited rỗng để đánh dấu các trạng thái đã duyệt.
   - Thiết lập biến đếm nodes_expanded để theo dõi số node đã mở rộng.
2. **Vòng lặp chính:** Cho đến khi stack rỗng hoặc tìm thấy đích:
   - Lấy trạng thái ở đầu stack ra kiểm tra.

- Nếu đã duyệt, bỏ qua.
- Nếu là trạng thái đích, trả về kết quả.
- Nếu chưa, đánh dấu đã duyệt, rồi đẩy các trạng thái kề chưa duyệt vào stack để duyệt tiếp.

Độ phức tạp:

- **Thời gian:** $O(b^m)$ với b là bậc phân nhánh và m là chiều sâu tối đa.
- **Không tối ưu:** DFS có thể tìm được đường đi nhưng không đảm bảo đó là đường đi ngắn nhất.
- **Không đầy đủ:** Nếu đồ thị có chu trình hoặc vô hạn, DFS có thể không bao giờ tìm được lời giải nếu không giới hạn chiều sâu.

2.2.2 Cài đặt thuật toán trong project

```
1
2 def dfs(graph, start, goal):
3     start_time = time.time()
4     tracemalloc.start()
5
6     stack = [(start, [start])]
7     visited = set()
8     nodes_expanded = 0
9
10    while stack:
11        current, path = stack.pop()
12        if current in visited:
13            continue
14        if current == goal:
15            end_time = time.time()
16            current_mem, peak_mem = tracemalloc.get_traced_memory()
17            tracemalloc.stop()
18            return {
19                'path': path,
```

```
                'nodes_expanded': nodes_expanded,
                'time_ms': (end_time - start_time) * 1000,
                'memory_kb': peak_mem/ 1024
            }
        visited.add(current)
        nodes_expanded += 1
        for neighbor in graph.get(current, []):
            if neighbor not in visited:
                stack.append((neighbor, path + [neighbor]))

    end_time = time.time()
    tracemalloc.stop()
    return None
```

Listing 2: Hàm tìm đường cho Ma Hồng

Giải thích cách hoạt động: Thuật toán sử dụng ngăn xếp stack để lưu các trạng thái cần duyệt. Mỗi phần tử trong stack gồm: trạng thái hiện tại và đường đi đến đó.
Mỗi lần duyệt, nếu trạng thái đã được thăm thì bỏ qua. Nếu gặp đích thì trả về kết quả.
Nếu chưa đến đích, các trạng thái kề chưa được duyệt sẽ được thêm vào stack để tiếp tục đi sâu. Các trạng thái được thêm và duyệt theo thứ tự: phải, trái, xuống, lên.
Nhóm sử dụng tracemalloc để đo bộ nhớ và time.time() để đo thời gian chạy, giúp đánh giá hiệu suất từng thuật toán rõ ràng hơn.

2.3 UCS (Uniform-Cost Search) - Orange Ghost Implementation

2.3.1 Giải thích thuật toán

Uniform-Cost Search (UCS) là thuật toán tìm kiếm theo đồ thị nhằm tìm đường đi có tổng chi phí thấp nhất từ điểm xuất phát đến đích. UCS đảm bảo tìm được đường đi tối ưu toàn cục bằng cách luôn mở rộng nút có chi phí tích lũy thấp nhất từ nút gốc.
UCS sử dụng hàng đợi ưu tiên (priority queue) để quản lý tập biên (frontier). Khác với BFS chỉ quan tâm đến số lượng bước đi, UCS xét đến chi phí thực tế của mỗi hành động, nên đặc biệt hiệu quả khi các hành động có chi phí không đồng nhất.

Trong trò chơi Pac-Man, UCS được áp dụng cho Ma Cam (Orange Ghost) để tìm đường đi tối ưu đến Pac-Man, với hàm chi phí được thiết kế phù hợp với đặc tính của ma này.

Algorithm 3 Uniform-Cost Search

```
1: function UNIFORM-COST-SEARCH(problem)
2:     node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
3:     frontier ← a priority queue ordered by PATH-COST with node as the element
4:     explored ← an empty set
5:     loop
6:         if frontier is empty then
7:             return failure
8:         end if
9:         node ← POP(frontier)                    ▷ chooses the lowest-cost node in frontier
10:        if problem.GOAL-TEST(node.STATE) then
11:            return SOLUTION(node)
12:        end if
13:        add node.STATE to explored
14:        for all action in problem.ACTIONS(node.STATE) do
15:            child ← CHILD-NODE(problem, node, action)
16:            if child.STATE is not in explored and not in frontier then
17:                frontier ← INSERT(child, frontier)
18:            else if child.STATE is in frontier with higher PATH-COST then
19:                replace that frontier node with child
20:            end if
21:        end for
22:    end loop
23: end function
```

Đặc điểm và ưu điểm:

- **Tối ưu:** Luôn tìm được đường đi có chi phí thấp nhất từ nút xuất phát đến đích.
- **Đầy đủ:** Nếu mỗi hành động có chi phí không âm và số lượng hành động từ mỗi trạng thái là hữu hạn, UCS sẽ tìm ra lời giải nếu tồn tại.
- **Thích ứng:** Phù hợp cho các bài toán có chi phí không đồng nhất giữa các hành động.

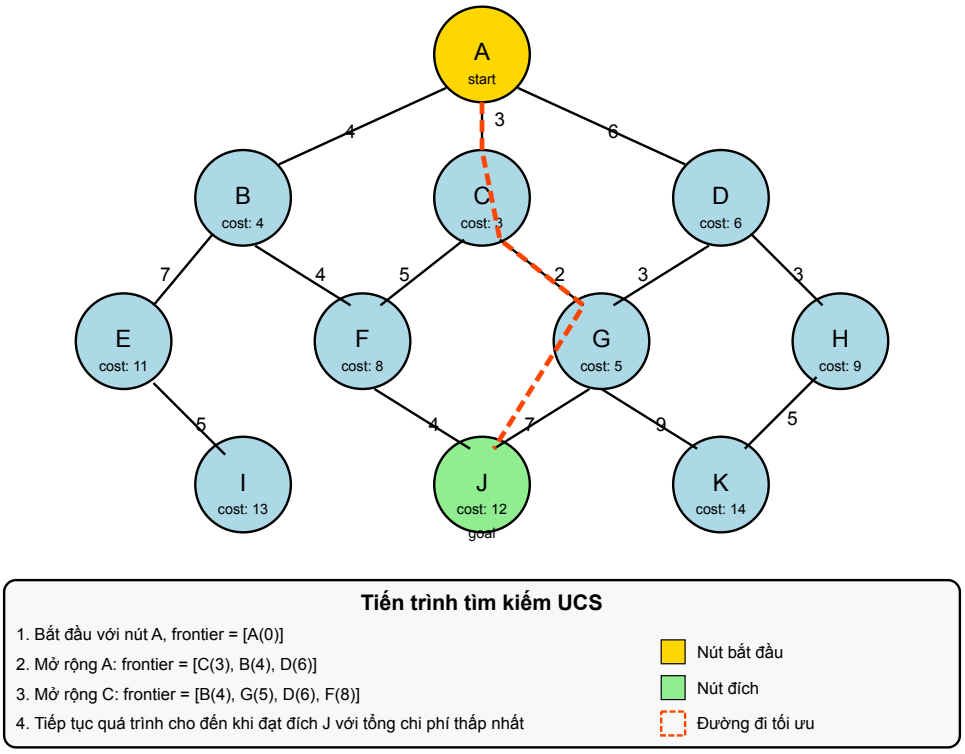
Quá trình thực hiện:

1. **Khởi tạo:** Tạo nút gốc với trạng thái ban đầu (chi phí = 0), thêm vào frontier, và khởi tạo tập explored rỗng.

2. **Vòng lặp chính:** Lặp lại cho đến khi tìm thấy đích hoặc frontier rỗng:
   - Chọn nút có chi phí thấp nhất từ frontier.
   - Nếu nút hiện tại là trạng thái đích, trả về lời giải.
   - Đánh dấu nút hiện tại đã được khám phá (explored).
   - Mở rộng nút hiện tại, tạo các nút con.
   - Với mỗi nút con: nếu chưa được khám phá và chưa có trong frontier, thêm vào frontier; nếu đã có trong frontier nhưng với chi phí cao hơn, cập nhật thành chi phí thấp hơn.

Độ phức tạp: Trong trường hợp xấu nhất, UCS có độ phức tạp thời gian là $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, trong đó b là hệ số phân nhánh, C^* là chi phí của lời giải tối ưu, và ϵ là chi phí hành động nhỏ nhất.

2.3.2 Hình ảnh và biểu đồ minh họa



Hình 1: Minh họa quá trình tìm kiếm của UCS trên đồ thị có trọng số

Hình 1 minh họa quá trình tìm kiếm UCS. Mỗi nút được mở rộng theo thứ tự tăng dần của chi phí tích lũy. Nút A là điểm xuất phát, nút J là đích. Các số trên cạnh biểu thị chi phí di chuyển giữa các nút. Chi phí của mỗi nút được hiển thị bên dưới tên nút. UCS sẽ mở rộng các nút theo thứ tự ưu tiên dựa trên chi phí tích lũy thấp nhất, bắt đầu từ A và khám phá qua C, G để đến đích J. Đường đi tối ưu cuối cùng là A → C → G → J với tổng chi phí 12.

2.3.3 Cài đặt chi tiết cho Ma Cam (Orange Ghost)

Trong trò chơi Pac-Man nguyên bản, Ma Cam (Clyde) có hành vi "rụt rè" - tìm đường đến Pac-Man khi ở xa, nhưng lại tránh xa khi đến gần. Ở phần cài đặt, điều này được mô phỏng thông qua hàm chi phí tùy chỉnh, khiến Ma Cam ưu tiên các đường đi xa trung tâm bản đồ.

```
def ucs(start, goal, graph, blocked_positions=[]):
    """
    Uniform Cost Search algorithm implementation

    This algorithm helps the Ghost find the optimal path to Pac-Man based on a
    custom cost function. UCS always expands the node with the lowest
   accumulated
    cost, ensuring that the optimal path is found.

    Args:
        start: Starting position of the Ghost, as a tuple (x, y)
        goal: Pac-Man's position, as a tuple (x, y)
        graph: Maze graph represented as a dictionary (adjacency list)
        blocked_positions: List of blocked positions (other ghosts or walls)

    Returns:
        dict: Information about the found path, including:
            - path: List of positions along the path
            - nodes_expanded: Number of nodes expanded/explored
            - time_ms: Execution time (milliseconds)
            - memory_kb: Memory used (KB)
            - cost: Total path cost
        Or None if no path is found
    """
    nodes_expanded = 0
```

```
25
26      # Start measuring time and memory to evaluate algorithm performance
27      # tracemalloc tracks memory usage, time.perf_counter() measures precise time
28      tracemalloc.start()
29      start_time = time.perf_counter()
30
31      # Priority queue with structure (cost, counter, node, path, actual_cost)
32      # Counter is used to ensure stability when two nodes have the same cost
33      frontier = [(0, 0, start, [start], 0)]
34
35      # Dictionary storing nodes in frontier with their corresponding costs
36      # Helps quickly check if a node is in the frontier and what its cost is
37      # Structure: {node: cost}
38      frontier_dict = {start: 0}
39
40      # Dictionary storing explored nodes and their optimal costs
41      # Helps avoid reconsidering nodes that already have optimal paths
42      # Structure: {node: optimal_cost}
43      explored = {}
44
45      # Incrementing counter, used to ensure queue stability when comparing nodes
    with the same cost
46      counter = 1
47
48      while frontier:
49          # Get the node with the lowest cost from the priority queue
50          priority, _, current, path, actual_cost = heapq.heappop(frontier)
51
52          # Remove the current node from frontier_dict as it's being processed
53          if current in frontier_dict:
54              del frontier_dict[current]
55
56          # Check if we've reached the goal
57          if current == goal:
58              # End time and memory measurement
59              end_time = time.perf_counter()
60              current_mem, peak_mem = tracemalloc.get_traced_memory()
```

```
            tracemalloc.stop()

            # Return detailed information about the found path
            return {
                'path': path,  # List of positions along the path
                'nodes_expanded': nodes_expanded,  # Number of nodes expanded during search
                'time_ms': (end_time - start_time) * 1000,  # Execution time (ms)
                'memory_kb': peak_mem / 1024,  # Maximum memory used (KB)
                'cost': actual_cost  # Total path cost
            }

        # Skip if the node has already been explored with a better cost
        if current in explored and explored[current] <= actual_cost:
            continue

        # Mark the current node as explored with the new cost
        explored[current] = actual_cost
        nodes_expanded += 1

        # Consider all neighbors of the current node
        for neighbor in graph[current]:
            # Skip blocked positions
            if neighbor in blocked_positions:
                continue

            # Calculate the cost of moving from the current node to the neighbor
            step_cost = calculate_cost(current, neighbor, pacman_pos=goal)
            new_actual_cost = actual_cost + step_cost

            # Only consider the neighbor if one of two conditions is met:
            # 1. Node has never been explored, or a better path has been found
            # 2. Node is in the frontier but a better path has been found
            if (neighbor not in explored or explored[neighbor] > new_actual_cost) and \
```

```
                    (neighbor not in frontier_dict or frontier_dict[neighbor] >
    new_actual_cost):
                # Update or add to frontier
                frontier_dict[neighbor] = new_actual_cost
                heapq.heappush(frontier,
                               (new_actual_cost, counter, neighbor, path + [
    neighbor], new_actual_cost))
                counter += 1  # Increase counter to ensure priority queue
    stability

     # End search if no path is found
     end_time = time.perf_counter()
     tracemalloc.stop()
     return None
```

Listing 3: Cài đặt thuật toán UCS cho Ma Cam

Hàm chi phí đặc biệt: Tính cách "rụt rè" của Ma Cam được mô phỏng thông qua hàm chi phí tùy chỉnh, tính toán chi phí di chuyển dựa trên khoảng cách đến trung tâm bản đồ:

```
def calculate_cost(current, neighbor, pacman_pos):
    """
    Cost calculation function for the Orange Ghost

    The Orange Ghost has a special movement strategy:
    - Avoid the central area of the map (where danger is typically higher)
    - Maintain a moderate distance from Pac-Man: not too close to avoid
      direct confrontation, but not too far to keep track of Pac-Man

    Args:
        current: Current position of the Orange Ghost, as a tuple (x, y)
        neighbor: Neighbor position being considered, as a tuple (x, y)
        pacman_pos: Pac-Man's position, as a tuple (x, y)

    Returns:
        float: Cost of moving to the neighbor position (lower is prioritized)
    """
    # Map center - the area the Orange Ghost wants to avoid
```

```
    map_center = (14, 14)

    CENTER_WEIGHT = 10  # Priority level for avoiding the center
    PACMAN_FOLLOW_WEIGHT = 3  # Priority level for maintaining distance from Pac-Man

    # The closer to the center, the higher the cost -> Ghost will tend to avoid it
    center_distance = manhattan_distance(neighbor, map_center)

    # Estimate the maximum distance from any point to the center
    # Use half the map size as a standard (map approximately 30x32)
    max_possible_distance = max(30, 32) / 2

    # Normalize distance to range [0,1]
    # 0: At the center, 1: At the edge of the map
    normalized_distance = min(1.0, center_distance / max_possible_distance)

    # Calculate penalty cost for positions near the center
    # Using a non-linear function (1-d)^2:
    # - When d=0 (at center): Penalty = CENTER_WEIGHT * 1 = 10
    # - When d=1 (far from center): Penalty = CENTER_WEIGHT * 0 = 0
    center_penalty = CENTER_WEIGHT * (1 - normalized_distance) ** 2

    # Current distance to Pac-Man
    pacman_distance = manhattan_distance(neighbor, pacman_pos)

    # Ideal distance the Orange Ghost wants to maintain from Pac-Man
    # Not too close to avoid confrontation, not too far to keep tracking
    ideal_distance = 8

    # Cost based on the difference from the ideal distance
    # The more different from 8 cells, the higher the cost
    pacman_factor = PACMAN_FOLLOW_WEIGHT * abs(pacman_distance - ideal_distance) / 10

    # Base cost for each movement step
```

```
    base_cost = 1

    # Total cost = base cost + center avoidance cost + Pac-Man tracking cost
    # Lower cost is prioritized in the UCS algorithm
    total_cost = base_cost + center_penalty + pacman_factor

    return total_cost
```

Listing 4: Hàm tính chi phí tùy chỉnh cho Ma Cam

Hàm tính toán chi phí này được cài đặt để phản ánh hành vi đặc trưng của Ma Cam trong game gốc và tối ưu hóa chiến thuật săn đuổi Pac-Man:

1. **Mô phỏng tính cách "rụt rè":** Trong Pac-Man nguyên bản, Ma Cam (Clyde) nổi tiếng với tính cách lẩn tránh Pac-Man khi ở gần và chỉ truy đuổi khi ở xa. Hàm tính chi phí mô phỏng hành vi này thông qua qua:
    - *Tránh khu vực trung tâm:* Trung tâm bản đồ là nơi Pac-Man thường xuyên hoạt động và có nhiều ngã rẽ, dẫn đến rủi ro cao. Ma Cam tránh khu vực này nhờ vào chi phí `CENTER_WEIGHT * (1 - normalized_distance)²`.
    - *Duy trì khoảng cách an toàn:* Hàm ưu tiên khoảng cách 8 ô với Pac-Man, đủ xa để tránh đối đầu trực tiếp nhưng vẫn theo dõi và đuổi được mục tiêu.
2. **Chiến thuật tấn công gián tiếp:** Khác với Ma Đỏ thường tấn công trực diện, Ma Cam tạo ra chiến thuật phức tạp hơn:
    - *Bao vây từ bên ngoài:* Di chuyển ở khu vực ngoài rìa giúp Ma Cam tạo thành một phần của chiến thuật bao vây Pac-Man từ nhiều hướng khác nhau.
    - *Khó dự đoán:* Cân bằng giữa hai yếu tố đối lập (tránh trung tâm và theo dõi Pac-Man) tạo ra hành vi khó đoán, giúp gây khó khăn cho người chơi trong việc tính toán đường đi di chuyển.

