

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ
MINH TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ
NHIÊN KHOA CÔNG NGHỆ THÔNG TIN**



**GRAPH SEARCH
ROBOR TÌM ĐƯỜNG**

Môn học: Cơ Sở Trí Tuệ Nhân Tạo
Lớp: 21_1

Giảng viên hướng dẫn: ThS Bùi Duy Đăng

MỤC LỤC

Thông tin nhóm.....	3
Mức độ hoàn thành.....	3
I. Mức 1: Thuật toán tìm đường từ S đến G Thuật toán BFS (Breadth-First Search) 4	
II. Mức 2: Chi tiết các thuật toán tìm đường.....	8
1. Thuật toán DFS (Depth-First Search).....	8
2. Thuật toán UCS (Uniform Cost Search).....	12
3. Thuật toán A*	16
4. Sự khác nhau giữa 3 thuật toán	20
III. Mức 3: Đón tất cả các điểm rồi đến trạng thái G	20
1. Hàm tìm đường đi ngắn nhất từ S đến G và đi qua tất cả các điểm đón (các điểm đón không phân biệt thứ tự)	20
2. Hàm liệt kê tất cả hoán vị của các điểm đón (hàm phụ).....	22
IV. Mức 4: Di động hình đa giác	23
V. Mức 5: Mô hình trên không gian 3D	26
VI. Hướng dẫn sử dụng chương trình	30
VII. Tham khảo	32

Thông tin nhóm

STT	MSSV	Tên	Phân công	
1	21120526	Lê Văn Đan Phong	Thuật toán BFS	Tìm tổng đường đi nhỏ nhất
2	21120268	Nguyễn Việt Khanh	Thuật toán UCS	Thể hiện mô hình trên không gian 3 chiều
3	21120410	Nguyễn Tuấn Anh	Thuật toán A*	Thuật toán di động hình đa giác
4	21120608	Nguyễn Thiện Khánh Đoan	Thuật toán DFS	Viết báo cáo

Mức độ hoàn thành

STT	Hoàn thành	Chưa hoàn thành	Phần trăm
Mức 1	Cài đặt thành công 1 thuật toán để tìm đường đi từ S tới G	-	100%
Mức 2	Cài đặt thành công 4 thuật toán tìm kiếm	-	100%
Mức 3	Đón các điểm đến trạng thái G với tổng đường đi nhỏ nhất	-	100%
Mức 4	Di động hình đa giác	-	100%
Mức 5	Thể hiện trên mô hình không gian 3 chiều	-	100%

I. Mức 1: Thuật toán tìm đường từ S đến G Thuật toán BFS (Breadth-First Search)

Ý tưởng:

Thuật toán tìm kiếm theo chiều rộng (BFS) là một thuật toán duyệt đồ thị theo cách khám phá tất cả các đỉnh liền kề với một đỉnh xuất phát trước khi chuyển sang các đỉnh ở mức xa hơn. Nó sử dụng cấu trúc dữ liệu hàng đợi để lưu trữ các đỉnh cần được khám phá.

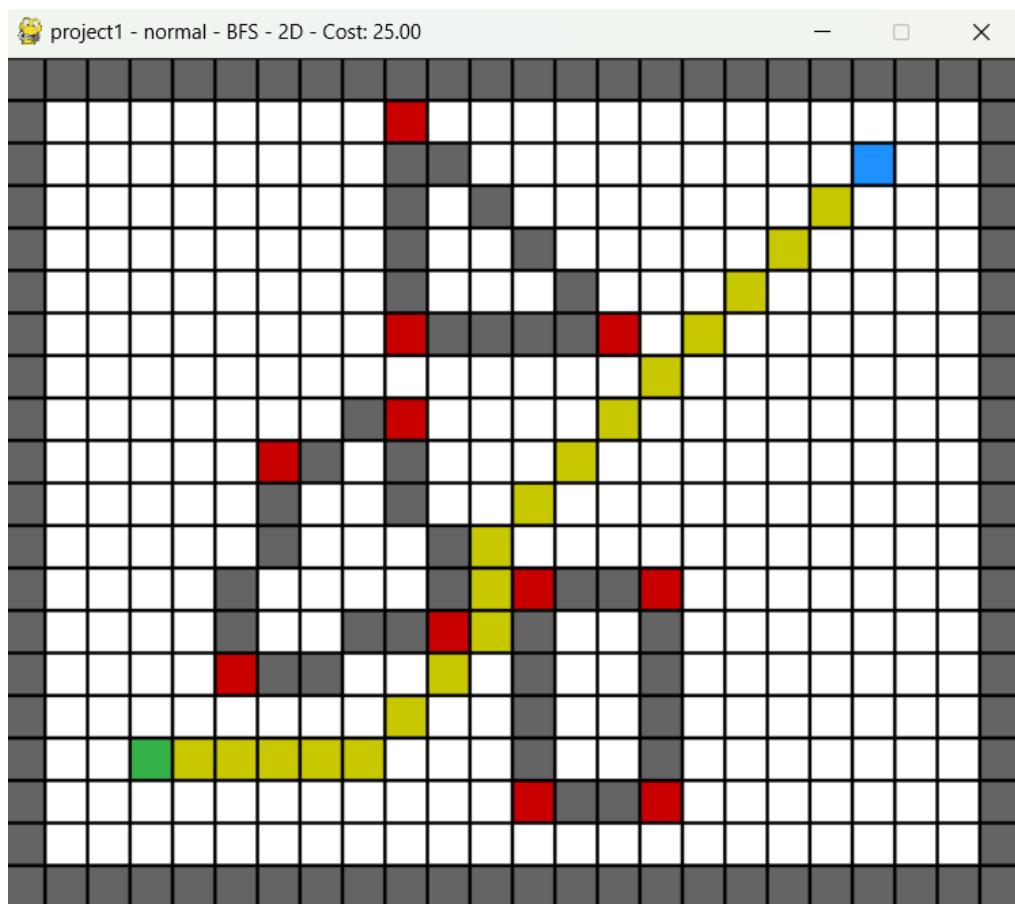
Chi tiết thuật toán:

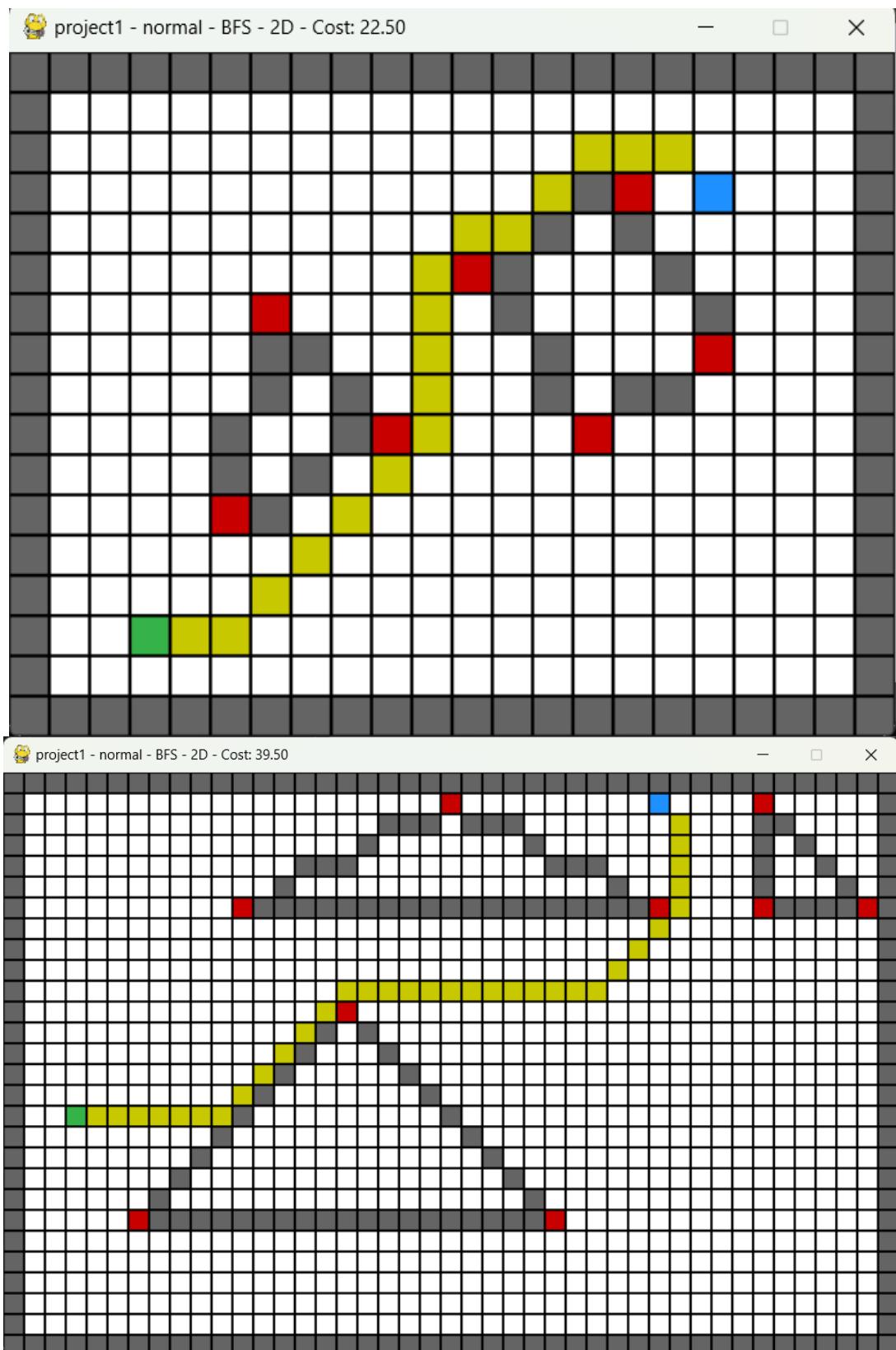
Tên hàm <i>BFS(g: Grid, sc:pygame.Surface)</i>		
Tham số	Chức năng	
	<i>g</i> : Đôi tượng Grid biểu diễn lưới ô vuông <i>sc</i> : Màn hình pygame để vẽ	
Tên biến	Chức năng	
	<i>open_set</i>	Lưu trữ các ô cần thăm dò, hoạt động như một hàng đợi.
	<i>closed_set</i>	Lưu trữ các ô đã thăm dò, giúp tránh lặp lại việc kiểm tra các ô đã được thăm dò.
	<i>father</i>	Lưu vết đường đi từ ô bắt đầu đến từng ô trong lưới, giúp truy ngược lại đường đi khi tìm được ô kết thúc.
	<i>path</i>	Danh sách các ID các ô trên đường đi từ ô kết thúc đến ô bắt đầu.
Thực hiện	<i>cost</i>	Chi phí của đường đi.
	1	Khởi tạo: <ul style="list-style-type: none"> <i>open_set</i>: Danh sách các ô cần thăm dò, ban đầu chỉ chứa ô bắt đầu. <i>closed_set</i>: Tập hợp các ô đã thăm dò, sử dụng <i>set</i> để kiểm tra nhanh hơn. <i>father</i>: Mảng lưu vết đường đi, mỗi ô lưu trữ ID của ô cha.
	2	Tạo vòng lặp while, lặp lại cho đến khi <i>open_set</i> rỗng: <ul style="list-style-type: none"> Lấy ô đầu tiên trong <i>open_set</i> ra và đánh dấu nó đã thăm dò (thêm vào <i>closed_set</i>). Kiểm tra xem ô hiện tại có phải là ô kết thúc hay không. Nếu đúng: <ul style="list-style-type: none"> Tìm danh sách các ô trên đường đi từ ô kết thúc đến ô bắt đầu bằng cách sử dụng <i>find_path</i>. Vẽ đường đi bằng <i>draw_path</i>. Tính chi phí bằng <i>calculate_cost</i>. Hiển thị chi phí lên màn hình bằng <i>show_cost</i>. Thoát khỏi vòng lặp. Lấy danh sách các ô lân cận của ô hiện tại. Duyệt qua các ô lân cận:

		<ul style="list-style-type: none"> ○ Nếu ô lân cận đã được thăm dò, bỏ qua nó. ○ Nếu ô lân cận chưa được thăm dò (không có trong <i>open_set</i> và <i>closed_set</i>), thêm nó vào <i>open_set</i>. ○ Lưu trữ ID của ô hiện tại là cha của ô lân cận trong mảng <i>father</i>.
3		<p>Nếu không tìm ra ô kết thúc:</p> <ul style="list-style-type: none"> • In thông báo "Không tìm được đường đi"

Ví dụ:

- Ví dụ minh họa cho 3 bản đồ khác nhau:





- Trường hợp không có đường đi:

```

PS C:\Users\abc\MON HOC\NAM 3\hk2\CSSTNT\DO AN\project1\source> python main.py --level n --algor BFS
pygame 2.0.2 (SDL 2.0.16, Python 3.10.11)
Hello from the pygame community. https://www.pygame.org/contrib
Kích thước ngang: 22
Kích thước dọc: 18
kích thước ngang màn hình: 600
kích thước dọc màn hình: 500
Kích thước cao: 20
Điểm bắt đầu: (2, 2)
Điểm kết thúc: (19, 16)
Các điểm đón: [(3, 12), (17, 8), (15, 14)]
Các đa giác:
Đa giác 1: [(4, 4), (5, 9), (8, 10), (9, 5)]
Đa giác 2: [(8, 12), (8, 17), (13, 12)]
Đa giác 3: [(11, 0), (11, 11), (14, 6), (14, 1)]
Không tìm được đường đi
|
```

Phân tích:

- Tính đầy đủ:** Thuật toán BFS đảm bảo tìm kiếm đầy đủ trong đồ thị không có chu trình (hoặc vô hạn) khi bắt đầu từ một đỉnh khởi đầu. Nếu có một đường đi từ đỉnh khởi đầu đến đỉnh đích, BFS sẽ tìm ra đường đi đó. Tuy nhiên, nếu không có đường đi từ đỉnh khởi đầu đến đỉnh đích, BFS sẽ duyệt qua tất cả các đỉnh có thể đạt được từ đỉnh khởi đầu trước khi dừng lại. Do đó, nếu không có đường đi, thuật toán BFS vẫn đảm bảo tìm kiếm đầy đủ.
- Tính tối ưu:** Thuật toán BFS đảm bảo tìm kiếm đường đi ngắn nhất từ đỉnh khởi đầu đến các đỉnh khác trong đồ thị không có trọng số. Với mỗi đỉnh, BFS duyệt qua các đỉnh cùng mức trước khi di chuyển xuống các đỉnh cấp dưới. Do đó, khi tìm thấy đỉnh đích, đường đi tìm thấy đầu tiên từ đỉnh khởi đầu đến đỉnh đích sẽ là đường đi ngắn nhất.

