TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG – HCM
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN:
PROJECT 1: SEARCH
"Let's chase Pac-Man!"

Giảng viên: NGUYỄN TIẾN HUY

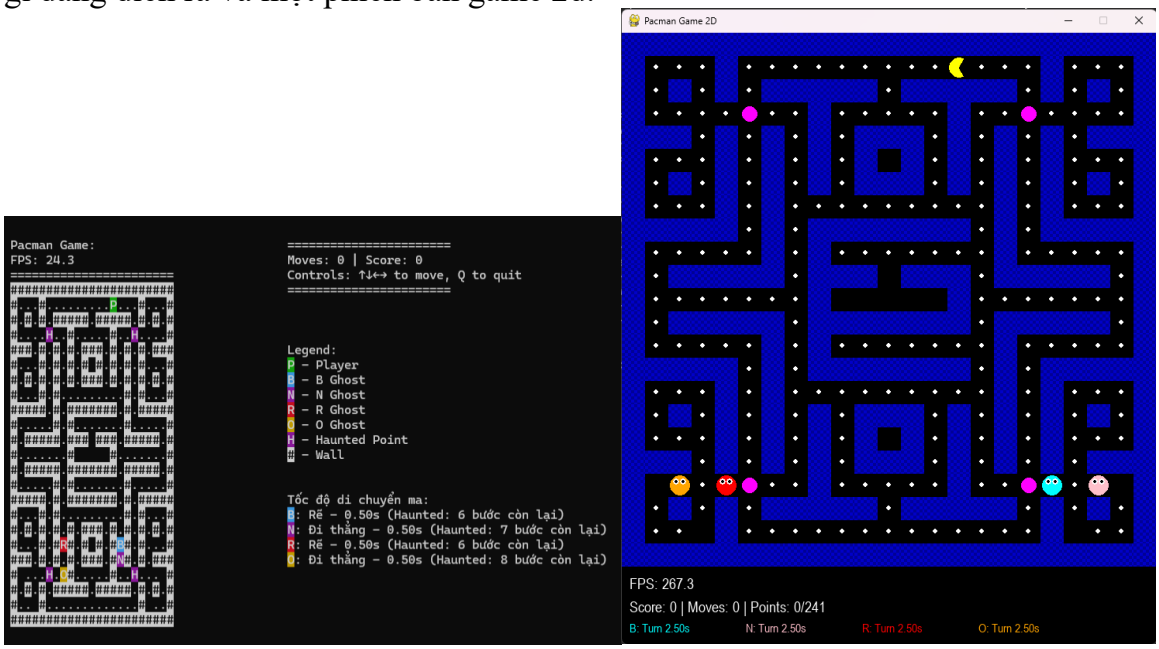
Họ và tên	MSSV
ĐINH TUẤN DUY	23127356
LỮ BẢO ĐẠT	23127338
BÙI ĐỨC ĐẠT	23127337
THIỀU QUANG VINH	23127143

TP. Hồ Chí Minh – 03/2025

Mục lục

1. GIỚI THIỆU

Pac-Man là một tựa game giải trí – nơi mà người chơi sẽ phải điều khiển nhân vật Pacman di chuyển khắp mê cung và thu thập nhiều chấm ghi điểm nhất có thể. Trò chơi sẽ kết thúc khi các chấm ghi điểm đã được thu thập hết hoặc khi Pac-Man bị con ma bắt được. Trong đồ án này, nhóm đã tạo ra một phiên bản đơn giản hơn của game bằng cách tối giản giao diện người dung (text-base game) để thể hiện trực quan nhất những gì đang diễn ra và một phiên bản game 2d.

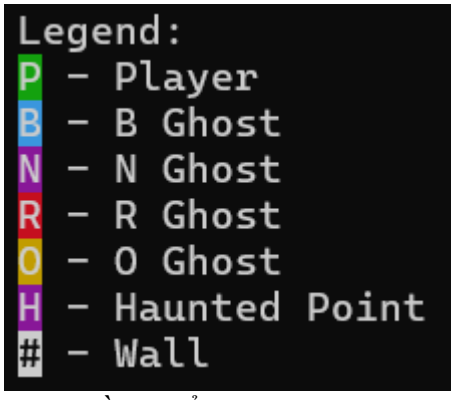


2. MÔ TẢ

2.1. Luật chơi

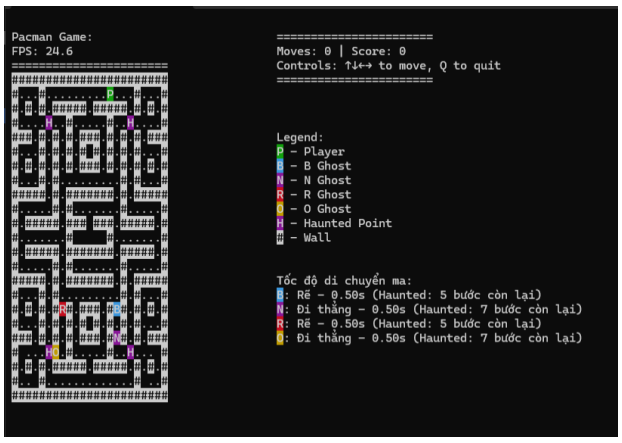
a) Người chơi sẽ di chuyển khắp mê cung để thu thập các chấm ghi điểm trong khi tránh chạm mặt với các con ma đang đuổi theo.
b) Trò chơi sẽ kết thúc khi các chấm ghi điểm đã hết hoặc khi người chơi đã bị ma bắt.
c) Mỗi con ma khi rẽ góc 90 độ hoặc quay lui 180 độ sẽ di chuyển chậm hơn đáng kể so với đi thẳng
d) Trên bản đồ sẽ xuất hiện những ô "ám". Các con ma bước vào ô sẽ được tăng tốc.
e) Người chơi sử dụng 4 phím WASD hoặc ←↑→↓ để di chuyển và ấn phím Q để đầu hàng. Mỗi lần sẽ di chuyển được 1 ô, tối thiểu 0.2 giây di chuyển một lần.