Cách tính chi phí này vừa phản ánh chân thực tính cách của Ma Cam trong game gốc, vừa tạo ra hành vi thông minh và đa dạng, góp phần nâng cao trải nghiệm game cho người chơi thông qua các chiến thuật tấn công phức tạp.

```python
def manhattan_distance(a, b):
    """
    Calculate Manhattan distance between two points

    Manhattan distance is the sum of the absolute differences between
    corresponding coordinates. Suitable for grid-based movement (up, down,
    left, right) in 2D space.

    Args:
        a: First point, as a tuple (x, y)
        b: Second point, as a tuple (x, y)

    Returns:
        int: Manhattan distance between the two points
    """
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

Listing 5: Hàm tính khoảng cách Manhattan

Triển khai đầy đủ: Toàn bộ quy trình tìm đường của Ma Cam được triển khai như sau:

```python
def orange_ghost_path(ghost_pos, pacman_pos, graph, blocked_positions=[]):
    """
    Determine the path for the Orange Ghost to follow Pac-Man using UCS
   algorithm

    Orange Ghost strategy:
    - Use UCS with a special cost function to find a path to Pac-Man
    - Move more cautiously than the Red Ghost, prioritizing peripheral areas
    - Tend to "trail" Pac-Man rather than attacking directly

    Args:
        ghost_pos: Current position of the Orange Ghost, as a tuple (x, y)
        pacman_pos: Current position of Pac-Man, as a tuple (x, y)
        graph: Graph representing the maze
        blocked_positions: List of blocked positions (other ghosts)

```

```
    Returns:
        list: List of positions along the path from Orange Ghost to Pac-Man
              or an empty list if no path is found
    """
    # Check special case: Ghost is already at Pac-Man's position
    if ghost_pos == pacman_pos:
        return []  # No path needed as already at destination

    # Call UCS algorithm to find the optimal path
    result = ucs(ghost_pos, pacman_pos, graph, blocked_positions)

    # Handle search result
    if result:
        # Print detailed information about the found path
        print(f"Orange ghost path found")
        print("Path:", result['path'])  # List of positions along the path
        print("Nodes expanded:", result['nodes_expanded'])  # Number of nodes considered during search
        print("Time (ms):", round(result['time_ms'], 3))  # Execution time (rounded to 3 decimal places)
        print("Memory (KB):", round(result['memory_kb'], 2))  # Memory usage (rounded to 2 decimal places)
        print("Path cost:", result['cost'])  # Total path cost
        return result['path']
    else:
        # Print message if no path is found
        print(f"No path found from ghost{ghost_pos} to pacman{pacman_pos}.\n")
        return []
```

Listing 6: Hàm tìm đường cho Ma Cam

2.4 A* Search (A-Star) - Red Ghost Implementation

2.4.1 Giải thích thuật toán

A* Search là một thuật toán tìm kiếm tiên tiến kết hợp ưu điểm của Uniform-Cost Search (UCS) và tìm kiếm heuristic. Thuật toán này sử dụng hàm đánh giá f(n) = g(n) + h(n), trong đó g(n) là

chi phí thực từ trạng thái ban đầu đến trạng thái hiện tại n, và h(n) là hàm heuristic ước lượng chi phí từ n đến trạng thái đích. Trong trò chơi Pac-Man, A* được sử dụng cho Ma Đỏ (Red Ghost) để tìm đường đi tối ưu đến Pac-Man với tốc độ nhanh hơn các thuật toán khác.

Algorithm 4 A* Search

```
1: function A-STAR-SEARCH(problem)
2:     node ← node with STATE = problem.INITIAL-STATE, PATH-COST = 0
3:     frontier ← priority queue ordered by f(n) = g(n) + h(n) with node as the first element
4:     explored ← empty set
5:     loop
6:         if frontier is empty then
7:             return failure
8:         end if
9:         node ← POP(frontier)                    ▷ choose the node with lowest f(n) in frontier
10:        if problem.GOAL-TEST(node.STATE) then
11:            return SOLUTION(node)
12:        end if
13:        add node.STATE to explored
14:        for all action in problem.ACTIONS(node.STATE) do
15:            child ← CHILD-NODE(problem, node, action)
16:            if child.STATE is not in explored and not in frontier then
17:                add child to frontier
18:            else if child.STATE is in frontier with higher f(n) then
19:                replace that node in frontier with child
20:            end if
21:        end for
22:    end loop
23: end function
```

Nguyên lý cơ bản và ưu điểm: A* kết hợp hai yếu tố quan trọng trong đánh giá một nút:

- **g(n) - Chi phí thực tế:** Chi phí đã biết từ trạng thái ban đầu đến trạng thái n (giống như UCS).
- **h(n) - Hàm heuristic:** Ước lượng chi phí từ trạng thái n đến trạng thái đích.

Ưu điểm nổi bật của A*:

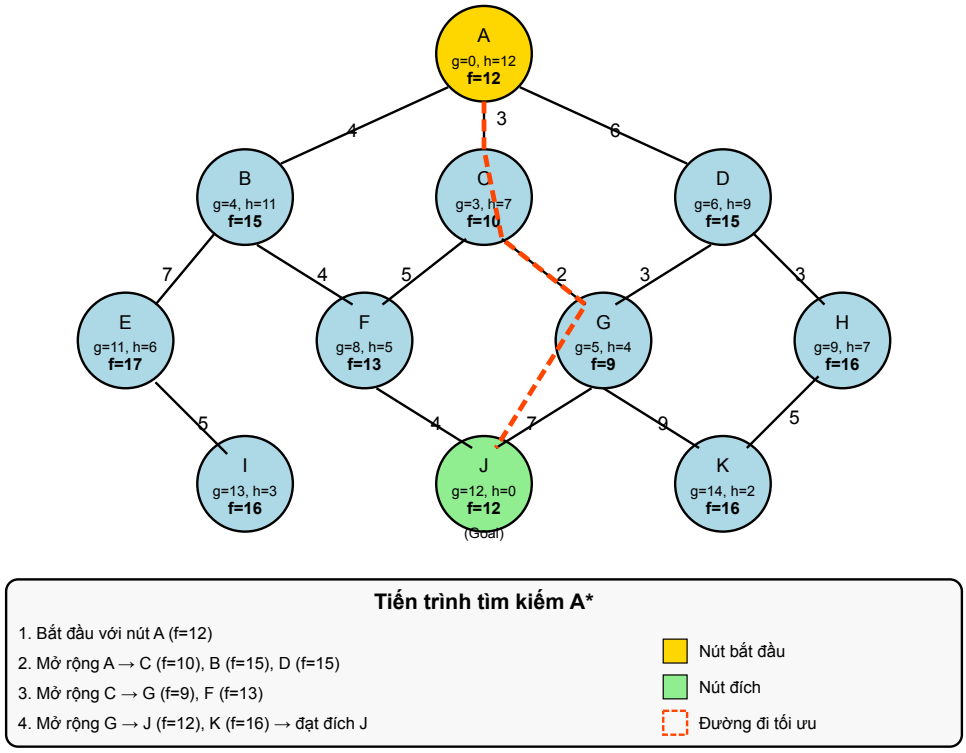
- **Tối ưu:** Khi hàm h(n) là *admissible* (không bao giờ ước lượng quá chi phí thực tế đến đích), A* đảm bảo tìm ra đường đi tối ưu.

- **Hiệu quả:** A* thường mở rộng ít nút hơn UCS đáng kể nhờ thông tin heuristic định hướng việc tìm kiếm.
- **Linh hoạt:** Có thể tùy chỉnh hàm heuristic cho từng bài toán cụ thể.

Quá trình thực hiện:

1. **Khởi tạo:**
   - Tạo nút gốc với trạng thái ban đầu, g(n) = 0
   - Tính h(n) cho nút gốc và f(n) = g(n) + h(n)
   - Đưa nút gốc vào hàng đợi ưu tiên (frontier)
2. **Lặp chính:**
   - Nếu frontier rỗng, trả về thất bại
   - Lấy nút có f(n) thấp nhất từ frontier
   - Nếu nút hiện tại là trạng thái đích, trả về lời giải
   - Thêm nút hiện tại vào tập explored
3. **Mở rộng:**
   - Xem xét tất cả hành động có thể từ nút hiện tại
   - Với mỗi hành động, tính g(n) mới, h(n) và f(n) cho nút con
   - Nếu nút con chưa thuộc explored và frontier, thêm vào frontier
   - Nếu nút con đã thuộc frontier nhưng với f(n) cao hơn, cập nhật giá trị thấp hơn

2.4.2 Hình ảnh và biểu đồ minh họa



Hình 2: Minh họa quá trình tìm kiếm của A* trên đồ thị có trọng số

Hình 2 minh họa quá trình tìm kiếm A* trên đồ thị có trọng số. Mỗi nút hiển thị ba giá trị: g(n), h(n) và f(n) = g(n) + h(n). Nút có f(n) thấp nhất được mở rộng trước. Thuật toán A* ưu tiên thăm các nút có tổng f(n) thấp, do đó quá trình tìm kiếm được dẫn hướng tốt hơn về phía đích.

2.4.3 Cài đặt chi tiết cho Ma Đỏ (Red Ghost)

Trong trò chơi Pac-Man nguyên bản, Ma Đỏ (tên gốc: Blinky) là ma hung hãn và quyết liệt nhất, luôn trực tiếp đuổi theo Pac-Man bằng đường đi ngắn nhất. Đặc điểm này được thể hiện thông qua việc sử dụng thuật toán A* với hàm heuristic là khoảng cách Manhattan, giúp Ma Đỏ tiếp cận Pac-Man nhanh chóng và hiệu quả.

```
def astar_search(start, goal, graph, blocked_positions=None):
    """
    A* (A-star) search algorithm

```

```
    This algorithm finds the optimal path from start to goal
    by combining actual cost (g_score) and heuristic estimate (h_score).
    Formula: f_score = g_score + h_score, where:
    - g_score: Actual cost from start position to current position
    - h_score: Estimated cost from current position to goal (Manhattan distance)

    Parameters:
        start: Ghost's starting position, as tuple (x, y)
        goal: Pac-Man's position, as tuple (x, y)
        graph: Graph representing the maze (dictionary of adjacency list)
        blocked_positions: List of blocked positions (other ghosts or walls)

    Returns:
        dict: Information about the found path, including:
            - path: List of positions along the path
            - nodes_expanded: Number of nodes expanded/explored
            - time_ms: Execution time (milliseconds)
            - memory_kb: Memory used (KB)
            - cost: Total path cost
        Or None if no path is found
    """
    # Initialize empty list if no blocked positions
    if blocked_positions is None:
        blocked_positions = []

    nodes_expanded = 0

    # Start measuring time and memory to evaluate algorithm performance
    # tracemalloc tracks memory usage, time.perf_counter() measures precise time
    tracemalloc.start()
    start_time = time.perf_counter()

    # Initialize priority queue with components:
    # - f_score: total estimated cost (g_score + h_score)
    # - counter: counter to ensure stability when two nodes have the same
   f_score
    # - node: current node being considered
```

```
    # - path: path from start to current node
    # - g_score: actual cost from start to current node
    frontier = [(heuristic(start, goal), 0, start, [start], 0)]

    # Dictionary tracking nodes in frontier with corresponding g_score
    # Structure: {node: g_score}
    frontier_dict = {start: 0}

    # Set of explored nodes to avoid revisiting
    explored = set()

    # Incrementing counter to ensure priority queue stability
    counter = 1

    while frontier:
        # Get node with highest priority (lowest f_score) from queue
        f_score, _, current, path, g_score = heapq.heappop(frontier)
        nodes_expanded += 1

        # If goal reached, end search and return result
        if current == goal:
            # End time and memory measurement
            end_time = time.perf_counter()
            current_mem, peak_mem = tracemalloc.get_traced_memory()
            tracemalloc.stop()

            # Return detailed results about the found path
            return {
                'path': path,  # List of nodes on the path
                'nodes_expanded': nodes_expanded,  # Number of nodes explored
                'time_ms': (end_time - start_time) * 1000,  # Execution time (ms
)
                'memory_kb': peak_mem / 1024,  # Maximum memory used (KB)
                'cost': g_score  # Total path cost
            }

        # Skip current node if already explored
```

```
        if current in explored:
            continue

        # Mark current node as explored
        explored.add(current)

        # Consider all neighbor nodes of current node
        for neighbor in graph[current]:
            # Skip neighbor if already explored or is a blocked position
            if neighbor in explored or neighbor in blocked_positions:
                continue

            # Calculate new g_score when moving from current node to neighbor
            new_g_score = g_score + calculate_cost(current, neighbor)

            if neighbor in frontier_dict:
                # If neighbor already in frontier with higher cost, update
                if frontier_dict[neighbor] > new_g_score:
                    # Update cost in frontier_dict
                    frontier_dict[neighbor] = new_g_score

                    # Calculate new h_score and f_score
                    h_score = heuristic(neighbor, goal)
                    f_score = new_g_score + h_score

                    # Add node with updated cost to frontier
                    heapq.heappush(frontier, (f_score, counter, neighbor, path +
    [neighbor], new_g_score))
                    counter += 1  # Increase counter to ensure stability
            else:
                # If neighbor not in frontier, add it
                frontier_dict[neighbor] = new_g_score

                # Calculate h_score and f_score
                h_score = heuristic(neighbor, goal)
                f_score = new_g_score + h_score

```

```
                # Add new node to frontier
                heapq.heappush(frontier, (f_score, counter, neighbor, path + [neighbor], new_g_score))
                counter += 1  # Increase counter to ensure stability

    # If no path to goal is found
    end_time = time.perf_counter()
    tracemalloc.stop()
    return None
```

Listing 7: Cài đặt thuật toán A* cho Ma Đỏ

Hàm heuristic - Khoảng cách Manhattan: Hàm heuristic là thành phần quan trọng nhất trong thuật toán A*. Đối với Ma Đỏ, lần này ta sẽ sử dụng khoảng cách Manhattan làm heuristic vì:

- **Admissible:** Trong không gian lưới 2D như mê cung Pac-Man mà nhóm đã triển khai thì các con ma chỉ có thể di chuyển theo 4 hướng (lên, xuống, trái, phải) và không thể di chuyển theo đường chéo, không thể di chuyển nhanh hơn khoảng cách Manhattan. Điều này đảm bảo hàm heuristic không bao giờ ước tính cao hơn chi phí thực tế, một điều kiện quan trọng để A* đảm bảo tìm ra đường đi tối ưu.

  Chứng minh tính admissible: Giả sử ta có node hiện tại n với tọa độ (x_1, y_1) và node đích g với tọa độ (x_2, y_2). Khoảng cách Manhattan giữa chúng là $h(n) = |x_1 - x_2| + |y_1 - y_2|$. Trong không gian lưới 2D với 4 hướng di chuyển, để đi từ n đến g, ta phải di chuyển ít nhất $|x_1 - x_2|$ bước theo chiều ngang và $|y_1 - y_2|$ bước theo chiều dọc. Vì không thể đi đường chéo và mỗi bước có chi phí là 1, nên tổng chi phí thực tế $h^*(n) \geq |x_1 - x_2| + |y_1 - y_2| = h(n)$. Trong trường hợp không có chướng ngại vật (tường), $h^*(n) = h(n)$. Khi có tường chặn, $h^*(n) > h(n)$ do phải đi vòng. Do đó, $h(n) \leq h^*(n)$ luôn đúng, chứng tỏ khoảng cách Manhattan là admissible.

- **Consistent (Monotonic):** Khoảng cách Manhattan còn thỏa mãn tính chất consistent, đảm bảo thuật toán A* không chỉ tìm ra đường đi tối ưu mà còn hiệu quả về mặt tính toán.

  Chứng minh tính consistent: Một hàm heuristic $h(n)$ là consistent khi $h(n) \leq c(n, n') + h(n')$ với mọi node n và successor n' của nó, trong đó $c(n, n')$ là chi phí để đi từ n đến n'. Xét node n với tọa độ (x_1, y_1) và successor n' với tọa độ (x_1', y_1'). Vì n' là successor trực tiếp của n

nên chỉ có một trong hai tọa độ thay đổi và thay đổi chính xác 1 đơn vị, tức $x_1' = x_1 \pm 1$ và $y_1' = y_1$, hoặc $x_1' = x_1$ và $y_1' = y_1 \pm 1$.

Xét trường hợp $x_1' = x_1 + 1$ và $y_1' = y_1$ (di chuyển sang phải):

Nếu $x_1 < x_2$: $h(n') = |x_1 + 1 - x_2| + |y_1 - y_2| = |x_1 - x_2| - 1 + |y_1 - y_2| = h(n) - 1$. Do đó, $h(n) = h(n') + 1 = c(n, n') + h(n')$.

Nếu $x_1 \geq x_2$: $h(n') = |x_1 + 1 - x_2| + |y_1 - y_2| = |x_1 - x_2| + 1 + |y_1 - y_2| = h(n) + 1$. Do đó, $h(n) = h(n') - 1 < c(n, n') + h(n')$.

Tương tự cho các trường hợp di chuyển khác, ta luôn có $h(n) \leq c(n, n') + h(n')$, chứng tỏ khoảng cách Manhattan là consistent.

- **Dễ tính toán:** Phép tính abs(x1-x2) + abs(y1-y2) rất đơn giản, giúp giảm thiểu chi phí tính toán trong quá trình thực thi thuật toán, đặc biệt quan trọng trong trò chơi thời gian thực.