S : độ sâu của lời giải gần nhất

n^i số lượng node trong cấp thứ i

- Độ phức tạp về thời gian:** Tương đương với số lượng đỉnh được duyệt qua trong quá trình BFS cho đến khi tìm được lời giải gần nhất (tức là độ sâu nhỏ nhất).

$$T(n) = 1 + n^2 + n^3 + \dots + n^s = O(n^s)$$

- Độ phức tạp của không gian:** Tương đương với kích thước tối đa của hàng đợi (fringe).

$$S(n) = O(n^s)$$

Nhận xét:

- Ưu điểm:
 - Dễ dàng để hiểu và triển khai.
 - Tìm ra đường đi ngắn nhất giữa hai đỉnh trong đồ thị không có trọng số.
 - Có thể được sử dụng để kiểm tra tính liên thông của đồ thị.
- Nhược điểm:
 - Có thể tồn nhiều bộ nhớ cho đồ thị lớn.
 - Có thể không hiệu quả cho đồ thị có nhiều chu trình.

II. Mức 2: Chi tiết các thuật toán tìm đường

1. Thuật toán DFS (Depth-First Search)

Ý tưởng: Thuật toán DFS là thuật toán là tìm kiếm đường đi duyệt sâu vào một nhánh đến khi không duyệt sâu được nữa, sau đó quay lui và duyệt các đỉnh lân cận khác. Thuật toán DFS dùng một stack để lưu trữ các đỉnh cần duyệt. Khi duyệt một đỉnh, các đỉnh lân cận của nó được thêm vào stack.

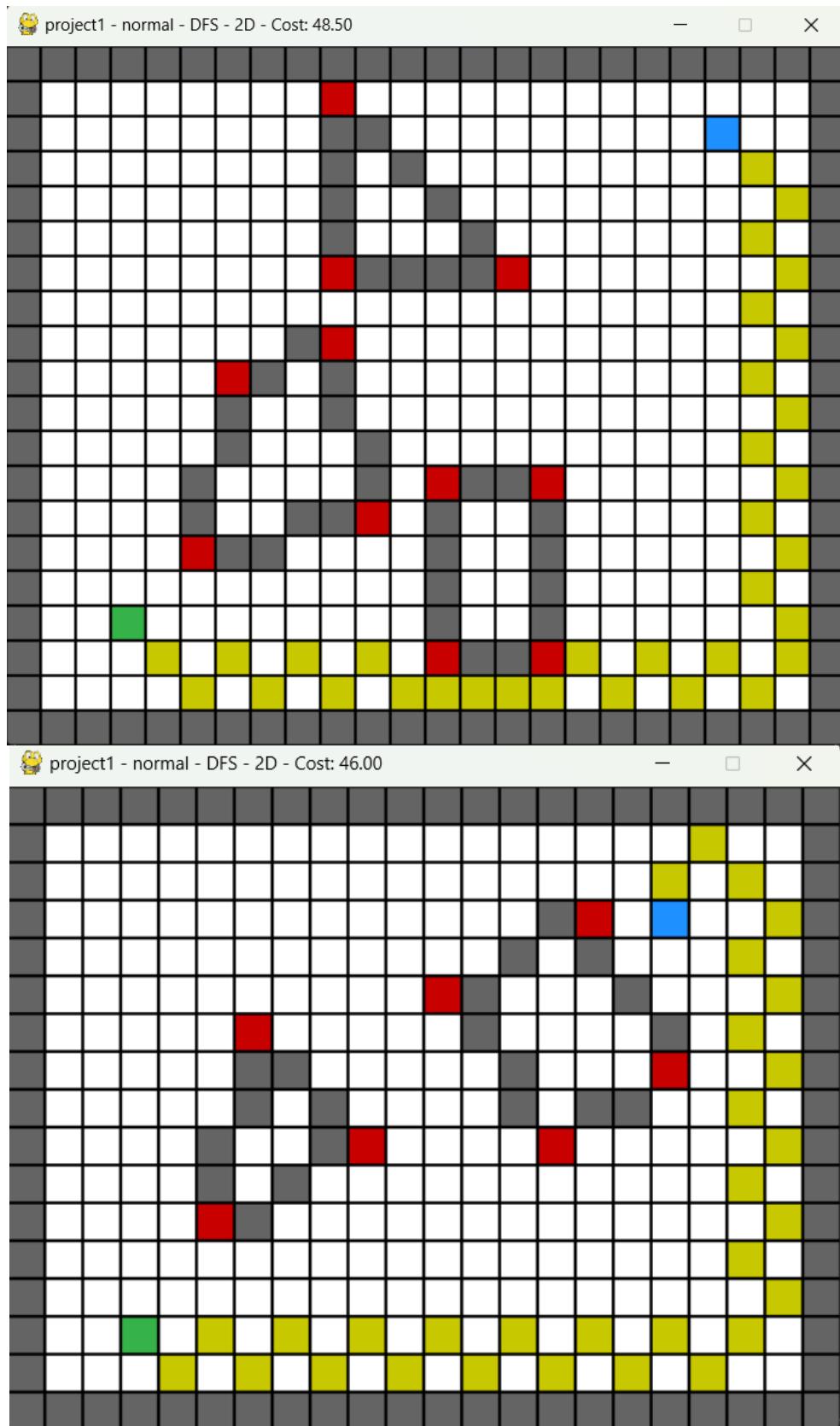
Chi tiết thuật toán:

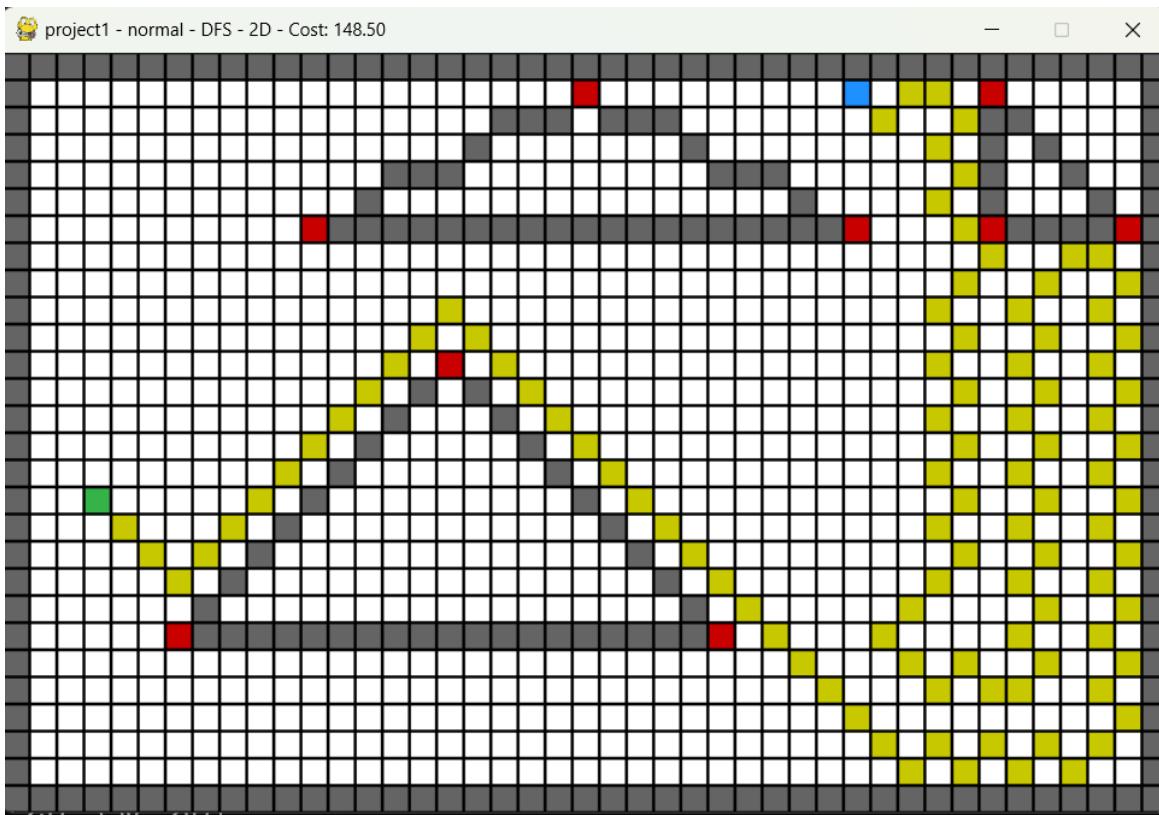
Tên hàm		DFS(<i>g</i> : Grid, <i>sc</i> : pygame.Surface)
Tham số		<i>g</i> : Đối tượng Grid biểu diễn lưới ô vuông <i>sc</i> : Màn hình pygame để vẽ
		Chức năng
Tên biến	<i>open_stack</i>	Lưu trữ các ô cần thăm dò, hoạt động như một ngăn xếp (stack).
	<i>closed_set</i>	Lưu trữ các ô đã thăm dò, giúp tránh lặp lại việc kiểm tra các ô đã được thăm dò.
	<i>father</i>	Lưu vết đường đi từ ô bắt đầu đến từng ô trong lưới, giúp truy ngược lại đường đi khi tìm được ô kết thúc.
	<i>path</i>	Danh sách các ID các ô trên đường đi từ ô kết thúc đến ô bắt đầu.
	<i>cost</i>	Chi phí của đường đi.
Thực hiện	1	Khởi tạo: <ul style="list-style-type: none"> <i>open_stack</i>: Stack chứa các ô cần thăm dò, ban đầu chỉ chứa ô bắt đầu.

		<ul style="list-style-type: none"> • <i>closed_set</i>: Tập hợp các ô đã thăm dò, sử dụng <i>set</i> để kiểm tra nhanh hơn. • <i>father</i>: Mảng lưu vết đường đi, mỗi ô lưu trữ ID của ô cha.
2		<p>Tạo vòng lặp while, lặp lại cho đến khi <i>open_set</i> rỗng:</p> <ul style="list-style-type: none"> • Trong mỗi lần lặp, lấy ô cuối cùng từ <i>open_stack</i> ra và đánh dấu nó là đã thăm dò. • Kiểm tra xem ô hiện tại có phải là ô đích hay không. Nếu đúng: <ul style="list-style-type: none"> ◦ Tìm đường đi từ ô đích về ô bắt đầu bằng cách sử dụng <i>find_path</i>. ◦ Vẽ đường đi bằng <i>draw_path</i>. ◦ Tính chi phí bằng <i>calculate_cost</i>. ◦ Hiển thị chi phí lên màn hình bằng <i>show_cost</i>. ◦ Thoát khỏi vòng lặp. • Lấy danh sách các ô lân cận của ô hiện tại. • Duyệt qua các ô lân cận: <ul style="list-style-type: none"> ◦ Nếu ô lân cận đã được thăm dò, bỏ qua nó ◦ Nếu ô lân cận chưa được thăm dò (không có trong <i>open_stack</i> và <i>closed_set</i>), thêm nó vào <i>open_stack</i>. ◦ Lưu trữ ID của ô hiện tại là cha của ô lân cận trong mảng <i>father</i>.
3		<p>Nếu không tìm ra ô kết thúc:</p> <ul style="list-style-type: none"> • In thông báo "Không tìm được đường đi"

Ví dụ:

- Ví dụ minh họa cho 3 bản đồ khác nhau:





Phân tích:

- **Tính đầy đủ:** Thuật toán DFS không đảm bảo tính đầy đủ, nó có thể không tìm ra được một số nút mục tiêu trong trường hợp đồ thị vô hạn hoặc đồ thị không liên thông. Nếu đồ thị có hạn, DFS có thể tìm ra được nút mục tiêu nếu tồn tại.
- **Tính tối ưu:** DFS không đảm bảo tìm ra lời giải tối ưu nhất. Nó chỉ tìm kiếm theo chiều sâu và dừng lại khi tìm thấy một lời giải. Điều này có nghĩa là nếu có nhiều lời giải, DFS không đảm bảo tìm ra lời giải tối ưu nhất. Để đạt được tính tối ưu, cần sử dụng các thuật toán tìm kiếm khác như thuật toán tìm kiếm theo chiều rộng (BFS) hoặc thuật toán tìm kiếm thông minh như thuật toán A*.
- **Độ phức tạp về thời gian:**
 - Trường hợp tốt nhất:

$$T(n) = O(n)$$

- Trường hợp xấu nhất:

$$T(n) = O(n + E)$$

- **Độ phức tạp về không gian:** phụ thuộc vào số lượng đỉnh trong đồ thị.

$$S(n) = O(n)$$

Nhận xét:

Ưu điểm:

- Dùng để kiểm tra đường đi có tồn tại
- Hiệu quả với cây và đồ thị lớn

Nhược điểm:

- Không đảm bảo tìm ra đường đi ngắn nhất.
- Dễ bị vòng lặp vô hạn.

2. Thuật toán UCS (Uniform Cost Search)

Ý tưởng:

Thuật toán UCS thuật toán là tìm kiếm đường đi từ một đỉnh xuất phát đến một đỉnh đích sao cho tổng trọng số của các cạnh trên đường đi là nhỏ nhất. Điều này được thực hiện bằng cách mở rộng các đỉnh gần nhất trước, rồi xem xét trọng số của các cạnh để quyết định đường đi tiếp theo.