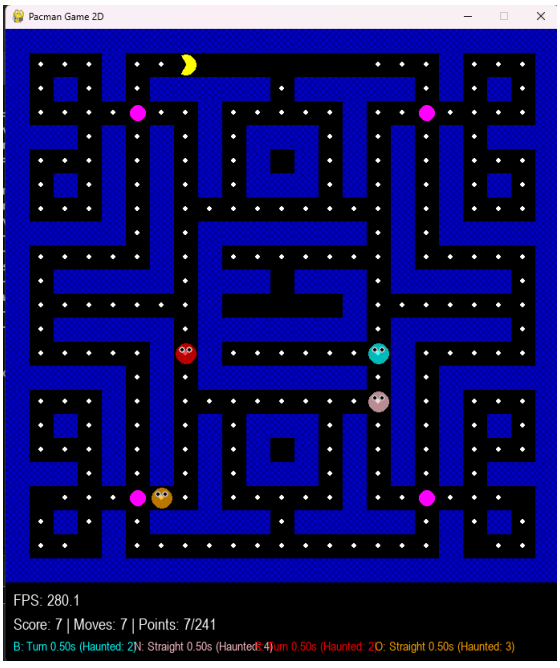
2.2.Các yếu tố trong game



Player: Nhân vật do người chơi điều khiển
B(nlue) Ghost: Con ma đuổi theo Player bằng thuật toán BFS
P(pink Ghost: Con ma đuổi theo Player bằng thuật toán A*
R(red) Ghost: Con ma đuổi theo Player bằng thuật toán UCS
O(orange) Ghost: Con ma đuổi theo Player bằng thuật toán DFS
Haunted Point: Ô gia tốc cho con ma
Wall: Vật cản không thể đi xuyên qua

2.3. Giao diện người chơi





3. HƯỚNG DẪN CÀI ĐẶT VÀ SỬ DỤNG

3.1. Tổng quan

Phiên bản python: python 3.13.2
Thư viện sử dụng: thư viện chuẩn của python
Thư viện cài đặt thêm:

+ pygame
+ colorama

Tham số dòng lệnh:

- -text: hiển thị game text-base
- -2d: hiển thị game 2d
- -graph: mô tả trực quan cách mà graph được biểu diễn
- -algo: hiển thị đường đi của thuật toán từ cùng một vị trí đến người chơi; mô tả các thông số kỹ thuật: thời gian xử lý, bộ nhớ , số bước đi, tổng trọng số, nước đi kế tiếp vị trí hiện tại, số vị trí duy nhất (không tính các nước đi lặp lại).

3.2. Hướng dẫn cài đặt.

B1: Chuẩn bị sẵn git, python3.13.2
B2: Nhập các lệnh sau trong command prompt để cài đặt nhanh source:

+ git clone https://github.com/dtduy23/Pacman.git

B3: Cài đặt thư viện:

+ pip install pygame
+ pip install colorama

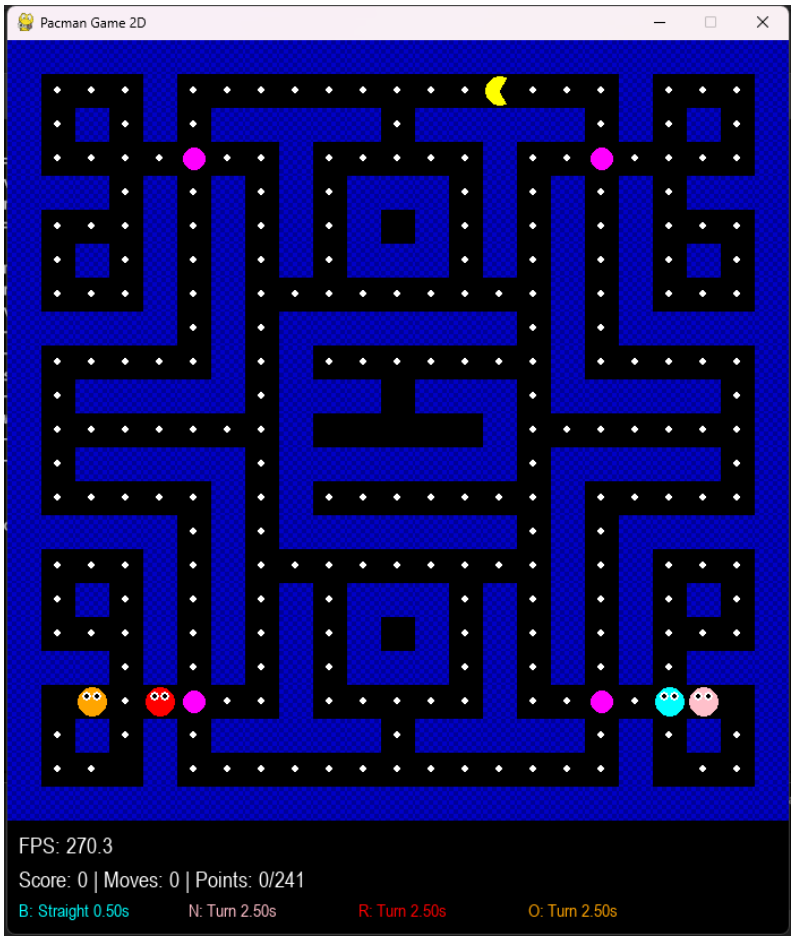
B4 : chạy file main (tùy chọn) để hiển thị những thông tin cần thiết.