```python
def heuristic(current, goal):
    """
    Heuristic function based on Manhattan distance

    Estimates the distance from current position to goal using Manhattan distance.
    Formula: |x1 - x2| + |y1 - y2|

    This is an admissible heuristic (never overestimates the actual cost), ensuring
    the A* algorithm finds the optimal path. Suitable for grid-based movement (up, down, left, right) in 2D space as in Pac-Man game.

    Parameters:
        current: Current position, as tuple (x, y)
        goal: Target position to reach, as tuple (x, y)

    Returns:
        int: Manhattan distance between the two positions
    """
    return abs(current[0] - goal[0]) + abs(current[1] - goal[1])
```

Listing 8: Hàm tính khoảng cách Manhattan

Hàm tính chi phí - Chi phí đồng đều: Hàm tính chi phí chi tiết để mô tả chính xác hành vi của Ma Đỏ trong trò chơi. Việc tính toán chi phí như sau:

```
def calculate_cost(current, next_node):
    """
    Function to calculate movement cost from current node to next node for Red Ghost

    This function provides the g_score value in the A* algorithm (actual cost to move from the starting position to the current position).

    Red Ghost is characterized by its aggression and determination:
    - Always finds the shortest path to Pac-Man
    - Doesn't care about risks or safety
    - Uses uniform cost function (each step = 1) to find the shortest path

    Uniform cost is appropriate for Red Ghost as it helps find paths with fewest steps,
    matching the direct pursuit character of Red Ghost in the Pac-Man game.

    Parameters:
        current: Current position, as tuple (x, y)
        next_node: Adjacent position being considered for movement, as tuple (x, y)

    Returns:
        int: Cost of moving from current position to adjacent position (always 1)
    """
    # Base cost for each movement step is 1
    # Simple cost function because Red Ghost doesn't care about factors
    # other than reaching Pac-Man as quickly as possible
    base_cost = 1

    # No additional costs for any type of movement
    # Different from other ghosts that might avoid dangerous areas or have more complex strategies
```

```
30      return base_cost
```

Listing 9: Hàm tính chi phí cho Ma Đỏ

Lý do sử dụng chi phí đồng đều cho hàm calculate_cost:

Hàm calculate_cost luôn trả về giá trị 1 cho mỗi bước di chuyển, thể hiện chi phí đồng đều. Sở dĩ đưa ra lựa chọn như vậy bởi vì các lý do sau:

- **Mô phỏng chính xác hành vi Ma Đỏ:** Trong trò chơi Pac-Man gốc, Ma Đỏ được thiết kế để truy đuổi Pac-Man một cách trực tiếp và quyết liệt, không e ngại bất kỳ khu vực nào trên bản đồ. Chi phí đồng đều phản ánh đúng tính cách này - Ma Đỏ coi mọi bước di chuyển là như nhau, không có sự ưu tiên hay né tránh.
- **Tối ưu hóa đường đi ngắn nhất:** Khi mọi bước di chuyển có chi phí như nhau, thuật toán A* sẽ tìm đường đi với số bước ít nhất. Đây chính xác là hành vi mong muốn cho Ma Đỏ - luôn tìm đường ngắn nhất để đến Pac-Man.

Triển khai đầy đủ: Toàn bộ quy trình tìm đường của Ma Đỏ được triển khai như sau:

```
1  def red_ghost_path(ghost_pos, pacman_pos, graph, blocked_positions=None):
2      """
3      Determine the path for Red Ghost to follow Pac-Man using A* algorithm
4
5      Red Ghost characteristics in Pac-Man:
6      - Very aggressive and direct in pursuing Pac-Man
7      - Always seeks the shortest path to catch Pac-Man
8      - Uses A* algorithm with Manhattan distance heuristic to optimize path
9
10     This function handles special cases:
11     1. Ghost is already at the same position as Pac-Man
12     2. No path to Pac-Man found (may be completely blocked)
13
14     Parameters:
15         ghost_pos: Current Red Ghost position, as tuple (x, y)
16         pacman_pos: Current Pac-Man position, as tuple (x, y)
17         graph: Graph representing maze (dictionary with position as key and
18                list of adjacent positions that can be moved to as value)
```

```
        blocked_positions: List of blocked positions (other ghosts or obstacles)
    ,
                           default is None

    Returns:
        list: List of positions along the path from Red Ghost to Pac-Man
              or empty list if no path found or already at Pac-Man's position
    """
    # Initialize empty list if no blocked positions
    if blocked_positions is None:
        blocked_positions = []

    # Check special case: Ghost already at the same position as Pac-Man
    if ghost_pos == pacman_pos:
        return []  # No path needed as already at destination

    # Perform A* search to find optimal path
    result = astar_search(ghost_pos, pacman_pos, graph, blocked_positions)

    # Process search result
    if result:
        # Print detailed information about the found path
        print(f"Red ghost path found")
        print("Path:", result['path'])  # List of positions along the path
        print("Nodes expanded:", result['nodes_expanded'])  # Number of nodes
    considered during search
        print("Time (ms):", round(result['time_ms'], 3))  # Execution time (
    rounded to 3 digits)
        print("Memory (KB):", round(result['memory_kb'], 2))  # Memory used (
    rounded to 2 digits)
        print("Path cost:", result['cost'])  # Total path cost
        return result['path']
    else:
        # Print message if no path found
        print(f"No path found from ghost{ghost_pos} to pacman{pacman_pos}.\n")
```