Chi tiết thuật toán:

Tên hàm			<i>UCS(g: Grid, Start: Cell, Goal: Cell)</i>
Tham số			<p><i>g</i>: Đối tượng Grid biểu diễn lưới ô vuông</p> <p><i>Start</i>: ô bắt đầu</p> <p><i>Goal</i>: ô kết thúc</p>
Tên biến			Chức năng
	<i>cell_dict</i>	Lưu trữ các ô theo ID.	
	<i>open_set</i>	Lưu trữ các ô cần thăm dò, sắp xếp theo chi phí.	
	<i>came_from</i>	Lưu vết đường đi từ ô bắt đầu đến từng ô trong lưới	
	<i>g_score</i>	Lưu trữ chi phí từ ô bắt đầu đến mỗi ô.	
	<i>closed_set</i>	Lưu trữ các ô đã thăm dò, giúp tránh lặp lại việc kiểm tra các ô đã được thăm dò.	
Thực hiện	1	Khởi tạo:	<ul style="list-style-type: none"> • <i>cell_dict</i>: Một từ điển để lưu trữ các ô dựa trên ID. • <i>open_set</i>: Hàng đợi ưu tiên khởi tạo với nút bắt đầu, sắp xếp theo chi phí. • <i>came_from</i>: Mảng lưu trữ nút cha cho mỗi ô. • <i>g_score</i>: Tính toán chi phí của mỗi ô từ ô bắt đầu. • <i>closed_set</i>: Tập hợp các ô đã thăm dò.
	2	Start: ô bắt đầu	<ul style="list-style-type: none"> • Trong mỗi lần lặp, lấy ô từ <i>open_set</i> có chi phí thấp nhất ra (bằng <i>heapq.heappop</i>) và đánh dấu nó là đã thăm dò (thêm

		<p>vào <i>closed_set</i>).</p> <ul style="list-style-type: none"> Kiểm tra xem ô hiện tại có phải là ô đích hay không. Nếu đúng: <ul style="list-style-type: none"> Tìm đường đi từ ô đích về ô bắt đầu bằng cách sử dụng <i>find_path</i>. Vẽ đường đi bằng <i>draw_path</i>. Tính chi phí bằng <i>calculate_cost</i>. Hiển thị chi phí lên màn hình bằng <i>show_cost</i>. Thoát khỏi vòng lặp. Lấy danh sách các ô lân cận của ô hiện tại Duyệt qua từng ô lân cận: <ul style="list-style-type: none"> Nếu ô lân cận đã được thăm dò, bỏ qua nó. Nếu ô lân cận chưa được thăm dò: tính toán chi phí đến ô lân cận bằng <i>cost</i> (khoảng cách Euclide từ ô hiện tại đến ô lân cận)
	3	<p>Nếu không tìm ra ô kết thúc:</p> <ul style="list-style-type: none"> In thông báo "Không tìm được đường đi"

Phân tích:

- Tính đầy đủ:** Nếu đồ thị không có chu trình, hoặc trong một đồ thị vô hạn mà không có đỉnh nào liên kết với vô hạn các đỉnh liền kề, đồng thời chi phí của toàn bộ cạnh đều > 0 thì UCS sẽ đảm bảo tìm kiếm đầy đủ.
- Tính tối ưu:** Do UCS luôn mở rộng đỉnh có tổng chi phí nhỏ nhất nên nó luôn tối ưu (vì các đường đi có chi phí nhỏ hơn đã được tìm thấy trước đó).
- Độ phức tạp về thời gian:** UCS mở rộng các đỉnh có chi phí nhỏ hơn lời giải nhỏ nhất. Gọi C^* là chi phí của lời giải và ϵ là các cạnh có chi phí nhỏ nhất, thì độ sâu ước chừng là $C^*/(\epsilon)$. Do đó, độ phức tạp thời gian trong trường hợp xấu nhất của UCS là $O(b^{C^*/\epsilon})$.
- Độ phức tạp của không gian:** Do phải lưu tầng xử lý cuối cùng nên sẽ bằng $O(b^{C^*/\epsilon})$.

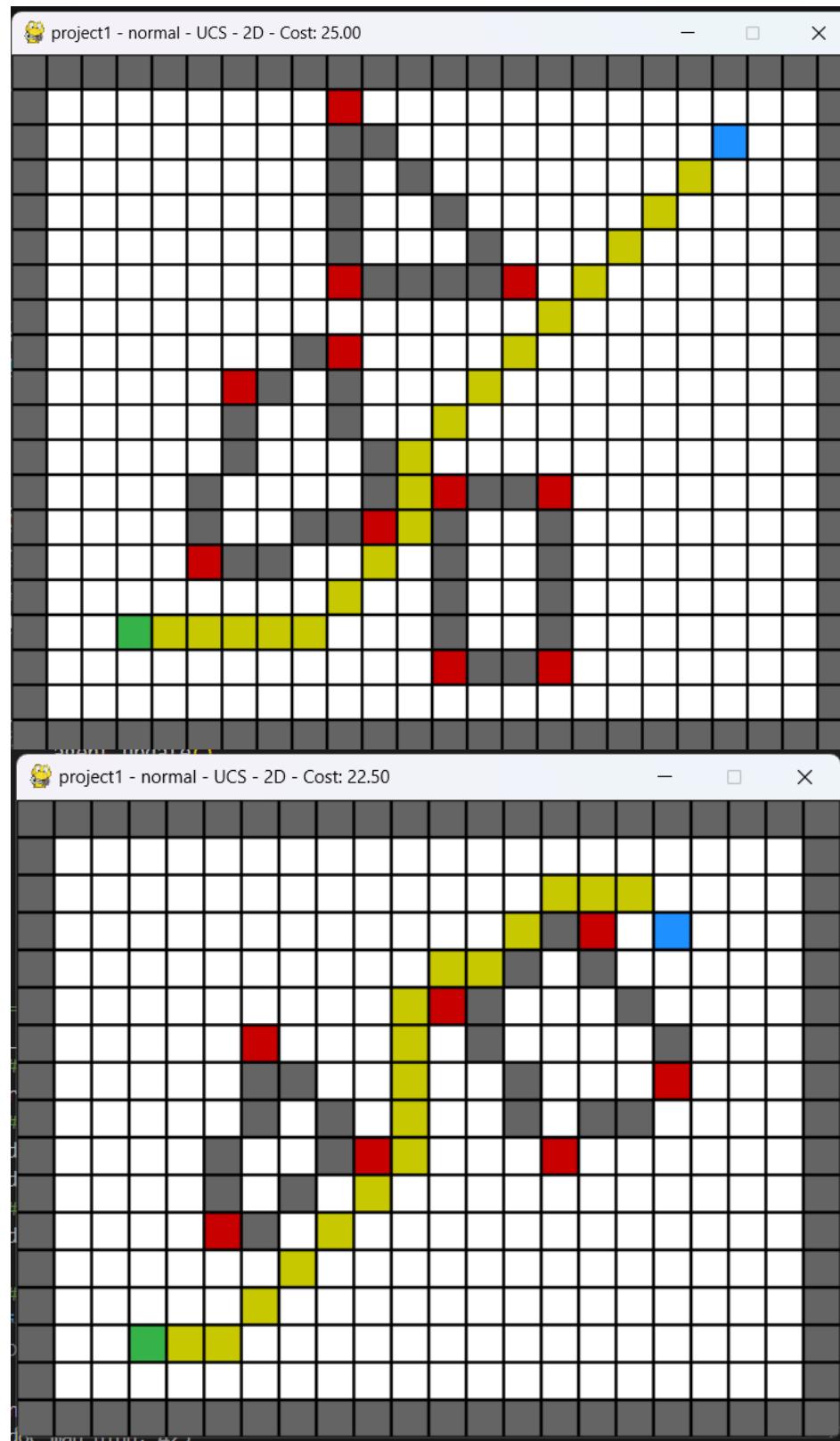
Nhận xét:

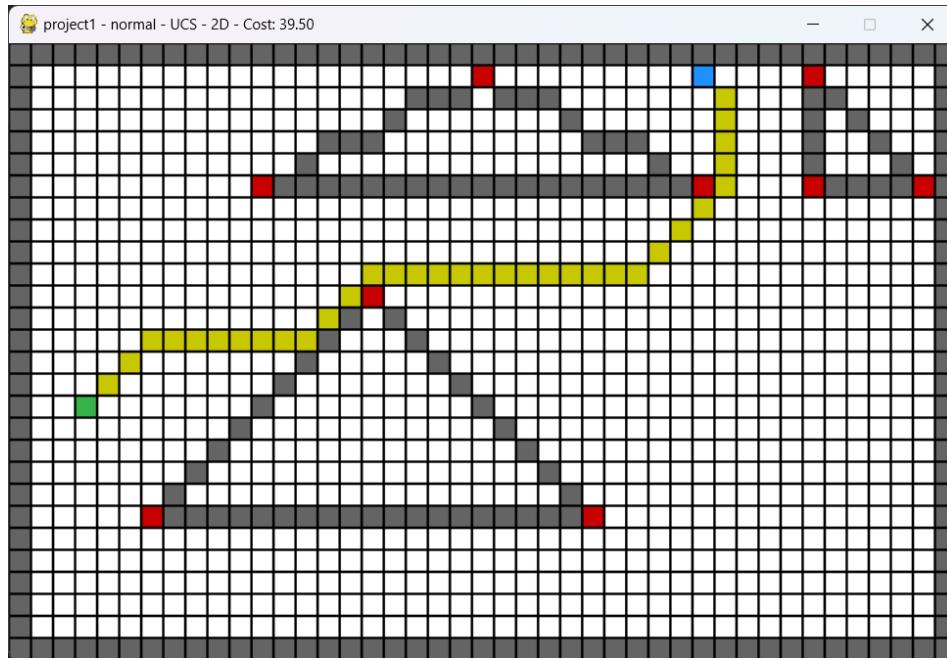
- Ưu điểm:
 - Uniform cost search luôn tối ưu vì ở mọi trạng thái, con đường có chi phí thấp nhất luôn được chọn.

- Nhược điểm:
 - UCS không quan tâm đến số bước liên quan đến việc tìm kiếm và chỉ quan tâm đến chi phí đường dẫn. Do đó thuật toán này có thể bị mắc kẹt trong một vòng lặp vô hạn.

Ví dụ:

Ví dụ minh họa cho 3 bản đồ khác nhau:





3. Thuật toán A*

Ý tưởng:

Thuật toán A* là thuật toán tìm đường đi trong đồ thị, dựa trên thuật toán Dijkstra và Best-First-Search. Ý tưởng chính của A* là sử dụng một hàm đánh giá $f(n)$ để quyết định thứ tự duyệt các nút. Hàm $f(n)$ được tính bằng cách cộng giữa hàm $g(n)$ và $h(n)$, trong đó:

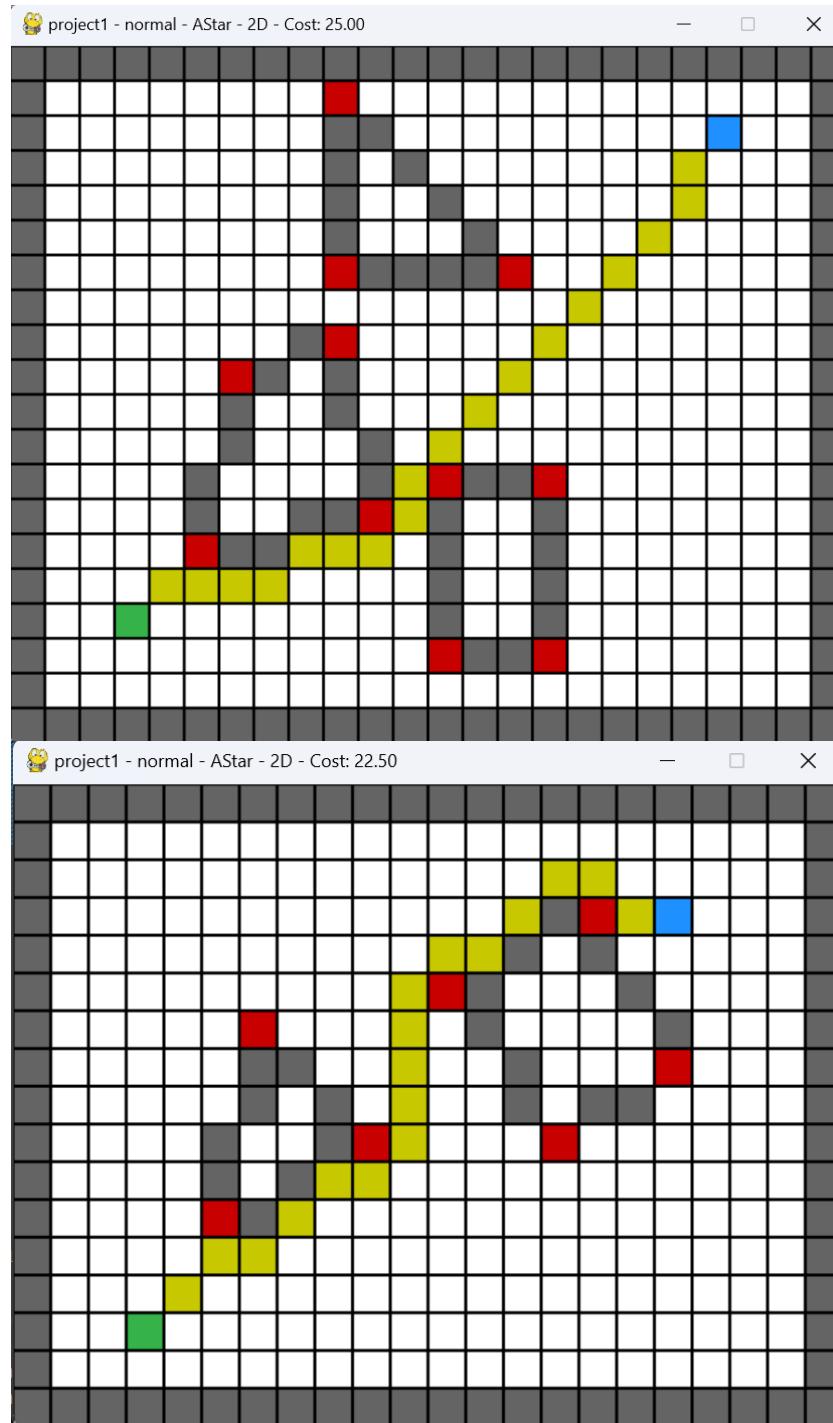
- $g(n)$ là chi phí thực tế từ nút bắt đầu đến nút n.
 - $h(n)$ là chi phí ước lượng từ nút n đến nút đích (thường được gọi là hàm heuristic).
- Thuật toán A* sẽ chọn nút có $f(n)$ nhỏ nhất để duyệt tiếp. Nếu hàm heuristic $h(n)$ không bao giờ đánh giá quá cao chi phí thực tế để đến mục tiêu, A* đảm bảo tìm thấy đường đi ngắn nhất.