+ python -u “path file main”
  ví dụ: python -u "d:\Năm 2\Kỳ 2\csAI\project1\source\main.py"
+ màn hình sẽ hiển thị các tham số cho người dung nhập vào:
  usage: main.py [-h] (-text | -2d | -graph | -algo)
  main.py: error: one of the arguments -text -2d -graph -algo is required

B5 : chạy chương trình

+ python -u “path file main” -text

+ python -u “path file main” -2d

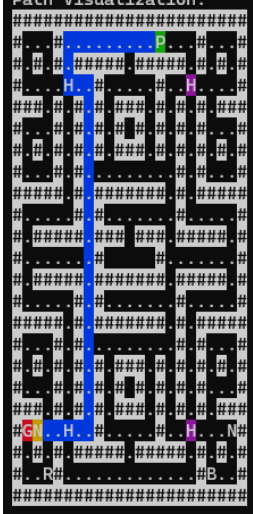


+ python -u “path file main” -graph
hiển thị vị trí mặc định : 1,1.
Vị trí thực nghiệm hiện tại 3,3

```
============================================================
Enter coordinates to examine state transitions (or q to quit):
Position (x,y): 4,4

=== EXAMINING ALL DIRECTIONS AT POSITION (4, 4) ===
This position doesn't exist in the graph!

============================================================
Enter coordinates to examine state transitions (or q to quit):
Position (x,y): 3,3

=== EXAMINING ALL DIRECTIONS AT POSITION (3, 3) ===

Coming from direction: UP

=== TRANSITIONS FROM POSITION (3, 3) COMING FROM UP ===
------------------------------------------------------------
Next Position   Direction  Weight   Turn Type
------------------------------------------------------------
(2, 3)          LEFT       10       TURN (weight 5)
(3, 2)          UP         2        STRAIGHT (weight 2)
(3, 4)          DOWN       20       BACK (weight 10)
(4, 3)          RIGHT      10       TURN (weight 5)

Coming from direction: DOWN

=== TRANSITIONS FROM POSITION (3, 3) COMING FROM DOWN ===
------------------------------------------------------------
Next Position   Direction  Weight   Turn Type
------------------------------------------------------------
(2, 3)          LEFT       10       TURN (weight 5)
(3, 2)          UP         20       BACK (weight 10)
(3, 4)          DOWN       2        STRAIGHT (weight 2)
(4, 3)          RIGHT      10       TURN (weight 5)

Coming from direction: LEFT

=== TRANSITIONS FROM POSITION (3, 3) COMING FROM LEFT ===
------------------------------------------------------------
Next Position   Direction  Weight   Turn Type
------------------------------------------------------------
(2, 3)          LEFT       2        STRAIGHT (weight 2)
(3, 2)          UP         10       TURN (weight 5)
(3, 4)          DOWN       10       TURN (weight 5)
(4, 3)          RIGHT      20       BACK (weight 10)

Coming from direction: RIGHT

=== TRANSITIONS FROM POSITION (3, 3) COMING FROM RIGHT ===
------------------------------------------------------------
Next Position   Direction  Weight   Turn Type
------------------------------------------------------------
(2, 3)          LEFT       20       BACK (weight 10)
(3, 2)          UP         10       TURN (weight 5)
(3, 4)          DOWN       10       TURN (weight 5)
(4, 3)          RIGHT      2        STRAIGHT (weight 2)

============================================================
Enter coordinates to examine state transitions (or q to quit):
Position (x,y):
```

+ python -u “path file main” -algo

```
C:\Users\Admin>python -u "d:\Năm 2\Kỳ 2\csAI\project1\source\main.py" -algo
pygame 2.6.1 (SDL 2.28.4, Python 3.13.2)
Hello from the pygame community. https://www.pygame.org/contribute.html
Map loaded successfully
Ghost: (1, 19)
Player: (14, 1)

------------------------------------------------------------------------------------------------------------------------
Algorithm Steps    Total Cost    Next Position    Unique Pos    Time (s)     Memory (MB)
------------------------------------------------------------------------------------------------------------------------
Testing UCS...
  Result: 36 steps, total cost: 78
  Next position: (2, 19)
  Unique positions visited: 36
  Execution time: 0.007190 seconds
  Memory usage: current=0.0806 MB, peak=0.1589 MB
UCS      36       78          (2, 19)         36          N/A          0.1589

Path Visualization:
```



```
Legend:
G - Ghost
P - Player
N - Next Position
. - Path
. - Haunted Point
# - Wall

Press Enter to continue to the next algorithm...
```