```
50          return []
```

Listing 10: Hàm tìm đường cho Ma Đỏ

Tóm lại, sự kết hợp giữa khoảng cách Manhattan làm heuristic và chi phí đồng đều cho mỗi bước di chuyển tạo ra một triển khai A* tối ưu cho Ma Đỏ, vừa đảm bảo tính chính xác của thuật toán (luôn tìm được đường đi tối ưu), vừa mô phỏng chính xác hành vi truy đuổi trực tiếp đặc trưng của Ma Đỏ trong trò chơi Pac-Man nguyên bản.

3 Experiments

3.1 BFS (Breadth-First-Search:

3.1.1 Phân tích hiệu suất và kết quả thực nghiệm

Để đánh giá hiệu suất của thuật toán BFS cho Blue ghost, nhóm đã thực hiện thử nghiệm với 5 trường hợp khác nhau, thay đổi vị trí của Ma Xanh và Pac-Man trong mê cung. Các thông số được ghi nhận bao gồm: thời gian thực thi, bộ nhớ sử dụng, số nút mở rộng và chi phí đường đi.

Bảng 1: Kết quả thực nghiệm BFS với Ma Xanh trong 5 trường hợp thử nghiệm

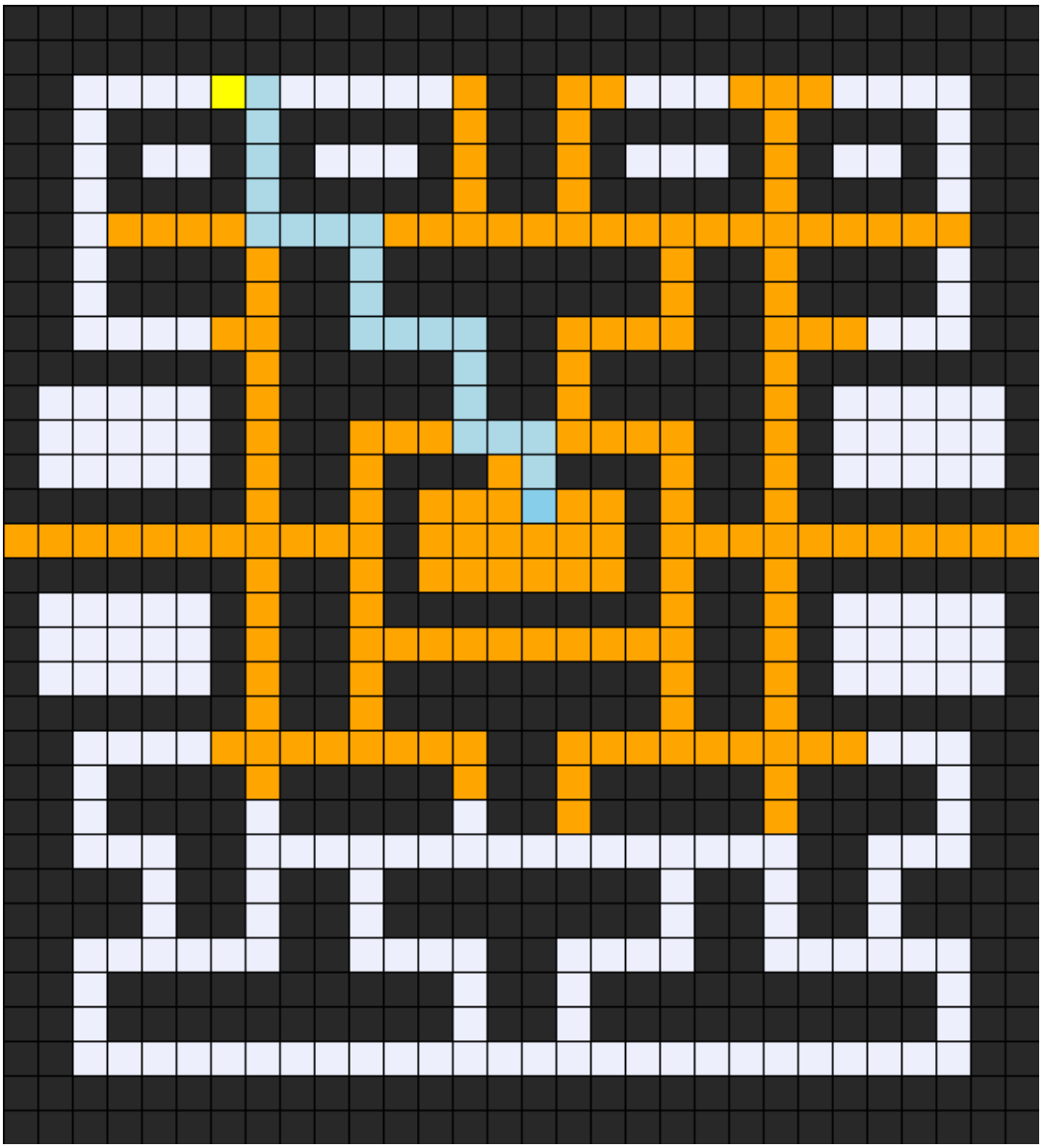
	Vị trí Ma Xanh	**Vị trí Pac-Man**	**Thời gian (ms)**	**Bộ nhớ (KB)**	**Số nút mở rộng**
1	(14, 15)	(2, 6)	1.095	15.08	186
2	(14, 15)	(30, 22)	1.907	45.33	317
3	(14, 15)	(20, 22)	0.825	15.08	137
4	(14, 15)	(27, 12)	1.577	15.78	285
5	(14, 15)	(2, 27)	1.511	14.63	246
TB	-	-	**1.383**	**21.18**	**234**

3.1.2 Hình ảnh minh họa đường đi trong các trường hợp thử nghiệm

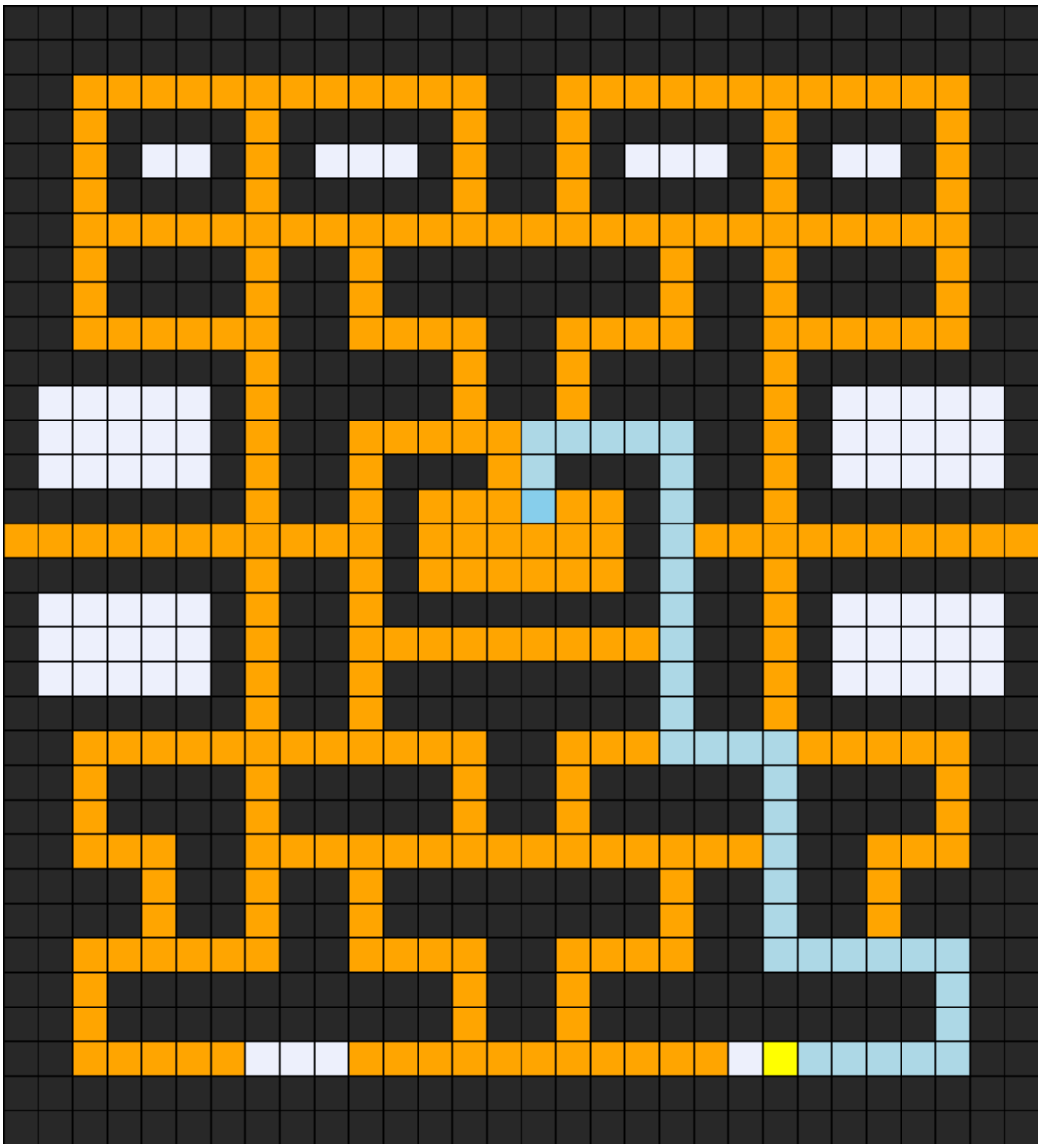
Hình 3 đến 7 minh họa kết quả thực nghiệm của thuật toán BFS cho Ma Xanh trong 5 trường hợp thử nghiệm. Trong mỗi hình: (phần vẽ này tham khảo của [1])

- Các ô màu cam biểu thị các nút đã được mở rộng (explored)
- Đường màu xanh nhạt biểu thị đường đi tối ưu cuối cùng từ Ma Xanh đến Pac-Man
- Ô màu xanh biểu thị vị trí Ma Xanh

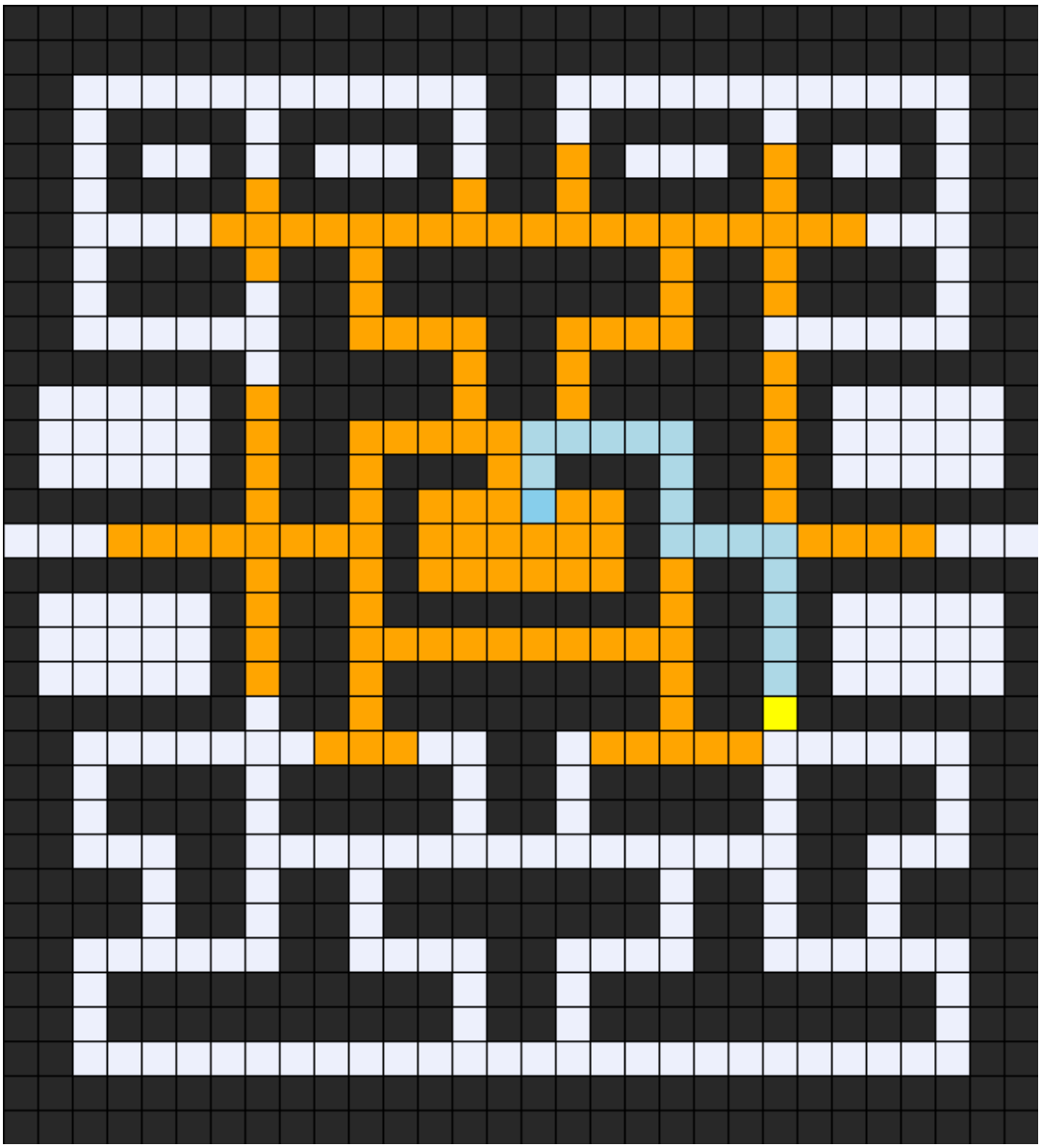
- Ô màu vàng biểu thị vị trí Pac-Man



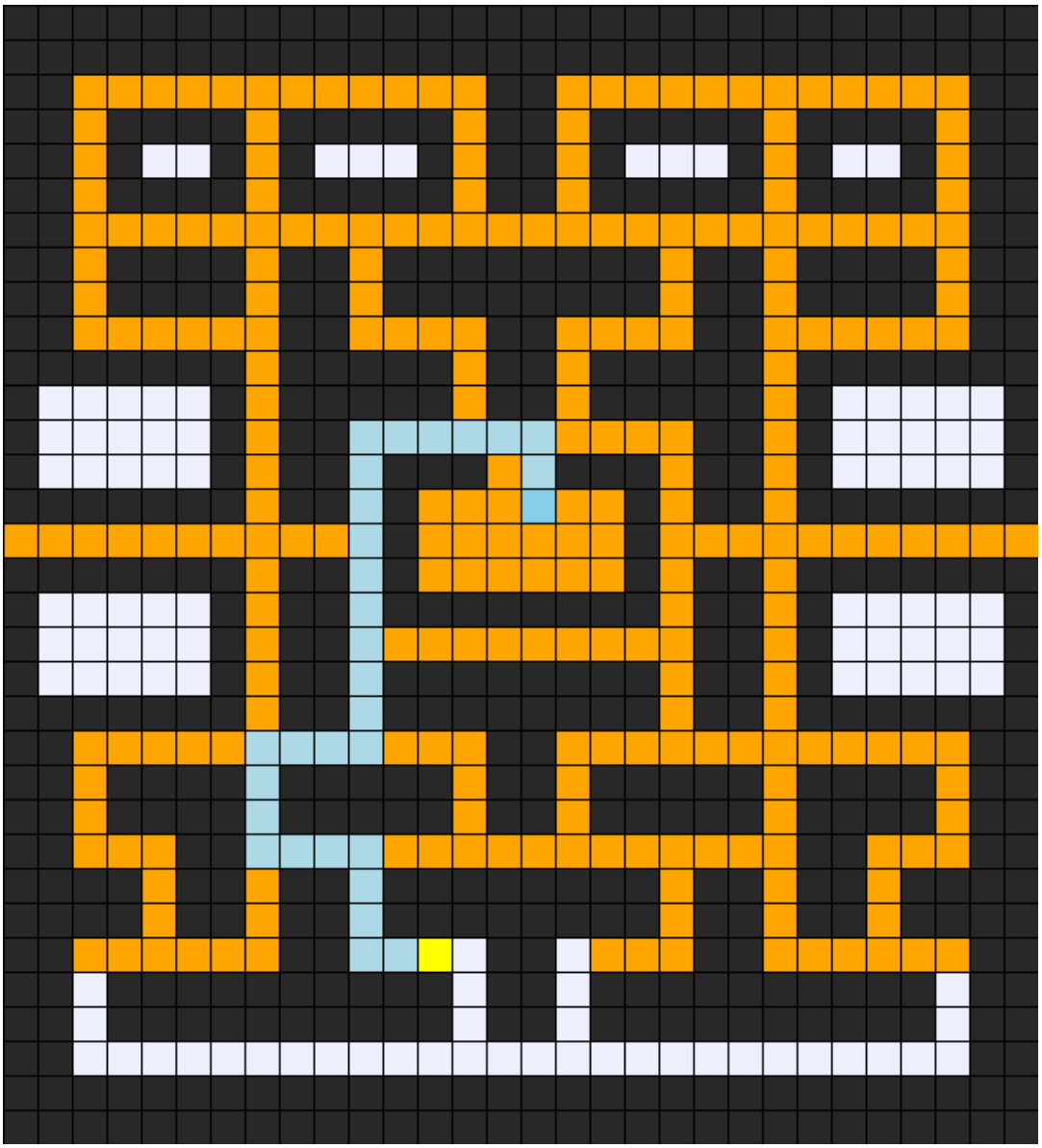
Hình 3: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 1: Ma Xanh (14, 15), Pac-Man (2, 6)



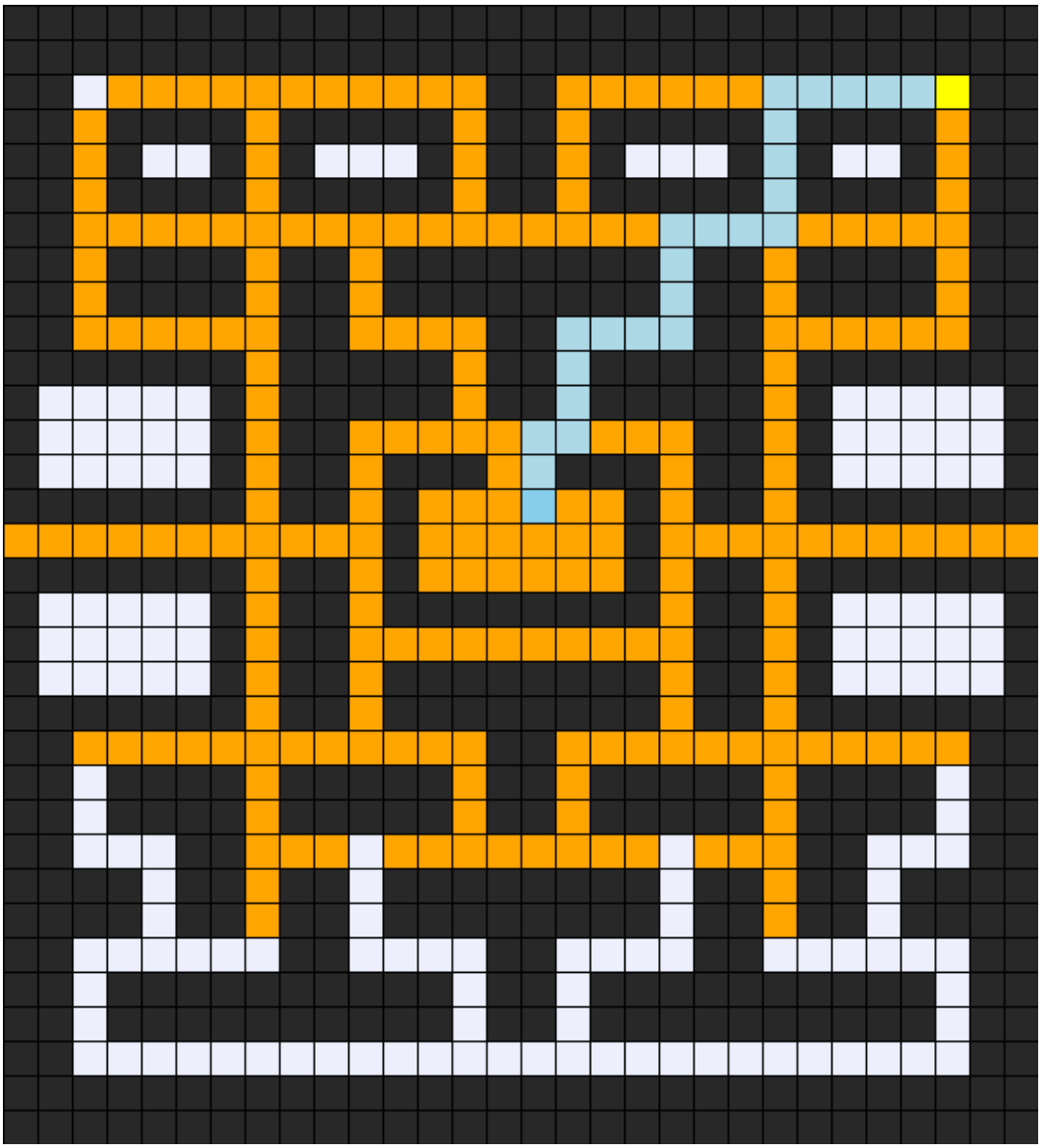
Hình 4: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 2: Ma Xanh (14, 15), Pac-Man (30, 22)



Hình 5: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 3: Ma Xanh (14, 15), Pac-Man (20, 22)



Hình 6: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 4: Ma Xanh (14, 15), Pac-Man (27, 12)



Hình 7: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 5: Ma Xanh (14, 15), Pac-Man (2, 27)

Phân tích kết quả:

- **Thời gian thực thi:** BFS cho kết quả thời gian thực thi khá nhanh, trung bình khoảng 1.383 ms, cho thấy BFS có thể áp dụng hiệu quả trong môi trường thời gian thực, nếu không yêu cầu tối ưu đường đi.
- **Bộ nhớ sử dụng:** Dù trong phần lớn trường hợp, bộ nhớ được giữ ở mức thấp (15 KB), nhưng có một trường hợp (test 2) vượt trội với 45.33 KB, cho thấy BFS có thể tiêu tốn nhiều bộ nhớ nếu khoảng cách giữa Ma Xanh và Pac-Man lớn hoặc không gian tìm kiếm rộng.

- **Số nút mở rộng:** Trung bình khoảng 234 nút, BFS mở rộng nhiều trạng thái do phải duyệt theo chiều rộng, không ưu tiên hướng nào, nên dễ dẫn đến mở rộng không cần thiết nếu đích ở xa hoặc mê cung phức tạp.