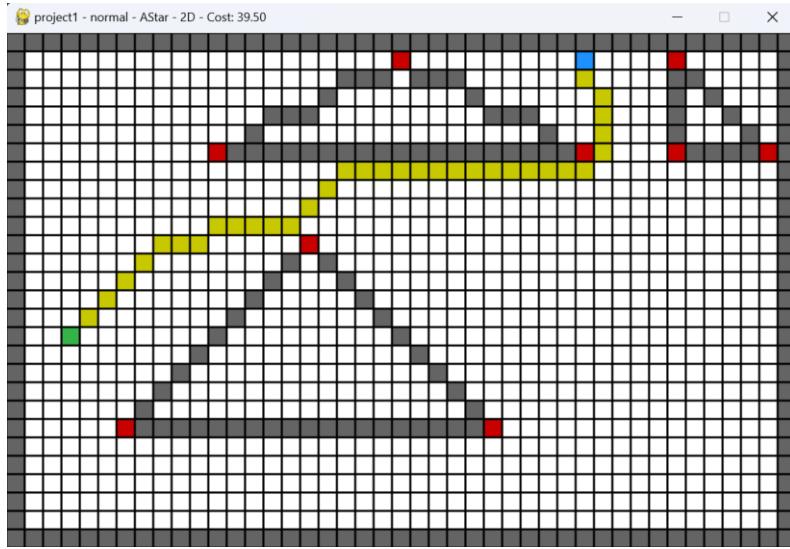
Chi tiết thuật toán:

Tên hàm		<i>AStar(g: Grid, Start: Cell, Goal: Cell)</i>
Tham số	<i>g</i> : Đồ thị Grid biểu diễn lưới ô vuông <i>Start</i> : ô bắt đầu <i>Goal</i> : ô kết thúc	
Tên biến		Chức năng
	<i>cell_dict</i>	Lưu trữ các ô theo ID.
	<i>open_set</i>	Lưu trữ các ô cần thăm dò, sắp xếp theo chi phí.
	<i>came_from</i>	Lưu vết đường đi từ ô bắt đầu đến từng ô trong lưới
	<i>g_score</i>	Lưu trữ chi phí từ ô bắt đầu đến mỗi ô.
	<i>f_score</i>	Lưu trữ chi phí từ ô bắt đầu đến ô đích thông qua ô hiện tại
	<i>h_score</i>	Lưu trữ chi phí ước lượng từ ô cụ thể đến ô đích.

	<i>closed_set</i>	Lưu trữ các ô đã thăm dò, giúp tránh lặp lại việc kiểm tra các ô đã được thăm dò.
Thực hiện	1	<p>Khởi tạo:</p> <ul style="list-style-type: none"> • <i>open_set</i>: Một hàng đợi ưu tiên và thêm ô bắt đầu vào <i>open_set</i> với chi phí là 0. • <i>came_from</i>: Một danh sách để lưu vết đường đi từ ô bắt đầu đến ô hiện tại. • <i>g_score</i> và <i>f_score</i>: Giá trị ban đầu của mỗi ô là vô cực. • <i>closed_set</i>: Lưu trữ các ô đã thăm.
	2	<p>Tạo vòng lặp while, lặp lại cho đến khi <i>open_set</i> rỗng:</p> <ul style="list-style-type: none"> • Trong mỗi lần lặp, lấy ô có <i>f_score</i> nhỏ nhất từ <i>open_set</i> và đánh dấu nó là đã thăm dò (thêm vào <i>closed_set</i>). • Kiểm tra xem ô hiện tại có phải là ô đích hay không. Nếu đúng: <ul style="list-style-type: none"> ◦ Tìm đường đi từ ô đích về ô bắt đầu bằng cách sử dụng <i>find_path</i>. ◦ Trả về kết quả đường đi. • Lấy danh sách các ô lân cận của ô hiện tại • Duyệt qua từng ô lân cận: <ul style="list-style-type: none"> ◦ Nếu ô lân cận đã được thăm dò, bỏ qua nó. ◦ Nếu ô lân cận chưa được thăm dò: tính toán chi phí đến ô lân cận bằng <i>g_score</i>.
	3	<p>Nếu không tìm ra ô kết thúc:</p> <ul style="list-style-type: none"> • In thông báo "Không tìm được đường đi"

Ví dụ:





Phân tích:

- **Tính đầy đủ:** Thuật toán A* là đầy đủ, nghĩa là nếu tồn tại một đường đi từ nút bắt đầu đến nút đích, thuật toán sẽ tìm thấy nó, miễn là hàm heuristic $h(n)$ không bao giờ đánh giá quá cao chi phí để đến mục tiêu.
- **Tính tối ưu:** Thuật toán A* cũng tối ưu, nghĩa là nó sẽ tìm thấy đường đi ngắn nhất từ nút bắt đầu đến nút đích, miễn là hàm heuristic $h(n)$ thỏa mãn điều kiện không quá lớn so với chi phí thực tế (điều kiện này được gọi là "admissible").
- **Độ phức tạp về thời gian:** Độ phức tạp thời gian của A* phụ thuộc vào hàm heuristic $h(n)$. Trong trường hợp xấu nhất, nếu $h(n)$ luôn bằng 0, A* trở thành thuật toán Dijkstra và có độ phức tạp thời gian là $O(|E| + |V|\log|V|)$, trong đó $|E|$ là số cạnh và $|V|$ là số đỉnh trong đồ thị. Tuy nhiên, với một hàm heuristic tốt, A* có thể tìm thấy đường đi ngắn nhất nhanh hơn nhiều.
- **Độ phức tạp của không gian:** Độ phức tạp không gian của A* cũng phụ thuộc vào hàm heuristic $h(n)$. Trong trường hợp xấu nhất, nếu $h(n)$ luôn bằng 0, A* có thể phải lưu trữ tất cả các nút trong đồ thị, do đó độ phức tạp không gian là $O(|V|)$. Tuy nhiên, với một hàm heuristic tốt, A* có thể giảm đáng kể số lượng nút cần lưu trữ.