4. THUẬT TOÁN

4.1. Cài đặt graph.

1. Cấu trúc Graph và biểu diễn trạng thái

- **Mô hình hóa đồ thị**
  + **Biểu diễn đồ thị**: Đồ thị có hướng và có trọng số (directed weighted graph)
  + **Định nghĩa trạng thái**: Mỗi trạng thái là một cặp (position, direction)
    - position: Tọa độ (x, y) trên bản đồ
    - direction: Hướng di chuyển hiện tại (UP, DOWN, LEFT, RIGHT)
  + **Cạnh và trọng số**: Mỗi cạnh biểu diễn một bước di chuyển hợp lệ với chi phí tương ứng
- **Cấu trúc dữ liệu:**

```
class MapGraph:
    def __init__(self, game_map):
        self.map = game_map
        self.graph = {}  # Dictionary lưu trữ đồ thị
        self.haunted_points = set(game_map.haunted_points)
        self.moves_since_haunted = 0
        self.build_graph()
```

+ **Cấu trúc của self.graph**: Dictionary với khóa là (position, direction) và giá trị là dictionary của các vị trí kề và chi phí di chuyển
+ **Tách biệt thông tin Haunted points**: Lưu trữ các điểm H riêng biệt khỏi cấu trúc đồ thị chính

2. Xây dựng đồ thị.
   - B1: Duyệt qua toàn bộ bản đồ để xác định các vị trí hợp lệ (không phải tường)
   - B2: Với mỗi vị trí hợp lệ, xem xét tất cả các hướng di chuyển có thể
   - B3: Tính toán chi phí cho mỗi bước di chuyển dựa trên sự thay đổi hướng:
     + Đi thẳng: 1 đơn vị
     + Rẽ 90°: 7 đơn vị
     + Quay đầu 180°: 14 đơn vị
   - B4: Thêm cạnh vào đồ thị nếu vị trí đích là hợp lệ
3. Truy vấn đồ thị và xử lý các trường hợp đặc biệt
   - **Các phương thức truy vấn**
     + **Lấy láng giềng với trọng số**: get_neighbors_with_weights(state)
     + **Kiểm tra tính hợp lệ của một vị trí**: is_valid_position(pos)
     + **Xác định các vị trí kề có thể đi đến**: get_valid_neighbors(pos, direction)
     + **Xử lý các trường hợp đặc biệt**
     + **Chướng ngại vật tạm thời**: Thêm/xóa chướng ngại vật tạm thời để tránh xung đột giữa các ma
4. **Tích hợp điểm Haunted (H) với đồ thị**
   Thiết kế tách biệt
   - **Điểm H không được tích hợp trực tiếp vào cấu trúc đồ thị** mà được lưu trữ riêng biệt
   - **Thuật toán tìm đường sẽ xử lý ảnh hưởng của điểm H on-the-fly.**
     Lý do thiết kế này:
     + Tách biệt dữ liệu tĩnh (cấu trúc đồ thị) và dữ liệu động (trạng thái haunted)
     + Cho phép nhiều thuật toán sử dụng cùng một đồ thị với các cách xử lý H khác nhau
     + Dễ dàng mở rộng và sửa đổi ảnh hưởng của H mà không cần xây dựng lại đồ thị

4.2. Breadth-first search (BFS)

1. Giới thiệu

- **Nguyên lý cơ bản**
  + Breadth-First Search (BFS) là thuật toán tìm kiếm theo chiều rộng, khám phá tất cả các nút ở cùng độ sâu trước khi đi sâu hơn
  + Sử dụng cấu trúc dữ liệu hàng đợi (queue) với cơ chế FIFO (First In, First Out)
  + Đảm bảo tìm được đường đi ngắn nhất về số bước (không phải chi phí) từ điểm xuất phát đến đích
- **Ý tưởng chính**

  Khởi tạo hàng đợi với trạng thái ban đầu
  + Lặp cho đến khi hàng đợi rỗng:
    - Lấy trạng thái đầu tiên từ hàng đợi
    - Kiểm tra đích, nếu đạt đích, trả về kết quả
    - Nếu chưa thăm, đánh dấu đã thăm
    - Thêm tất cả các trạng thái kề chưa thăm vào cuối hàng đợi

2. Cài đặt

- **Biểu diễn trạng thái**
  + Trạng thái trong BFS là một tuple (position, direction), bao gồm:
    - position: Tọa độ (x, y) trên bản đồ
    - direction: Hướng di chuyển hiện tại của ma
- **Cấu trúc dữ liệu sử dụng**
  + Hàng đợi (queue): Danh sách các trạng thái cần khám phá
  + Tập các trạng thái đã thăm (visited set): Tránh thăm lại các trạng thái
  + Đường đi (path): Danh sách các vị trí từ điểm xuất phát đến vị trí hiện tại
- **Quá trình khám phá**
  + BFS bắt đầu từ vị trí hiện tại của ma (ghost)
  + Khám phá các ô kề cạnh theo tất cả các hướng có thể
  + Đánh dấu các trạng thái đã thăm để tránh thăm lại
  + Tiếp tục mở rộng cho đến khi tìm thấy Pacman
- **Xử lý chi phí và ảnh hưởng của điểm Haunted**
  + BFS tìm đường ngắn nhất về số bước di chuyển, không quan tâm đến chi phí trong quá trình tìm kiếm
  + Sau khi tìm được đường đi, BFS mới tính toán chi phí thực tế
- **Xử lý điểm Haunted (H)**
  + BFS không xét đến điểm H trong quá trình tìm đường
  + Chỉ khi đã tìm thấy đường đi đến Pacman, BFS mới tính toán chi phí cuối cùng bằng hàm calculate_path_cost()
  + Trong tính toán chi phí cuối cùng, xét đến ảnh hưởng của các điểm H trên đường đi:
    - Sau khi đi qua điểm H, trong 10 bước tiếp theo tất cả chi phí di chuyển đều là 1 đơn vị
    - Chi phí thông thường: đi thẳng = 1, rẽ 90° = 7, quay đầu 180° = 14