- **Độ hiệu quả:** Vì BFS mở rộng đều các hướng, nên đường đi không tối ưu về chi phí thời gian nếu mê cung lớn hoặc có nhiều nhánh rẽ. Vị trí Pac-Man càng xa (như test 2), BFS càng mất nhiều thời gian và bộ nhớ – điều này phù hợp với tính chất BFS là mở rộng theo lớp.

  Thuật toán BFS luôn tìm ra đường đi tối ưu với chi phí thấp nhất giữa Ma Xanh và Pac-Man trong tất cả các trường hợp thử nghiệm Vì chi phí giữa các nút là bằng nhau. Minh chứng qua các hình ảnh trực quan từ Hình 3 đến 7.

3.1.3 Kết luận

BFS là lựa chọn hợp lý cho Ma Xanh, đặc biệt khi cần đảm bảo đường đi ngắn nhất.
Tuy nhiên, BFS mở rộng khá nhiều node, khiến bộ nhớ tăng – nên cẩn thận nếu dùng cho bản đồ lớn hơn.
Trong môi trường game thực tế, BFS có thể cân bằng tốt giữa tốc độ và độ chính xác, phù hợp cho ghost đơn giản như Ma Xanh.

3.2 DFS (Depth-First Search)

3.2.1 Phân tích hiệu suất và kết quả thực nghiệm

Để đánh giá hiệu suất của thuật toán DFS cho Pink ghost, nhóm đã thực hiện thử nghiệm với 5 trường hợp khác nhau, thay đổi vị trí của Ma Hồng và Pac-Man trong mê cung. Các thông số được ghi nhận bao gồm: thời gian thực thi, bộ nhớ sử dụng, số nút mở rộng và chi phí đường đi.

Bảng 2: Kết quả thực nghiệm DFS với Ma Hồng trong 5 trường hợp thử nghiệm

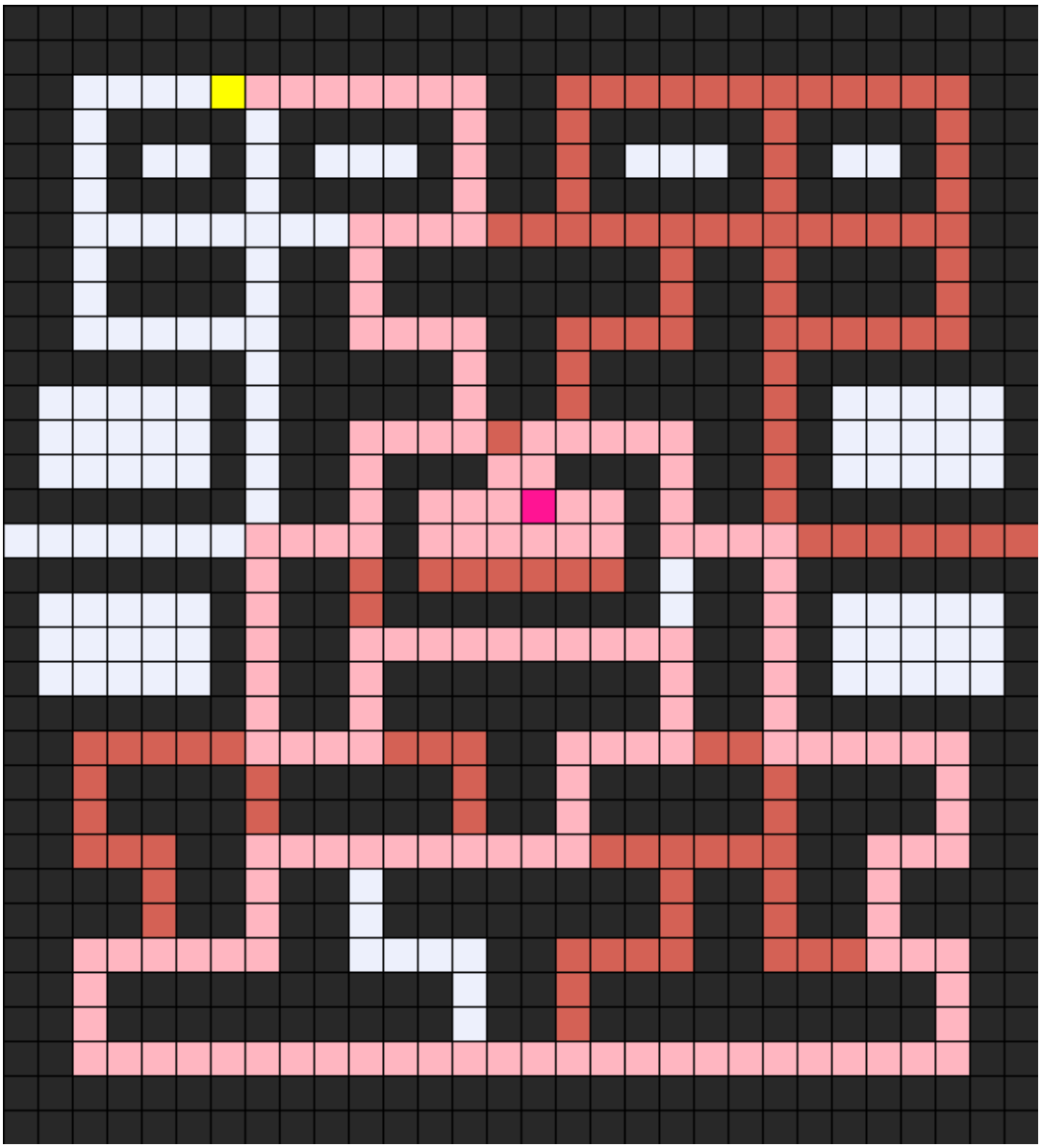
	Vị trí Ma Hồng	**Vị trí Pac-Man**	**Thời gian (ms)**	**Bộ nhớ (KB)**	**Nodes expanded**
1	(14, 15)	(2, 6)	1.55663	33.3281	271
2	(14, 15)	(30, 22)	1.00112	6.5703	66
3	(14, 15)	(20, 22)	1.05667	5.4688	42
4	(14, 15)	(27, 12)	2.44379	53.0391	318
5	(14, 15)	(2, 27)	2.35915	34.3672	235
TB	-	-	**1.68387**	**26.5547**	**186.4**

3.2.2 Hình ảnh minh họa đường đi trong các trường hợp thử nghiệm

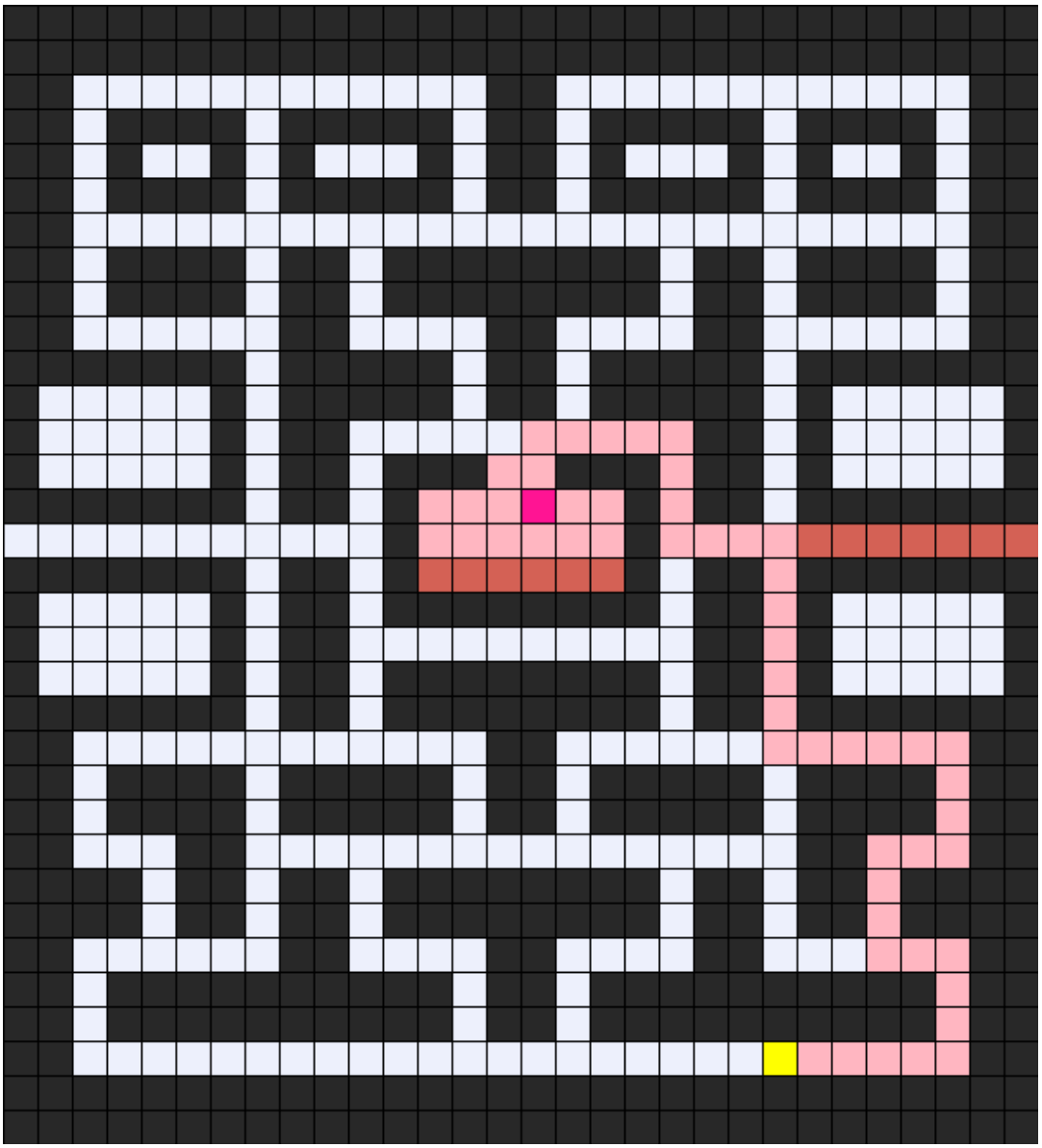
Hình 8 đến 12 minh họa kết quả thực nghiệm của thuật toán DFS cho Ma Hồng trong 5 trường hợp thử nghiệm. Trong mỗi hình: (phần vẽ này tham khảo của [1])

- Các ô màu đỏ biểu thị các nút đã được mở rộng (expanded)
- Đường màu hồng nhạt biểu thị đường đi tối ưu cuối cùng từ Ma Hồng đến Pac-Man
- Ô màu hồng biểu thị vị trí Ma Hồng
- Ô màu vàng đậm biểu thị vị trí Pac-Man

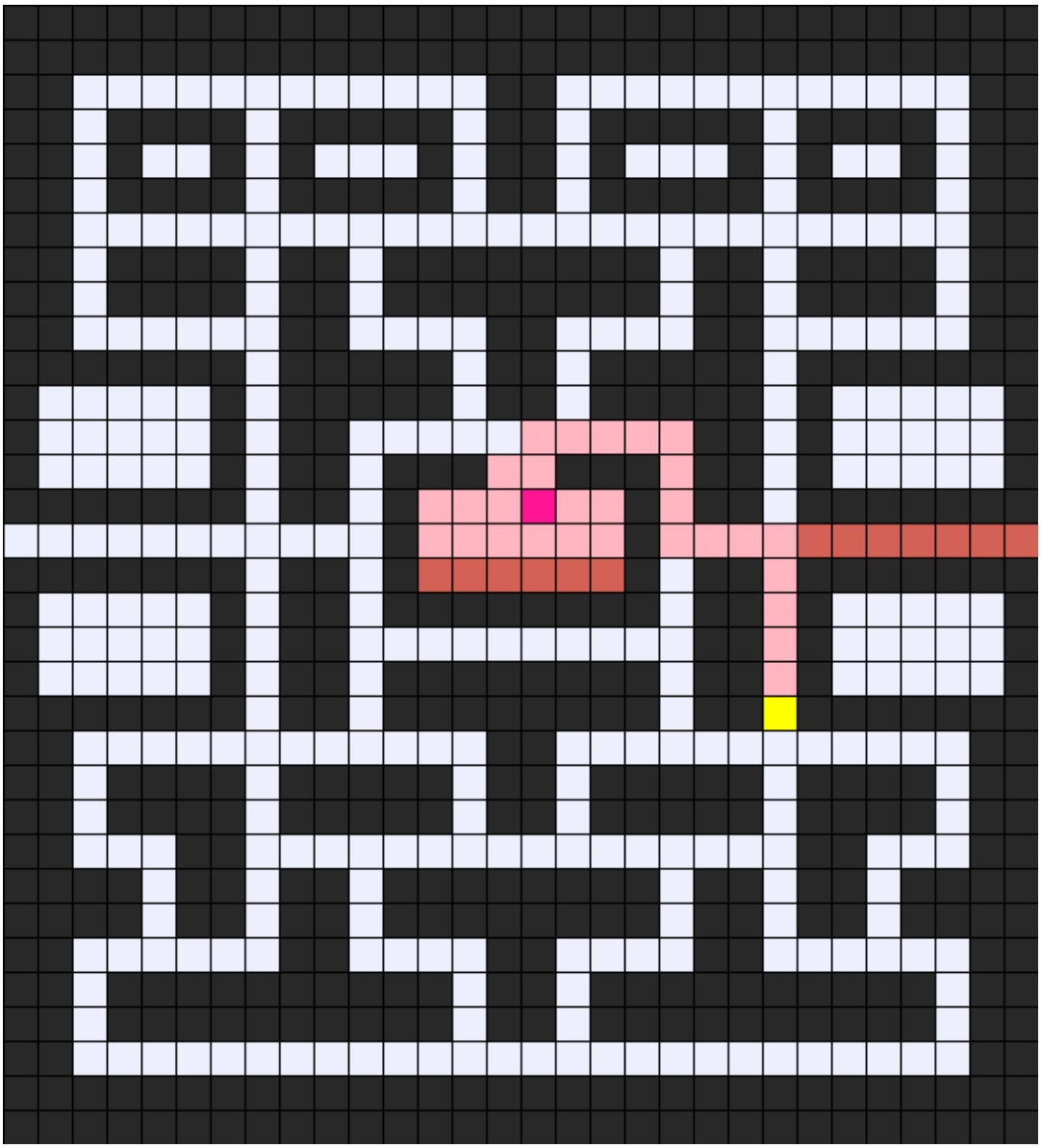
Chú ý: *các trạng thái con được duyệt theo thứ tự: phải, trái, xuống, lên.*



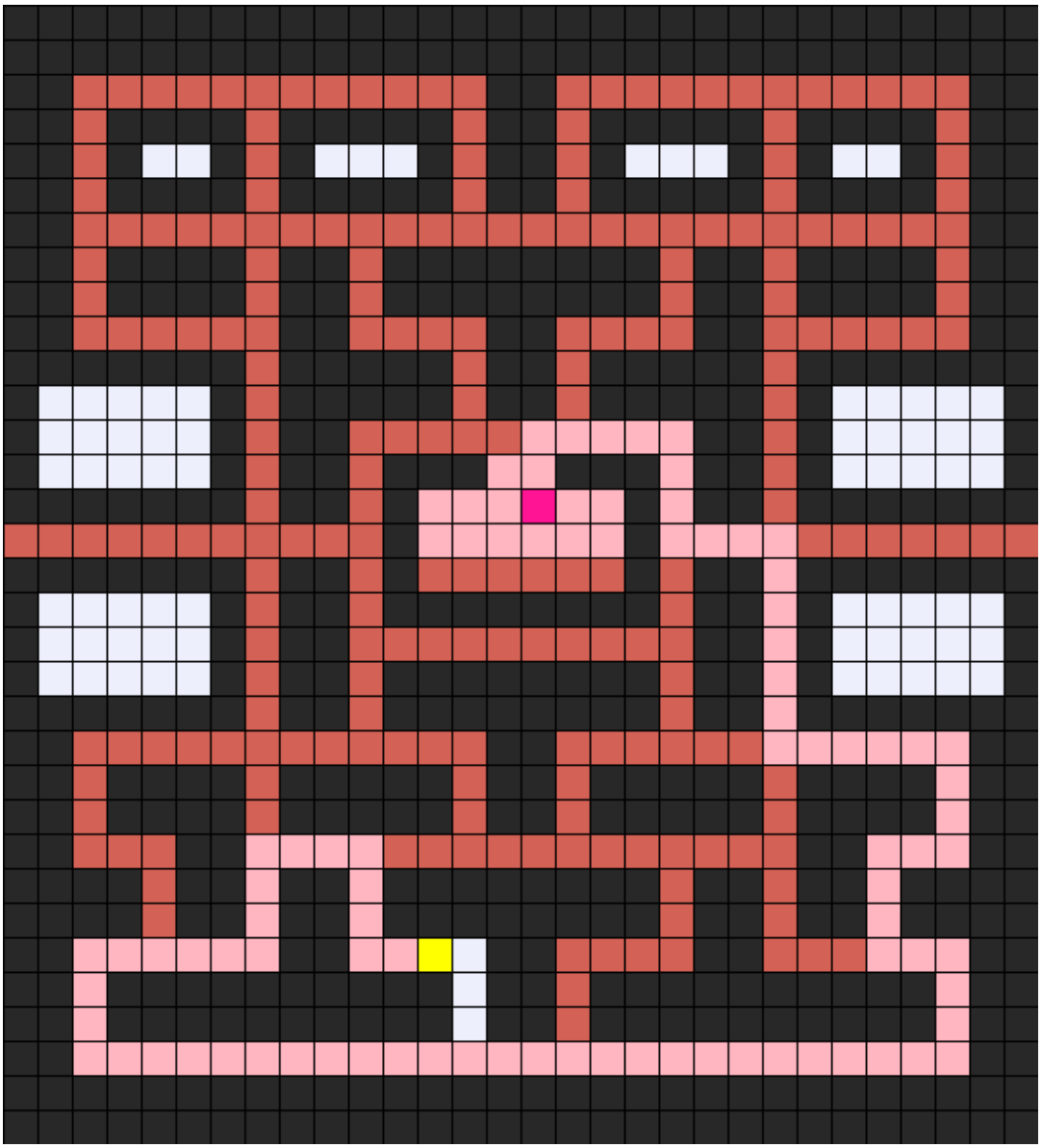
Hình 8: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 1: Ma Hồng (14, 15), Pac-Man (2, 6)



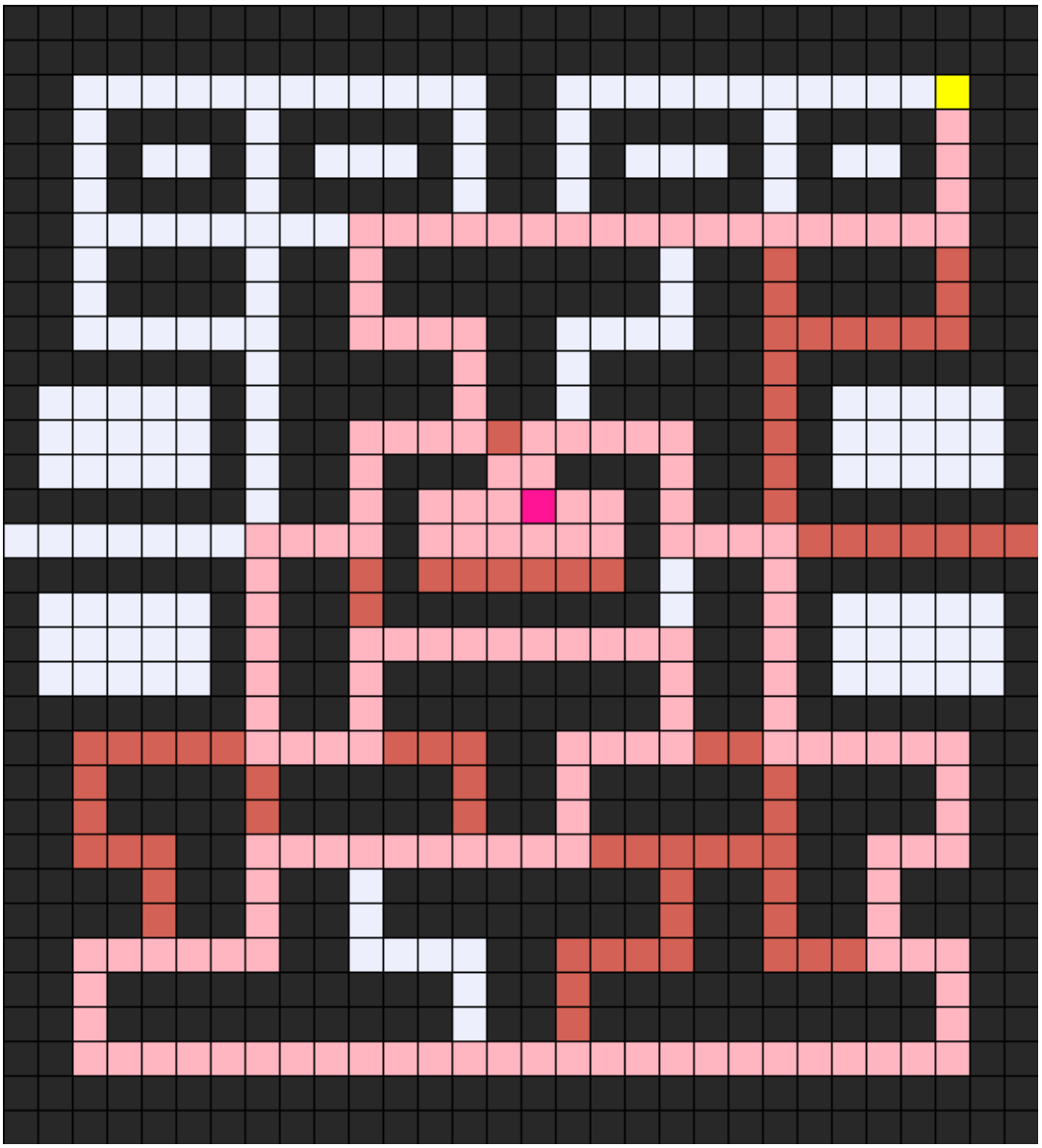
Hình 9: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 2: Ma Hồng (14, 15), Pac-Man (30, 22)



Hình 10: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 3: Ma Hồng (14, 15), Pac-Man (20, 22)



Hình 11: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 4: Ma Hồng (14, 15), Pac-Man (27, 12)



Hình 12: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 5: Ma Hồng (14, 15), Pac-Man (2, 27)

Phân tích kết quả:

- **Hiệu suất thời gian và bộ nhớ không ổn định:**

  Mặc dù DFS đôi khi cho kết quả nhanh và ít mở rộng (như ở Test 3: chỉ mở 42 node, 1.05 ms), nhưng có những trường hợp DFS mở rộng quá nhiều (Test 4: 318 nodes, 2.44 ms).

  Điều này cho thấy DFS rất phụ thuộc vào vị trí tương đối giữa Ma Hồng và Pac-Man, cũng như thứ tự duyệt node (phải → trái → dưới → trên).

- **Chiến lược tìm kiếm không đảm bảo tối ưu:**

  DFS không đảm bảo tìm đường đi ngắn nhất, và dễ bị lạc vào nhánh sâu không cần thiết — điều này phản ánh rõ ở Test 4 và Test 5 (minh chứng qua các hình ảnh trực quan từ Hình 11 đến 12.) với số node mở rộng lớn, dù Pac-Man không ở quá xa.

- **Bộ nhớ tiêu tốn không đều:** DFS sử dụng stack nên nhìn chung tiết kiệm bộ nhớ hơn so với BFS, nhưng vẫn có trường hợp tiêu tốn khá nhiều (ví dụ Test 4: 53 KB).

3.2.3 Kết luận

DFS phù hợp với các mê cung rộng, ít rẽ nhánh và có Pac-Man nằm gần theo hướng duyệt đầu tiên.

Tuy nhiên, DFS thiếu ổn định, không tối ưu hóa đường đi, và có thể mở rộng rất nhiều node nếu đi sai hướng ban đầu.

Với trò chơi như Pac-Man, nên kết hợp DFS với giới hạn độ sâu (IDS) hoặc chuyển sang thuật toán tìm kiếm có hướng (như A* hoặc UCS) để cải thiện hiệu suất và chất lượng đường đi.

3.3 Uniform-Cost Search (UCS):

3.3.1 Phân tích hiệu suất và kết quả thực nghiệm

Để đánh giá hiệu suất của thuật toán UCS cho Ma Cam, nhóm đã thực hiện thử nghiệm với 5 trường hợp khác nhau, thay đổi vị trí của Ma Cam và Pac-Man trong mê cung. Các thông số được ghi nhận bao gồm: thời gian thực thi, bộ nhớ sử dụng, số nút mở rộng và chi phí đường đi.

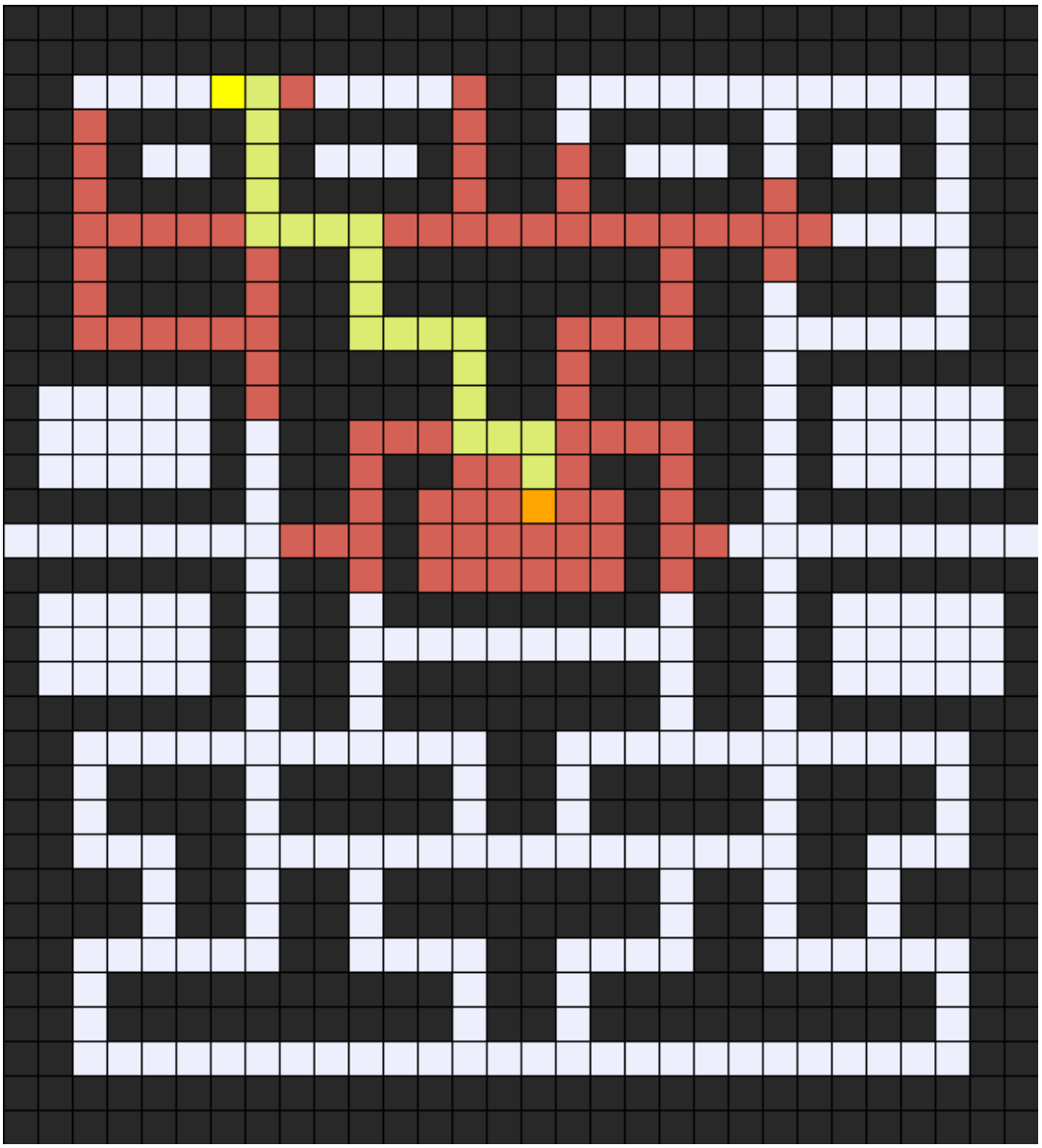
Bảng 3: Kết quả thực nghiệm UCS với Ma Cam trong 5 trường hợp thử nghiệm

	Vị trí Ma Cam	**Vị trí Pac-Man**	**Thời gian (ms)**	**Bộ nhớ (KB)**	**Số nút mở rộng**
1	(14, 15)	(2, 6)	2.543	14.52	107
2	(14, 15)	(30, 22)	5.592	24.09	192
3	(14, 15)	(20, 22)	2.177	13.11	92
4	(14, 15)	(27, 12)	5.328	27.82	219
5	(14, 15)	(2, 27)	2.565	12.66	101
TB	-	-	**3.641**	**18.84**	**142.2**

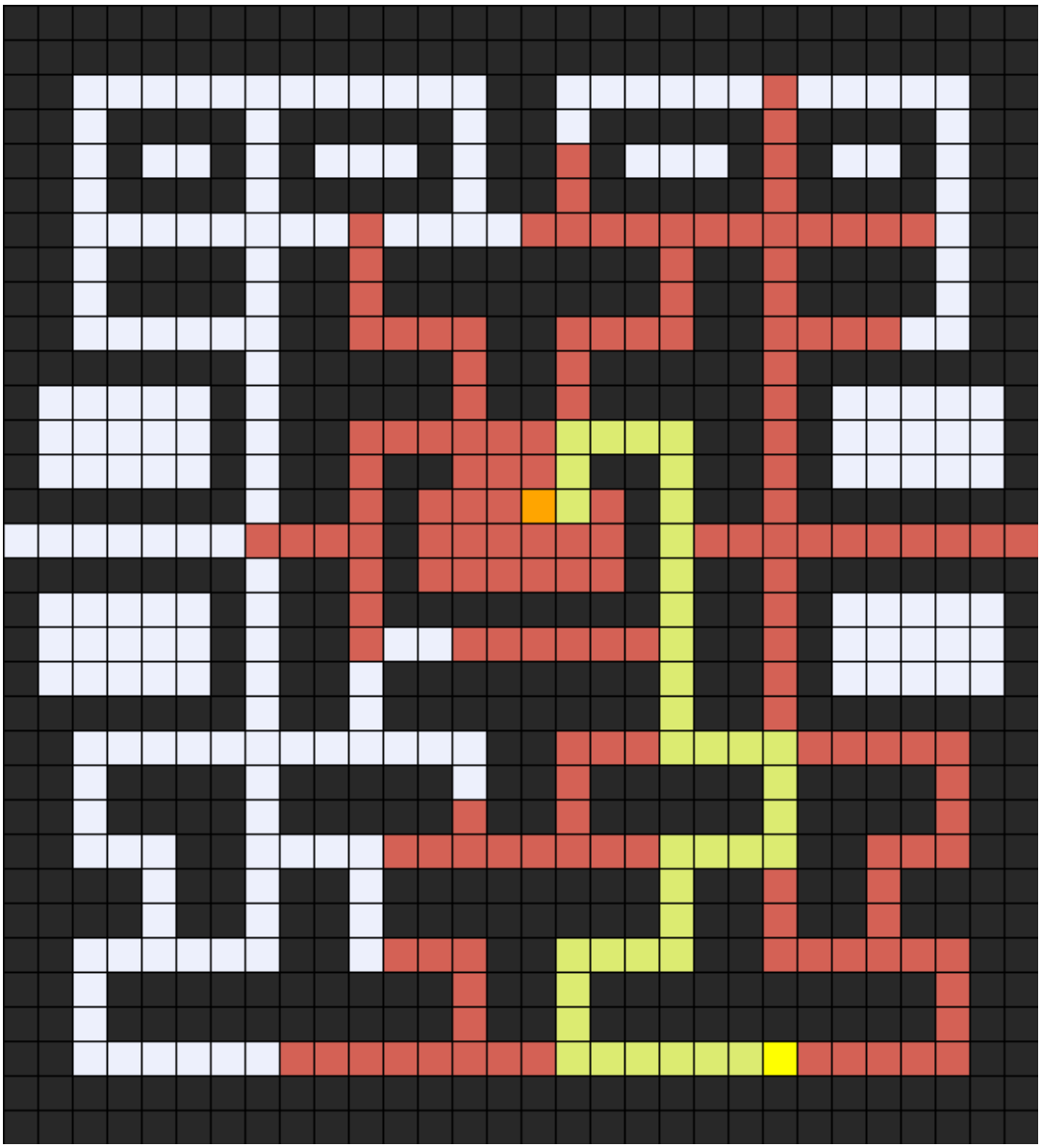
3.3.2 Hình ảnh minh họa đường đi trong các trường hợp thử nghiệm

Hình 13 đến 17 minh họa kết quả thực nghiệm của thuật toán UCS cho Ma Cam trong 5 trường hợp thử nghiệm. Trong mỗi hình: (phần vẽ này tham khảo của [1])

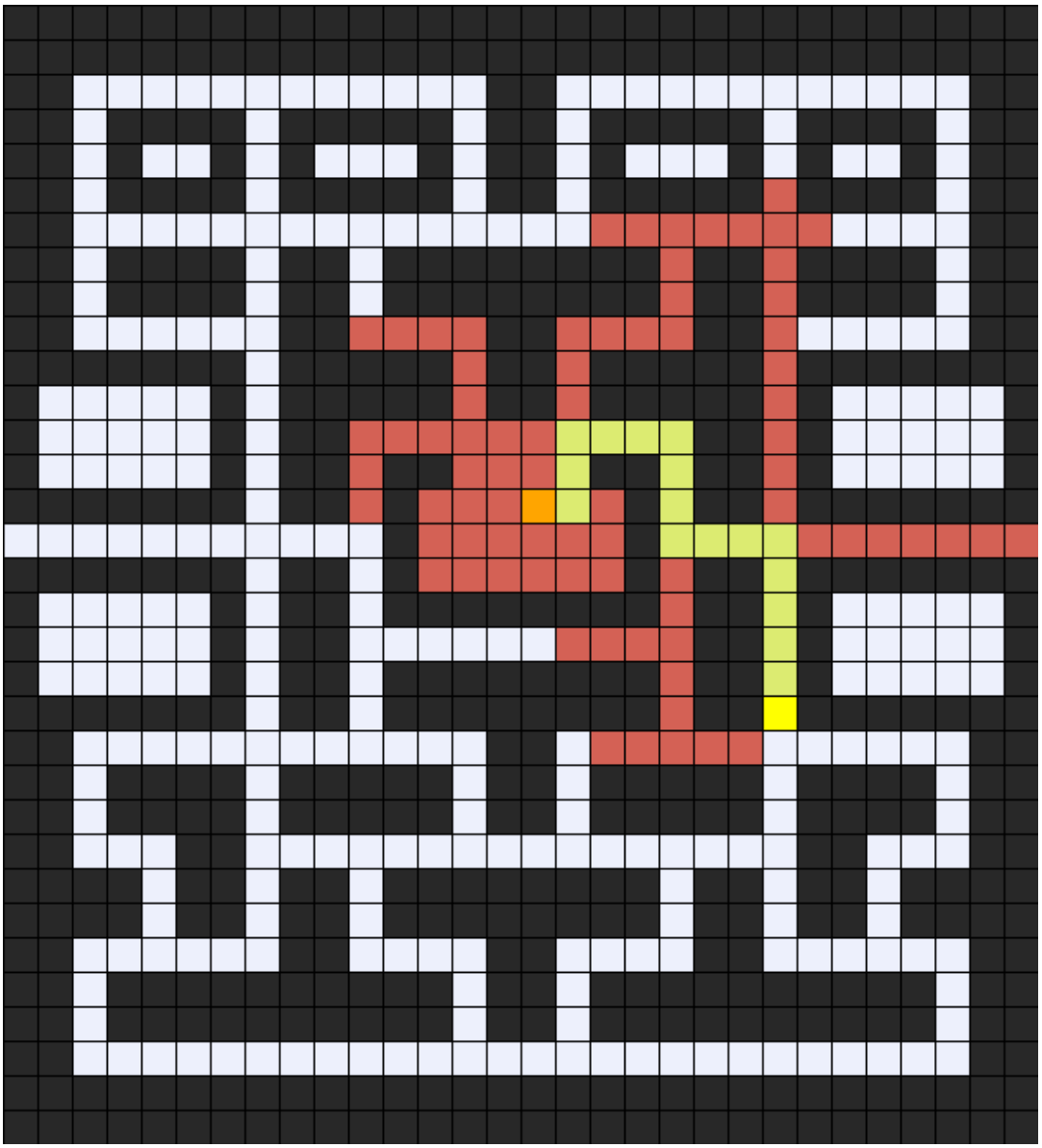
- Các ô màu đỏ biểu thị các nút đã được mở rộng (explored)
- Đường màu vàng nhạt biểu thị đường đi tối ưu cuối cùng từ Ma Cam đến Pac-Man
- Ô màu cam biểu thị vị trí Ma Cam
- Ô màu vàng đậm biểu thị vị trí Pac-Man



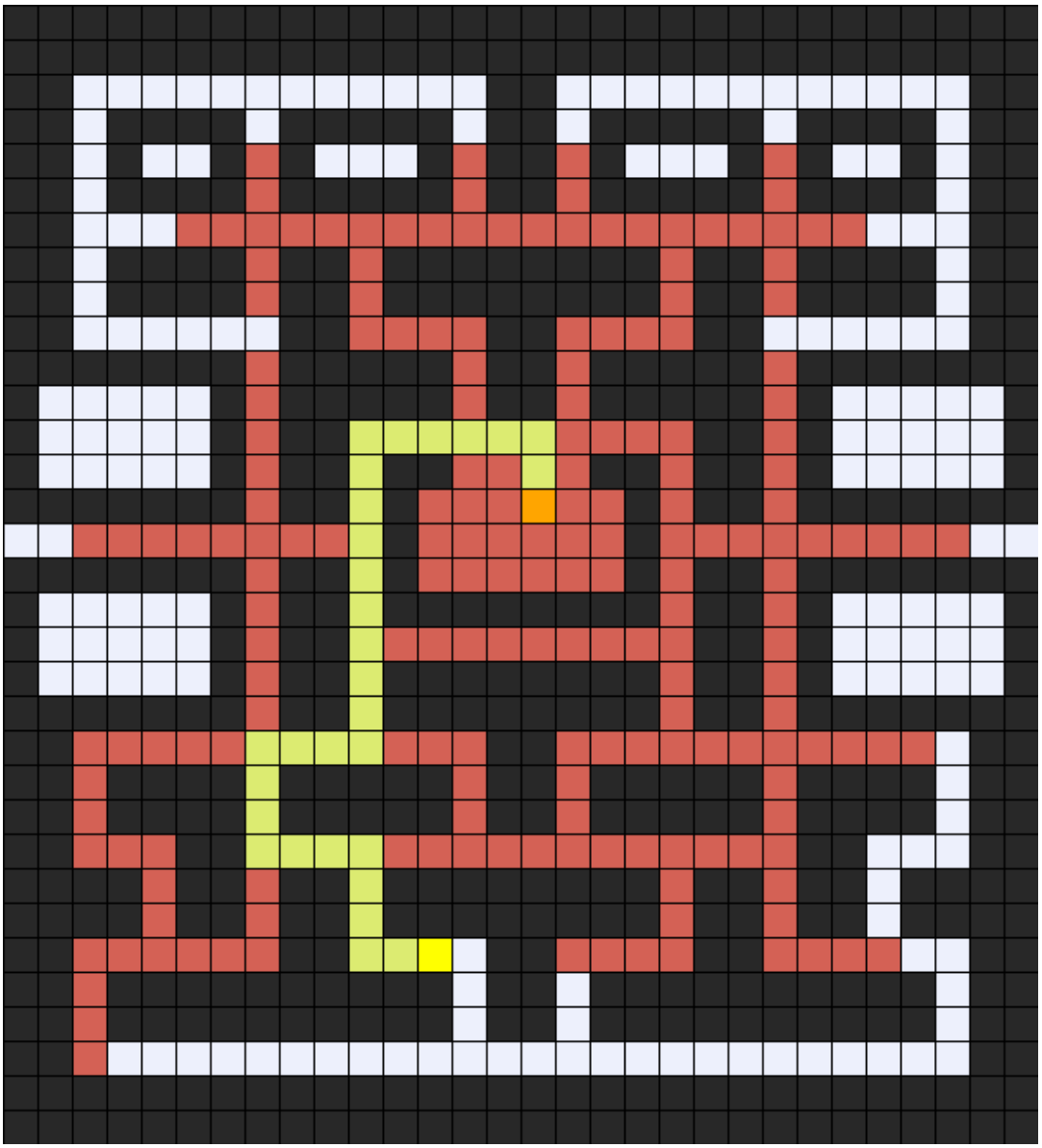
Hình 13: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 1: Ma Cam (14, 15), Pac-Man (2, 5)



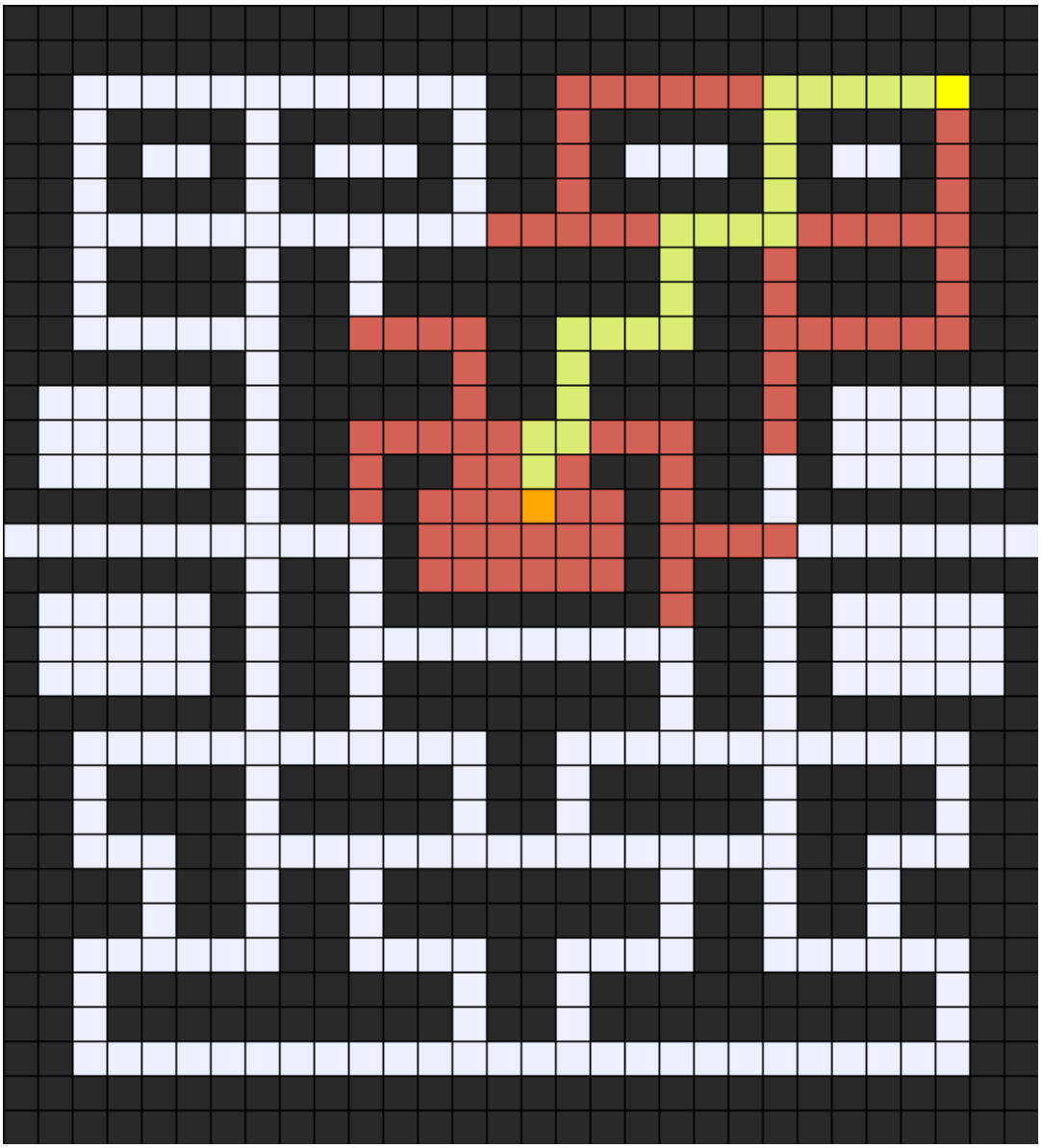
Hình 14: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 2: Ma Cam (14, 15), Pac-Man (30, 22)



Hình 15: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 3: Ma Cam (14, 15), Pac-Man (20, 22)



Hình 16: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 4: Ma Cam (14, 15), Pac-Man (27, 12)



Hình 17: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 5: Ma Cam (14, 15), Pac-Man (2, 27)