3. Phân tích độ phức tạp

- **Độ phức tạp thời gian**
  + Trường hợp tốt nhất: O(1) - khi Pacman ở kề với vị trí của ma
  + Trường hợp xấu nhất: O(|V| + |E|) - khi phải duyệt toàn bộ đồ thị

    - |V|: số đỉnh (các vị trí trên bản đồ)
    - |E|: số cạnh (các đường đi có thể)
- **Độ phức tạp không gian**
    + O(|V|) để lưu trữ hàng đợi và tập các trạng thái đã thăm

4.3. Uniform-cost search (UCS)

1. Giới thiệu thuật toán UCS

- **Nguyên lý cơ bản**
    + Uniform Cost Search (UCS) là thuật toán tìm kiếm đồ thị có trọng số, luôn mở rộng nút có chi phí tích lũy thấp nhất từ điểm xuất phát.
    + Sử dụng hàng đợi ưu tiên (priority queue) để quản lý các nút theo thứ tự tăng dần của chi phí.
    + Đảm bảo tìm được đường đi có tổng chi phí thấp nhất từ điểm xuất phát đến đích.
- **Ý tưởng chính**
    + Khởi tạo hàng đợi ưu tiên với trạng thái ban đầu (chi phí 0)
    + Lặp cho đến khi hàng đợi ưu tiên rỗng:
        - Lấy trạng thái có chi phí thấp nhất từ hàng đợi ưu tiên
        - Kiểm tra đích, nếu đạt đích, trả về kết quả
        - Nếu chưa thăm, đánh dấu đã thăm
        - Tính chi phí mới cho tất cả các trạng thái kề
        - Thêm các trạng thái kề với chi phí được cập nhật vào hàng đợi ưu tiên

2. Cài đặt

- **Biểu diễn trạng thái**
    + Trạng thái trong UCS là một tuple (position, direction), bao gồm:
        - position: Tọa độ (x, y) trên bản đồ
        - direction: Hướng di chuyển hiện tại của ma
    + Mỗi trạng thái lưu trữ thêm chi phí tích lũy từ điểm xuất phát
- **Cấu trúc dữ liệu sử dụng**
    + Hàng đợi ưu tiên (priority queue): Lưu trữ và sắp xếp các trạng thái theo chi phí tăng dần
    + Từ điển chi phí (cost_so_far): Lưu trữ chi phí tốt nhất để đến mỗi trạng thái
    + Tập các trạng thái đã thăm (visited set): Tránh thăm lại các trạng thái
    + Từ điển bước haunted (haunted_steps): Theo dõi số bước sau khi đi qua điểm H
- **Quá trình tìm đường**
    + UCS bắt đầu từ vị trí hiện tại của ma đỏ (red ghost)
    + Luôn mở rộng nút có chi phí tích lũy thấp nhất từ điểm xuất phát
    + Đánh dấu các trạng thái đã thăm để tránh lặp vô hạn
    + Cập nhật chi phí khi tìm thấy đường đi tốt hơn đến một trạng thái
    + Tiếp tục mở rộng cho đến khi tìm thấy Pacman

3. Xử lý chi phí di chuyển và điểm Haunted

- **Các loại chi phí di chuyển**
    + Đi thẳng (STRAIGHT): 1 đơn vị
    + Rẽ 90° (TURN): 7 đơn vị
    + Quay đầu 180° (BACK): 14 đơn vị
- **Xử lý điểm Haunted (H)**

+ Khi ma đi qua điểm H, trong 10 bước tiếp theo, tất cả chi phí di chuyển đều bằng 1 đơn vị
+ UCS theo dõi trạng thái "bị ma ám" cho mỗi đường đi bằng từ điển haunted_steps
+ Khi phát hiện một trạng thái ở điểm H, đặt lại bộ đếm haunted về 0
+ Điều chỉnh trọng số của các cạnh dựa trên trạng thái haunted:

4. Cách UCS ưu tiên điểm H trong tìm đường

- **Cơ chế ưu tiên tự nhiên**
  + UCS lựa chọn nút có chi phí tích lũy thấp nhất để mở rộng
  + Khi một đường đi đi qua điểm H, tất cả chi phí di chuyển trong 10 bước tiếp theo đều bằng 1
  + Điều này làm cho các đường đi qua H có chi phí tổng thấp hơn đáng kể
  + UCS tự nhiên ưu tiên các đường đi này do cơ chế lựa chọn nút dựa trên chi phí
- **Ví dụ minh họa**

  Xét hai đường đi từ ma đến Pacman:
  + Đường không qua H: 2 bước thẳng, 3 lần rẽ = 2*1* + *3*7 = 23 đơn vị
  + Đường qua H: 5 bước thẳng, 2 lần rẽ, nhưng đi qua H ở bước thứ 2
    - Chi phí đến H: 1*1* + *1*7 = 8 đơn vị
    - Chi phí sau H: 4*1* + *1*1 = 5 đơn vị (tất cả đều bằng 1 nhờ hiệu ứng H)
    - Tổng chi phí: 8 + 5 = 13 đơn vị

  Đường thứ hai có chi phí thấp hơn mặc dù dài hơn về số bước, nên UCS sẽ ưu tiên đường này.