Phân tích kết quả:

- **Thời gian thực thi:** Trung bình 1.707 ms, với thời gian cao nhất là 2.615 ms (trường hợp 4) và thấp nhất là 1.008 ms (trường hợp 3). Thời gian thực thi có xu hướng tăng theo khoảng cách và độ phức tạp của đường đi.
- **Bộ nhớ sử dụng:** Trung bình 15.182 KB, với trường hợp sử dụng bộ nhớ nhiều nhất là trường hợp 4 (23.8 KB) khi Pac-Man ở vị trí (27, 12), có thể do cấu trúc mê cung phức tạp hơn trong khu vực đó.

- **Số nút mở rộng:** Trung bình 143 nút, cao nhất ở trường hợp 4 với 219 nút và thấp nhất ở trường hợp 3 với 92 nút. Số nút mở rộng có mối tương quan với bộ nhớ sử dụng và phản ánh mức độ tìm kiếm mà thuật toán phải thực hiện.
- **Độ hiệu quả:** Thuật toán UCS luôn tìm ra đường đi tối ưu với chi phí thấp nhất giữa Ma Cam và Pac-Man trong tất cả các trường hợp thử nghiệm, minh chứng qua các hình ảnh trực quan từ Hình 13 đến 17.

3.3.3 Kết luận

Thuật toán Uniform-Cost Search (UCS) thể hiện hiệu quả trong việc tìm đường đi tối ưu cho Ma Cam trong trò chơi Pac-Man. Với hàm chi phí tùy chỉnh, UCS cho phép mô phỏng hành vi đặc trưng của Ma Cam - di chuyển rụt rè, ưu tiên vùng ngoài mê cung.

3.4 A* Search (A-Star)

3.4.1 Phân tích hiệu suất và kết quả thực nghiệm

Để đánh giá hiệu suất của thuật toán A* cho Ma Đỏ, nhóm đã thực hiện thử nghiệm với 5 trường hợp giống như đã thử nghiệm với Ma Cam (UCS). Các thông số được ghi nhận bao gồm: thời gian thực thi, bộ nhớ sử dụng, số nút mở rộng và chi phí đường đi.

Bảng 4: Kết quả thực nghiệm A* với Ma Đỏ trong 5 trường hợp thử nghiệm

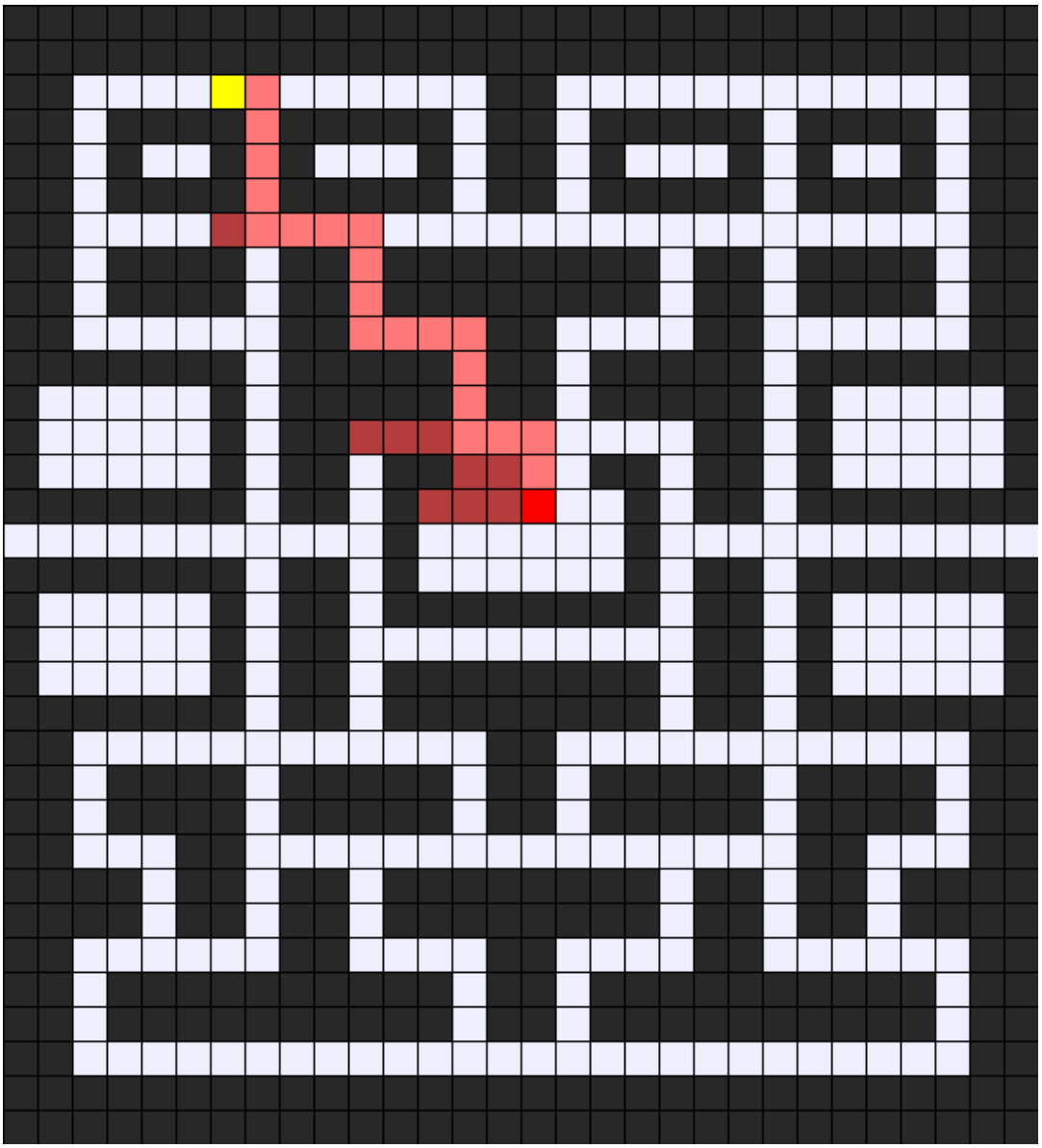
	Vị trí Ma Đỏ	**Vị trí Pac-Man**	**Thời gian (ms)**	**Bộ nhớ (KB)**	**Số nút mở rộng**
1	(14, 15)	(2, 6)	0.426	5.32	30
2	(14, 15)	(30, 22)	1.493	16.88	136
3	(14, 15)	(20, 22)	0.495	7.66	38
4	(14, 15)	(27, 12)	1.045	16.95	111
5	(14, 15)	(2, 27)	0.595	8.12	38
TB	-	-	**0.8108**	**10.586**	**71**

3.4.2 Hình ảnh minh họa đường đi trong các trường hợp thử nghiệm

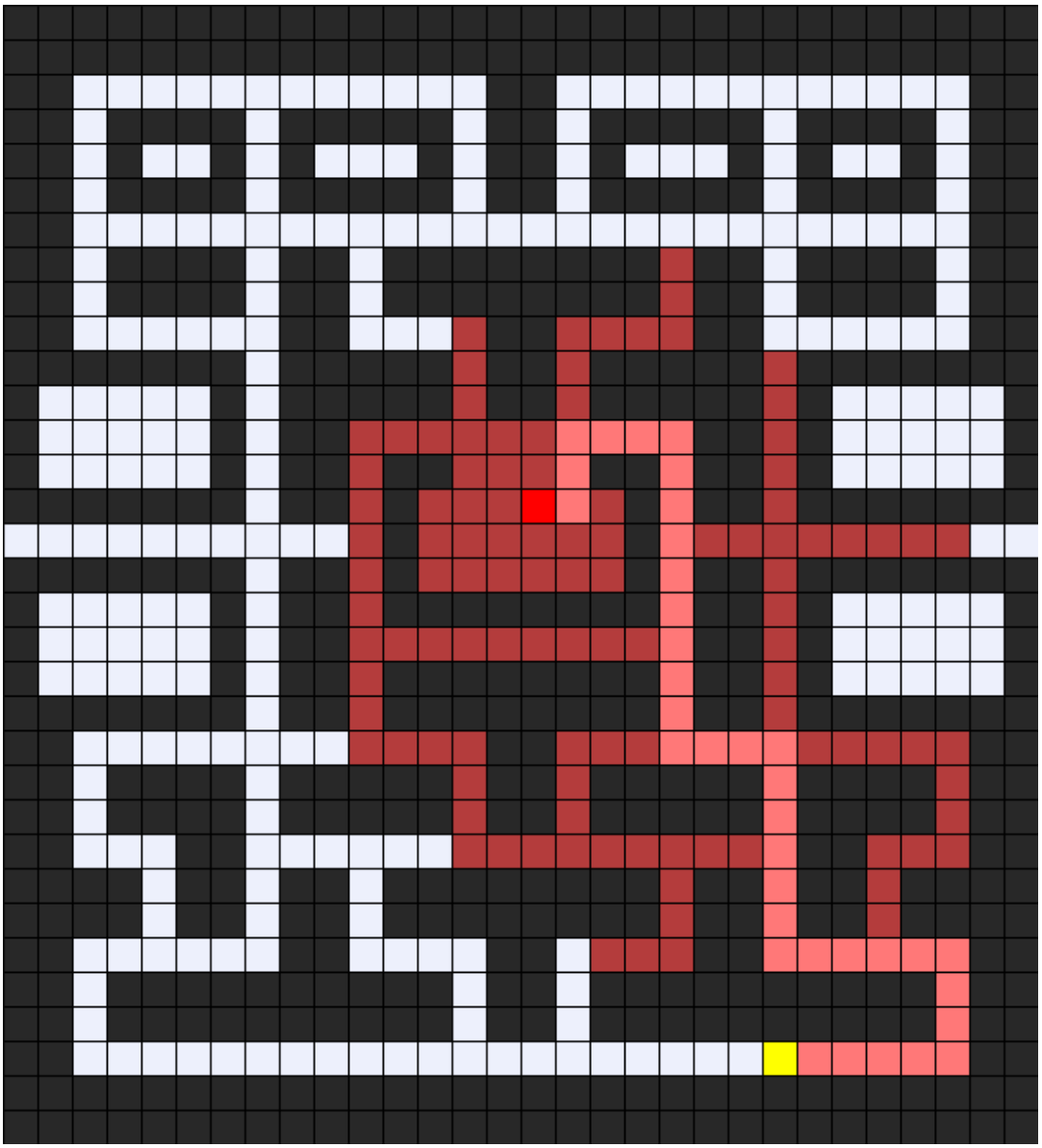
Hình 18 đến 22 minh họa kết quả thực nghiệm của thuật toán A* cho Ma Đỏ trong 5 trường hợp thử nghiệm. Trong mỗi hình: (phần vẽ này tham khảo của [1])

- Các ô màu đỏ đậm biểu thị các nút đã được khám phá (explored)

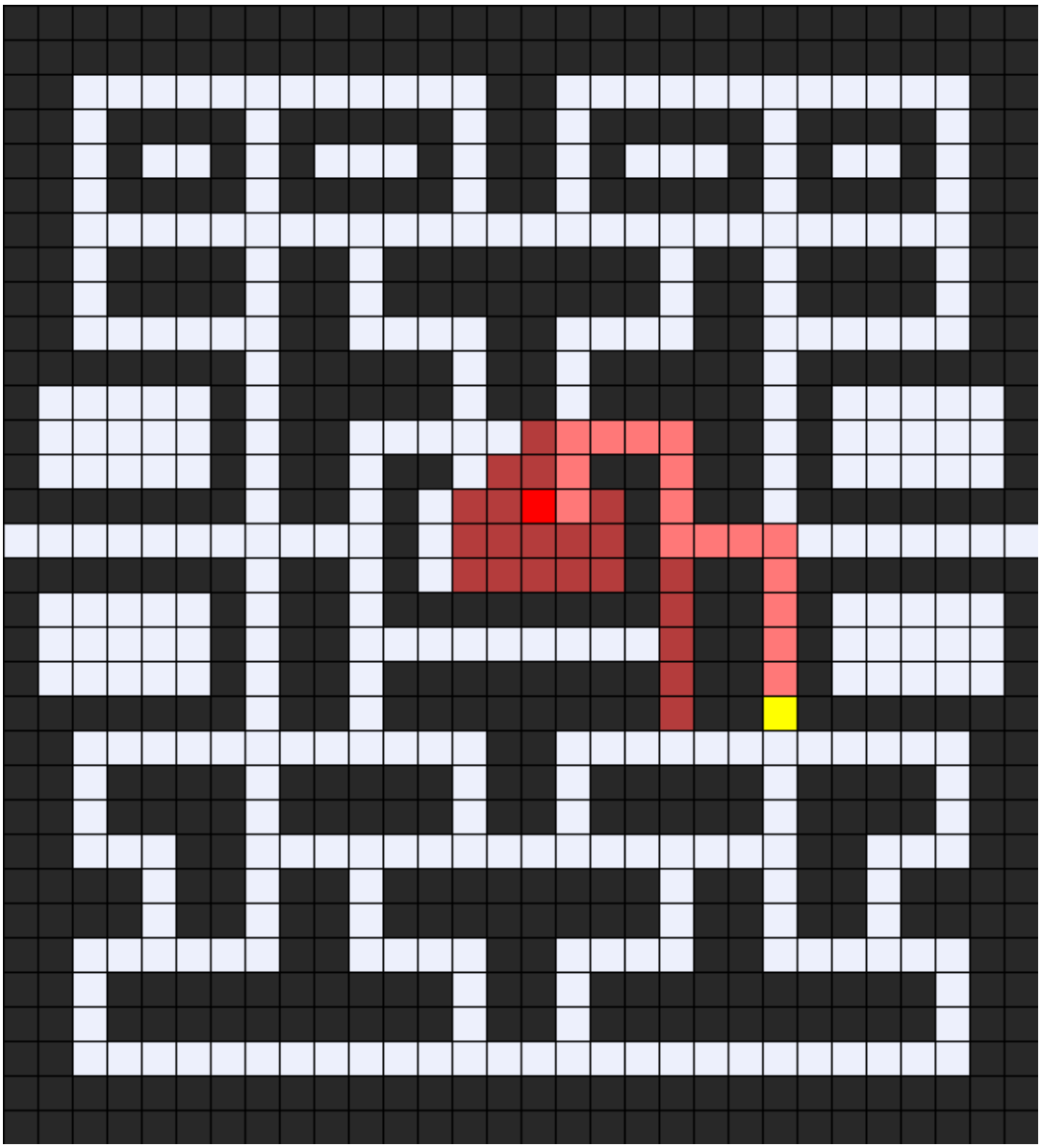
- Đường màu đỏ nhạt biểu thị đường đi tối ưu cuối cùng từ Ma Đỏ đến Pac-Man
- Ô màu đỏ biểu thị vị trí Ma Đỏ
- Ô màu vàng đậm biểu thị vị trí Pac-Man



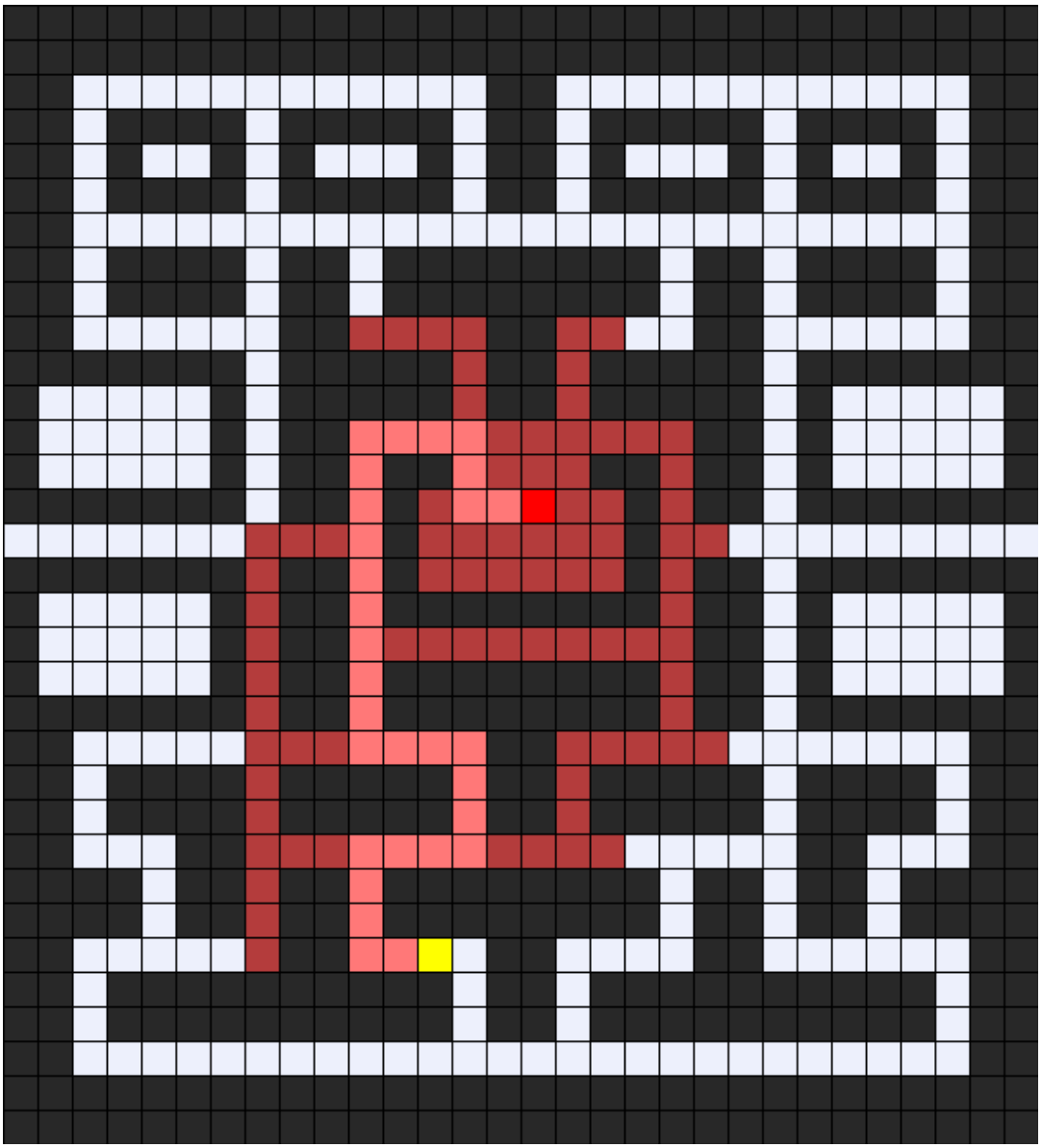
Hình 18: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 1: Ma Đỏ (14, 15), Pac-Man (2, 5)



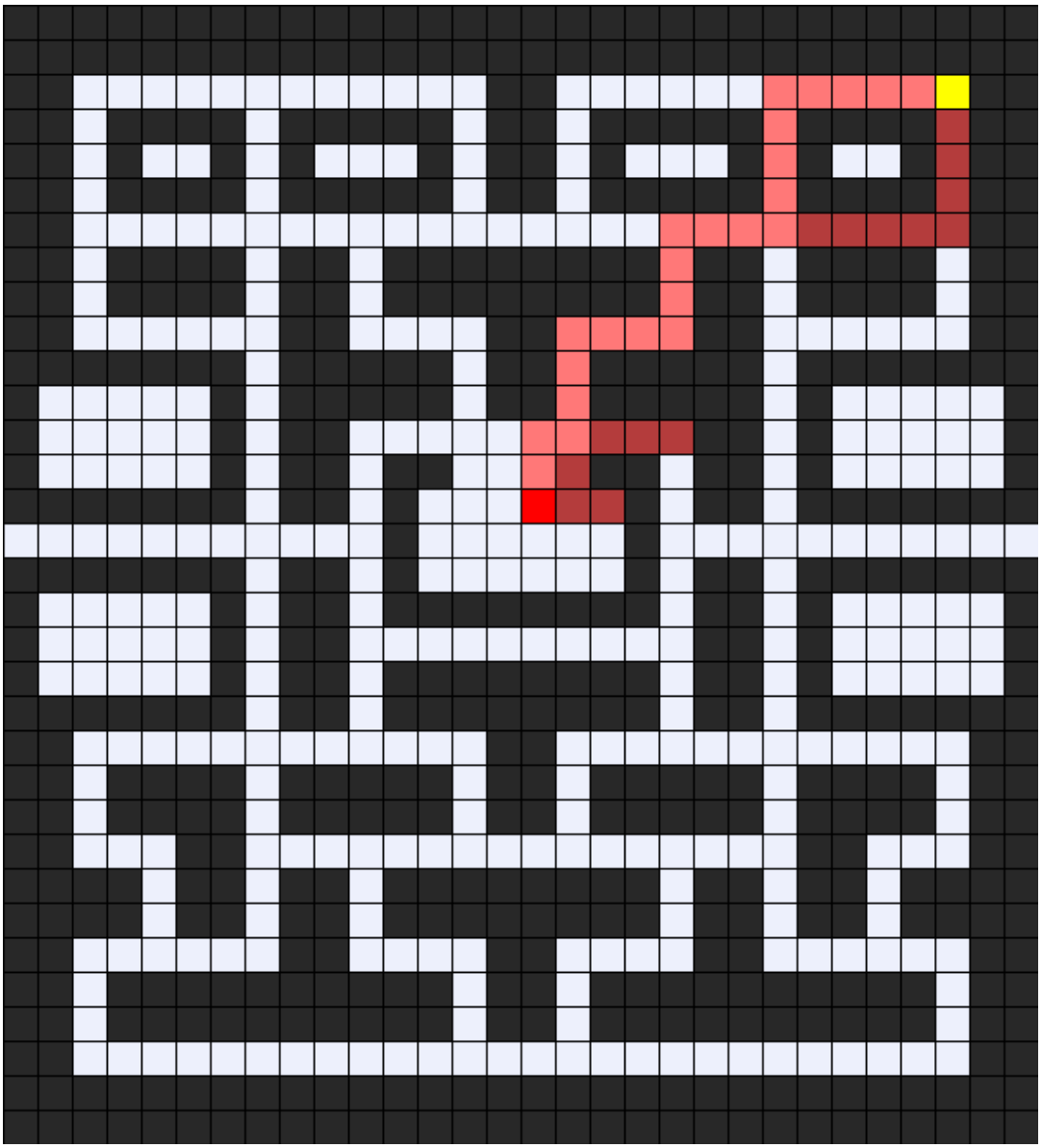
Hình 19: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 2: Ma Đỏ (14, 15), Pac-Man (30, 22)



Hình 20: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 3: Ma Đỏ (14, 15), Pac-Man (20, 22)



Hình 21: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 4: Ma Đỏ (14, 15), Pac-Man (27, 12)



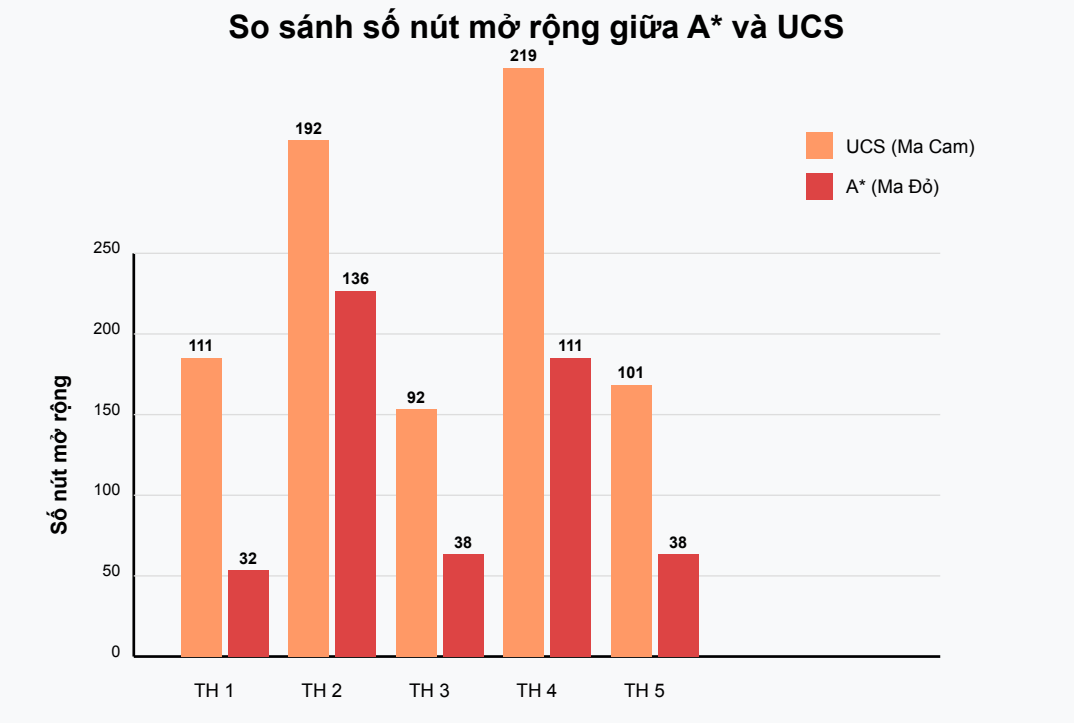
Hình 22: Minh họa đường đi và các nút đã mở rộng trong trường hợp thử nghiệm 5: Ma Đỏ (14, 15), Pac-Man (2, 27)

Phân tích kết quả:

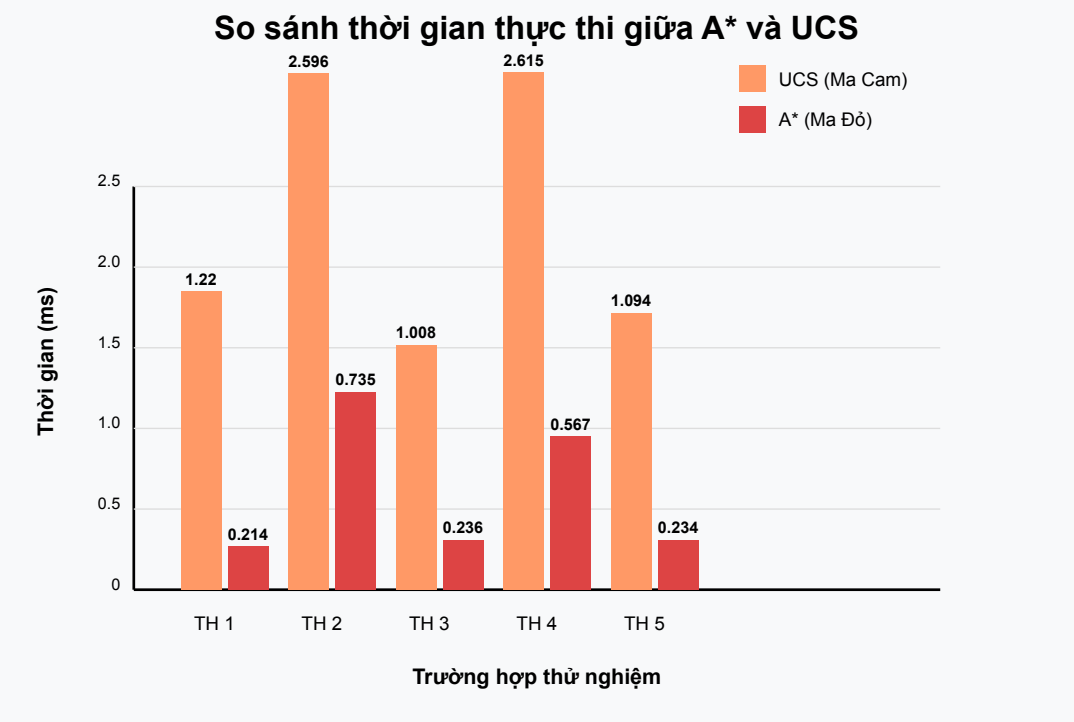
- **Thời gian thực thi:** Trung bình 0.3972 ms, nhanh hơn đáng kể so với UCS (1.707 ms), giảm khoảng 77%.
- **Bộ nhớ sử dụng:** Trung bình 11.126 KB, tiêu tốn ít bộ nhớ hơn UCS (15.182 KB), giảm khoảng 27%.
- **Số nút mở rộng:** Trung bình 71 nút, giảm khoảng 50% so với UCS (143 nút), cho thấy hiệu

quả vượt trội của hàm heuristic trong việc định hướng tìm kiếm.

- **Tính ổn định:** Trong tất cả các trường hợp thử nghiệm, A* luôn tìm ra đường đi tối ưu tương tự UCS, nhưng với hiệu suất cao hơn nhiều lần.



Hình 23: Biểu đồ so sánh số nút mở rộng giữa A* và UCS trong 5 trường hợp thử nghiệm



Hình 24: Biểu đồ so sánh thời gian thực thi giữa A* và UCS trong 5 trường hợp thử nghiệm

3.4.3 Kết luận

Thuật toán A* Search thể hiện hiệu quả vượt trội trong việc tìm đường đi cho Ma Đỏ trong trò chơi Pac-Man. Sự kết hợp giữa chi phí thực tế (g) và ước lượng heuristic (h) giúp A* định hướng quá trình tìm kiếm hiệu quả, giảm đáng kể số nút cần mở rộng đồng thời vẫn đảm bảo tìm ra đường đi tối ưu.

A* là một thuật toán hiệu quả kết hợp ưu điểm của thuật toán tìm kiếm có thông tin và đảm bảo tối ưu về đường đi, phù hợp nhất với đặc điểm của Ma Đỏ - luôn trực tiếp và thông minh trong việc săn đuổi Pac-Man.

3.5 Tổng kết:

Tổng so sánh kết quả trung bình của 4 thuật toán ta được:

Nhận xét:

- **A*** là thuật toán hiệu quả nhất cả về thời gian, bộ nhớ và số nút mở rộng, nhờ vào heuristic hướng dẫn quá trình tìm kiếm.

Bảng 5: So sánh hiệu suất giữa các thuật toán tìm đường

Thuật toán	Thời gian (ms)	Bộ nhớ (KB)	Số nút mở rộng	Tối ưu đường đi
DFS	1.684	26.55	186	Không
BFS	1.383	21.18	234	Có
UCS	3.641	18.84	142	Có
A*	**0.8108**	**10.586**	**71**	Có

- **UCS** cũng tìm được đường đi tối ưu nhưng thời gian cao hơn do không có heuristic hỗ trợ.
- **BFS** đơn giản, đảm bảo tối ưu đường đi nhưng tiêu tốn nhiều bộ nhớ và mở rộng nhiều node.
- **DFS** có thời gian thấp, nhưng không đảm bảo tìm được đường đi tối ưu và tiêu tốn bộ nhớ khi đi sâu.
- Trong môi trường thời gian thực như trò chơi Pac-Man, **A*** là lựa chọn tốt nhất cho Ma tìm Pacman nhờ hiệu quả cao và phản hồi nhanh.

4 Sử dụng tài liệu tham khảo

Thuật toán nhóm cài đặt giao diện cho game Pacman có tham khảo hướng dẫn trong video [4]

References

[1] CS50. *maze.py - CS50 AI 2020, Lecture 0*. Accessed: 2025-04-17. 2020. URL: https://cdn.cs50.net/ai/2020/spring/lectures/0/src0/maze.py.

[2] GeeksforGeeks contributors. *Breadth First Search (BFS) for a Graph*. Accessed April 15, 2025. n.d. URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/#bfs-from-a-given-source.

[3] GeeksforGeeks contributors. *Depth First Search (DFS) for a Graph*. Accessed April 15, 2025. n.d. URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/.

[4] LeMaster Tech. *How to Make Pac-Man in Python!* Accessed April 4, 2025. 2022. URL: https://www.youtube.com/watch?v=9H27CimgPsQ.