5. Phân tích độ phức tạp

- **Độ phức tạp thời gian**
  + Trường hợp tốt nhất: O(|E|) - khi đích nằm gần với vị trí xuất phát
  + Trường hợp xấu nhất: O(|E| + |V| log |V|)
    - |V|: số đỉnh (các vị trí trên bản đồ)
    - |E|: số cạnh (các đường đi có thể)
    - log |V|: chi phí cho thao tác trên hàng đợi ưu tiên
- Phân tích: Mỗi đỉnh được thêm/lấy khỏi hàng đợi ưu tiên tối đa một lần (O(|V| log |V|)) và mỗi cạnh được xem xét tối đa một lần (O(|E|))
- **Độ phức tạp không gian**
  + O(|V|) - để lưu trữ:
    - Hàng đợi ưu tiên
    - Tập các trạng thái đã thăm
    - Từ điển chi phí
    - Từ điển bước haunted

4.4. Depth-first search (DFS)

1. Giới thiệu thuật toán DFS

- **Nguyên lý cơ bản**
  + Depth-First Search (DFS) là thuật toán tìm kiếm đồ thị theo chiều sâu, khám phá càng xa càng tốt theo mỗi nhánh trước khi quay lui.
  + Sử dụng cấu trúc dữ liệu ngăn xếp (stack) với cơ chế LIFO (Last In, First Out).

+ Tạo ra các đường đi không nhất thiết tối ưu nhưng thường có tính không dự đoán cao.

- **Ý tưởng chính**
  + Khởi tạo ngăn xếp với trạng thái ban đầu
  + Lặp cho đến khi ngăn xếp rỗng:
    - Lấy trạng thái trên cùng của ngăn xếp
    - Kiểm tra đích, nếu đạt đích, trả về kết quả
    - Nếu chưa thăm, đánh dấu đã thăm
    - Thêm tất cả các trạng thái kề chưa thăm vào đỉnh ngăn xếp

2. Cài đặt

- **Biểu diễn trạng thái**
  + Trạng thái trong DFS là một tuple (position, direction), bao gồm:
    - position: Tọa độ (x, y) trên bản đồ
    - direction: Hướng di chuyển hiện tại của ma
  + Mỗi trạng thái cũng lưu trữ đường đi từ điểm xuất phát đến vị trí hiện tại
- **Cấu trúc dữ liệu sử dụng**
  + Ngăn xếp (stack): Lưu trữ các trạng thái cần khám phá
  + Tập các trạng thái đã thăm (visited set): Tránh thăm lại các trạng thái
  + Đường đi (path): Danh sách các vị trí từ điểm xuất phát đến vị trí hiện tại
- **Quá trình tìm đường**
  + DFS bắt đầu từ vị trí hiện tại của ma cam (orange ghost)
  + Khám phá theo chiều sâu, đi sâu nhất có thể theo một hướng trước khi quay lui
  + Đánh dấu các trạng thái đã thăm để tránh lặp vô hạn
  + Tiếp tục khám phá cho đến khi tìm thấy Pacman hoặc đã thăm hết các nút có thể đến được

3. Xử lý chi phí di chuyển và điểm Haunted: không có
4. Phân tích độ phức tạp

- **Độ phức tạp thời gian**
  + Trường hợp tốt nhất: O(1) - khi Pacman ở gần vị trí ban đầu và được khám phá sớm
  + Trường hợp xấu nhất: O(|V| + |E|) - khi phải duyệt toàn bộ đồ thị
    - |V|: số đỉnh (các vị trí trên bản đồ)
    - |E|: số cạnh (các đường đi có thể)
- Độ phức tạp không gian
  + Trường hợp tốt nhất: O(d) - trong đồ thị không chu trình, với d là độ sâu của cây tìm kiếm
  + Trường hợp xấu nhất: O(|V|) - để lưu trữ ngăn xếp và tập các trạng thái đã thăm

4.5. A* search

1. Giới thiệu thuật toán A*

- **Nguyên lý cơ bản**
  + A* là thuật toán tìm kiếm đồ thị có trọng số, kết hợp chi phí thực tế từ điểm xuất phát (g) và ước lượng chi phí đến đích (h).
  + Dựa trên UCS nhưng sử dụng thêm hàm heuristic để ưu tiên các nút hướng về phía đích.
  + Sử dụng hàm ước lượng f(n) = g(n) + h(n) để đánh giá nút, trong đó:

- g(n): Chi phí thực tế từ điểm xuất phát đến nút n
- h(n): Ước lượng chi phí từ nút n đến đích
- f(n): Tổng chi phí ước tính của đường đi qua nút n

- **Ý tưởng chính**
  + Khởi tạo open_set với trạng thái ban đầu, closed_set rỗng
  + Lặp cho đến khi open_set rỗng:
    - Lấy trạng thái có f(n) thấp nhất từ open_set
    - Nếu đạt đích, trả về đường đi
    - Thêm trạng thái vào closed_set
    - Với mỗi nút kề:
  - Tính g(n) mới
  - Nếu nút đã có trong closed_set và g(n) mới không tốt hơn, bỏ qua
  - Nếu nút chưa có trong open_set hoặc g(n) mới tốt hơn:
    Cập nhật g(n), h(n), f(n)
    Thêm vào open_set

2. Cài đặt

- **Biểu diễn trạng thái**
  + Trạng thái trong A* là một tuple (position, direction), bao gồm:
    - position: Tọa độ (x, y) trên bản đồ
    - direction: Hướng di chuyển hiện tại của ma

+ Mỗi trạng thái lưu trữ cả g(n), h(n) và f(n)

- **Cấu trúc dữ liệu sử dụng**
  + Hàng đợi ưu tiên (open_set): Lưu trữ và sắp xếp các trạng thái theo f(n) tăng dần
  + Tập các trạng thái đã đánh giá (closed_set): Đánh dấu các trạng thái đã xử lý
  + Từ điển điểm số g (g_scores): Lưu trữ chi phí thực tế đến mỗi trạng thái
  + Từ điển điểm số f (f_scores): Lưu trữ chi phí ước tính tổng cộng của mỗi trạng thái
  + Từ điển bước haunted (haunted_steps): Theo dõi số bước sau khi đi qua điểm H
  + Bộ nhớ đệm heuristic (cache): Tăng tốc tính toán heuristic
- **Heuristic sử dụng**
  + Manhattan distance (khoảng cách Manhattan): Tổng khoảng cách theo trục x và y
  + Được nhân với chi phí di chuyển thẳng (STRAIGHT) để đảm bảo tính admissible
  + Tính một lần và lưu cache để tối ưu hiệu suất

3. Xử lý chi phí di chuyển và điểm Haunted

- **Các loại chi phí di chuyển**
  + Đi thẳng (STRAIGHT): 1 đơn vị
  + Rẽ 90° (TURN): 7 đơn vị
  + Quay đầu 180° (BACK): 14 đơn vị
- **Xử lý điểm Haunted (H)**
  + A* theo dõi trạng thái "bị ma ám" cho mỗi đường đi một cách độc lập
  + Khi ma đi qua điểm H, đặt lại bộ đếm haunted_steps về 0
  + Trong 10 bước tiếp theo, tất cả chi phí di chuyển đều bằng 1 đơn vị

+ Điều chỉnh trọng số của các cạnh dựa trên trạng thái haunted:

4. **Cơ chế xử lý tie-breaking:** A* đặc biệt sử dụng cơ chế tie-breaking để xử lý trường hợp nhiều nút có cùng giá trị f(n):
   + Thêm bộ đếm để tạo ra một ID duy nhất cho mỗi nút
   + Khi hai nút có cùng f_score, ưu tiên nút có g_score thấp hơn (gần điểm xuất phát hơn)
   + Nếu cả f_score và g_score đều bằng nhau, sử dụng counter để đảm bảo tính ổn định
5. Phân tích độ phức tạp

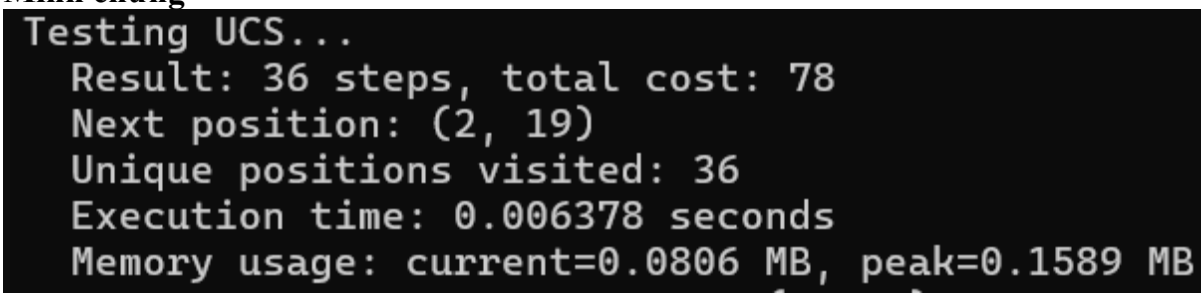
- **Độ phức tạp thời gian**
  + Trường hợp tốt nhất: O(1) - khi đích nằm gần và heuristic hiệu quả
  + Trường hợp xấu nhất: O(|E| + |V| log |V|)
    - |V|: số đỉnh (các vị trí trên bản đồ)
    - |E|: số cạnh (các đường đi có thể)
    - log |V|: chi phí cho thao tác trên hàng đợi ưu tiên
- Phân tích:
    - A* có thể khám phá ít nút hơn UCS nhờ heuristic
    - Nếu heuristic không hiệu quả, A* có thể trở nên tương tự như UCS
    - Hiệu suất của A* phụ thuộc nhiều vào chất lượng của heuristic
- **Độ phức tạp không gian**
  + O(|V|) - để lưu trữ open_set, closed_set, g_scores, f_scores và cache

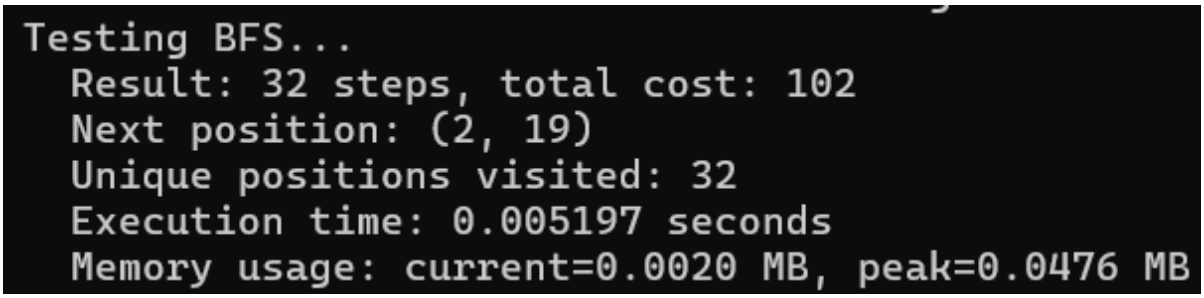
5. Nhận Xét chung

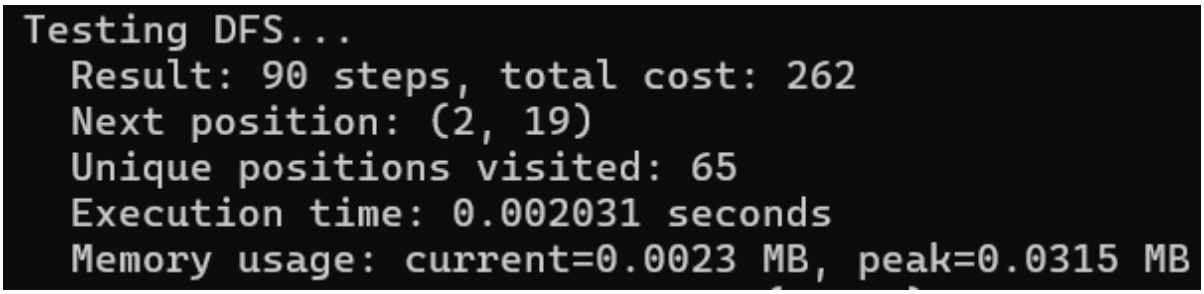
1. Đặc điểm và hiệu quả

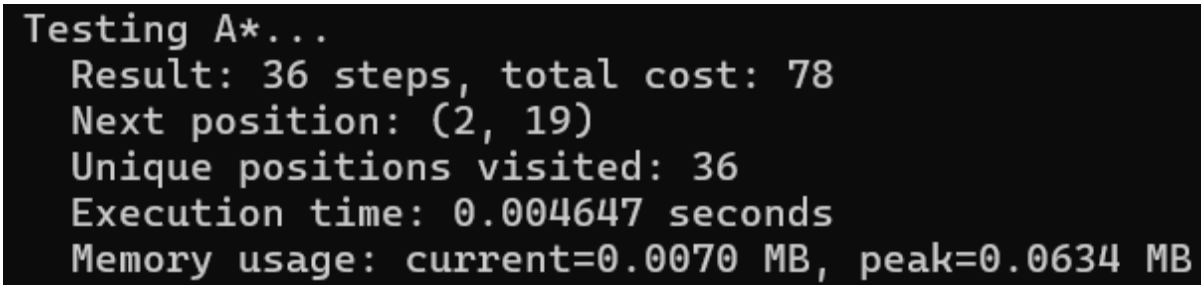
- **Hiệu quả về tìm đường**
  - **A*** thể hiện hiệu quả cao nhất, kết hợp ưu điểm của UCS (tối ưu chi phí) với heuristic (giảm không gian tìm kiếm).
  - **UCS** đảm bảo tìm ra đường đi tối ưu về chi phí nhưng có thể khám phá nhiều nút không cần thiết.
  - **BFS** tìm ra đường ngắn nhất về số bước nhưng không tối ưu về chi phí di chuyển.
  - **DFS** không đảm bảo tìm ra đường đi tối ưu, nhưng tạo ra hành vi khó đoán.
- **Khả năng tận dụng điểm Haunted (H)**
  - **UCS** và **A*** tận dụng hiệu quả nhất các điểm H trong quá trình tìm đường, điều chỉnh trọng số cạnh "on-the-fly".
  - **BFS** và **DFS** không xét đến điểm H trong quá trình tìm kiếm, chỉ tính toán ảnh hưởng khi đã tìm thấy đường đi.
- **Độ phức tạp thời gian**
  - **DFS**: O(|V| + |E|), nhưng có thể tìm thấy đường đi (không tối ưu) nhanh hơn trong một số trường hợp.
  - **BFS**: O(|V| + |E|), tối ưu khi tìm đường ngắn nhất về số bước.
  - **UCS**: O(|E| + |V| log |V|), tốn thời gian hơn do phải sắp xếp các nút theo chi phí.
  - **A***: O(|E| + |V| log |V|) trong trường hợp xấu nhất, nhưng thường hiệu quả hơn UCS nhờ heuristic.

2. Minh chứng






Testing DFS...
Result: 90 steps, total cost: 262
Next position: (2, 19)
Unique positions visited: 65
Execution time: 0.002031 seconds
Memory usage: current=0.0023 MB, peak=0.0315 MB


Testing A*...
Result: 36 steps, total cost: 78
Next position: (2, 19)
Unique positions visited: 36
Execution time: 0.004647 seconds
Memory usage: current=0.0070 MB, peak=0.0634 MB

BẢNG PHÂN CÔNG CÔNG VIỆC

STT	THÀNH VIÊN	MSSV	NHIỆM VỤ	MỨC HOÀN THÀNH
1	ĐINH TUẤN DUY	23127356	Cài đặt đồ thị. Cài đặt thuật toán UCS. Cải tiến BFS, A*, DFS. Chỉnh sửa level5, level 6. Viết báo cáo	100%
2	LỮ BẢO ĐẠT	23127338	Cài đăt thuật toán BFS, fix bug. Cài đặt Level 5	100%
3	THIỀU QUANG VINH	23127143	Cài đăt thuật toán A* Viết báo cáo, quay dựng video, kiểm tra lỗi, fix bug.	100%
4	BÙI ĐỨC ĐẠT	23127337	Cài đăt thuật toán DFS. Kiểm tra Search performance.	100%

TÀI LIỆU THAM KHẢO

[1] Stuart Russell and Peter Norvig. April 2020. *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

Link demo video: https://www.youtube.com/watch?v=0lzkR2CX7Bg