Nhận xét:

- **Ưu điểm:**
 - Tối ưu và đầy đủ.

- Hiệu quả.
- Linh hoạt.
- Nhược điểm:
 - Phụ thuộc vào hàm heuristic
 - Độ phức tạp không gian cao.
 - Không thể xử lý trọng số âm.

4. Sự khác nhau giữa 3 thuật toán

Thuật toán	Khác
BFS	Khám phá các nút lân cận của một nút hiện tại trước khi chuyển sang các nút lân cận của các nút tiếp theo
DFS	Duyệt sâu vào một nhánh của đồ thị trước khi quay lại và thăm nhánh khác Không đảm bảo tìm ra đường đi ngắn nhất
UCS	Ưu tiên mở rộng đường đi có chi phí thấp nhất Chỉ dựa vào chi phí thực tế từ ô đầu tiên đến ô đích
A*	Kép hợp giữa chi phí từ ô đầu tiên đến thực tại và ước lượng từ hiện tại đến mục tiêu Ưu tiên các đường đi có tổng chi phí ước lượng thấp nhất

III. Mức 3: Đón tất cả các điểm rồi đến trạng thái G

Ý tưởng:

Thuật toán sử dụng phương pháp ghép để tìm đường đi ngắn nhất từ điểm bắt đầu S đến điểm kết thúc G, đi qua các điểm đón (pickup points) theo thứ tự không xác định.

Quy trình:

- 1. Liệt kê tất cả các hoán vị:
 - Sử dụng hàm find_permutations để tạo danh sách chứa tất cả các hoán vị khả thi của các điểm đón.
- 2. Tính toán độ dài đường đi cho mỗi hoán vị:
 - Sử dụng thuật toán AStar để tính toán độ dài đường đi từ S đến từng điểm đón trong một hoán vị, sau đó tính toán đường đi từ điểm đón cuối cùng đến G.
 - Lưu trữ cả đường đi và độ dài tương ứng cho mỗi hoán vị.
- 3. Chọn hoán vị có đường đi ngắn nhất:
 - Tìm hoán vị có độ dài đường đi ngắn nhất trong danh sách đã tính toán.
 - Trả về đường đi ngắn nhất tương ứng với hoán vị đó.

Chi tiết thuật toán:

1. Hàm tìm đường đi ngắn nhất từ S đến G và đi qua tất cả các điểm đón (các điểm đón không phân biệt thứ tự)

Tên hàm	<i>find_path_with_pickup_points_using_matching</i>	
Tham số	g: lớp Grid đại diện cho bản đồ start: ô bắt đầu goal: ô kết thúc Pickup_points: danh sách điểm đón	
Trả về	Danh sách các ô trên đường đi ngắn nhất (bao gồm điểm đầu, điểm đón và điểm kết thúc).	
Thực hiện	1	Liệt kê các hoán vị: <ul style="list-style-type: none"> Sử dụng hàm find_permutations để liệt kê tất cả các hoán vị của danh sách pickup_points. Mỗi hoán vị đại diện cho một thứ tự khác nhau để ghé thăm các điểm đón.
	2	Tính toán độ dài đường đi cho mỗi hoán vị: <ul style="list-style-type: none"> Lặp qua từng hoán vị: <ul style="list-style-type: none"> Khởi tạo một danh sách path để lưu trữ các ô trên đường đi. Thêm ô bắt đầu start vào path. Lặp qua từng điểm đón trong hoán vị: <ul style="list-style-type: none"> Sử dụng thuật toán A* để tìm đường đi ngắn nhất từ ô hiện tại trong path đến điểm đón. Thêm đường đi ngắn nhất được tìm thấy vào path. Sử dụng thuật toán A* để tìm đường đi ngắn nhất từ ô cuối cùng trong path đến ô đích goal. Thêm ô đích goal vào path. Tính toán tổng chi phí của đường đi bằng hàm calculate_cost. Lưu trữ tổng chi phí và path vào các danh sách path_lengths và paths tương ứng.
	3	Chọn hoán vị có đường đi ngắn nhất: <ul style="list-style-type: none"> Tìm chỉ số của hoán vị có đường đi ngắn nhất bằng cách sử dụng hàm min trên danh sách path_lengths. Trả về đường đi ngắn nhất (danh sách path) tương ứng với chỉ số được tìm thấy.

Ví dụ:

Cho 3 bản đồ khác nhau

Phân tích:

- Độ phức tạp thời gian:
 - Thuật toán có độ phức tạp thời gian là $O(n! * (m + n))$, với:
 - n là số lượng điểm đón.

- m là số lượng ô trên đường đi từ S đến G (tính cả điểm bắt đầu và kết thúc).
- Lý do:
 - Việc liệt kê tất cả các hoán vị có độ phức tạp $O(n!)$.
 - Việc tính toán đường đi cho mỗi hoán vị có độ phức tạp $O(m + n)$ do sử dụng thuật toán AStar.
- Độ phức tạp không gian:
 - Thuật toán có độ phức tạp không gian là $O(n!)$, do cần lưu trữ tất cả các hoán vị của điểm đón.

Nhận xét:

- Ưu điểm:
 - Thuật toán tìm được đường đi ngắn nhất bắt đầu từ các điểm đón.
 - Có thể áp dụng cho các bản đồ có nhiều điểm đón.
 - Dễ dàng triển khai và sửa đổi.
- Nhược điểm:
 - Độ phức tạp thời gian khá cao, đặc biệt khi có nhiều điểm đón.
 - Có thể tốn nhiều bộ nhớ cho các bản đồ lớn và nhiều điểm đón.
 - Hiệu quả có thể giảm sút nếu có nhiều chu trình trong bản đồ.

2. Hàm liệt kê tất cả hoán vị của các điểm đón (hàm phụ)

Tên hàm		<i>find_permutations</i>
Tham số	Pickup_points: danh sách điểm đón	
Trả về	Danh sách chứa tất cả các hoán vị của pickup_points.	
Thực hiện	<ul style="list-style-type: none"> 1 2 	<p>Trường hợp cơ bản:</p> <ul style="list-style-type: none"> • Nếu danh sách pickup_points rỗng, hàm sẽ trả về danh sách rỗng []. Lý do là vì không có điểm đón nào để hoán vị. <p>Trường hợp tổng quát:</p> <ul style="list-style-type: none"> • Hàm lặp qua từng điểm trong danh sách pickup_points. • Đối với mỗi điểm hiện tại current_point: <ul style="list-style-type: none"> ○ Tạo danh sách remaining_points bao gồm tất cả các điểm còn lại, trừ current_point. ○ Gọi hàm find_permutations để quy để tìm kiếm tất cả các hoán vị của remaining_points. ○ Lặp qua từng hoán vị con sub_permutation: <ul style="list-style-type: none"> ▪ Tạo một hoán vị mới bằng cách thêm current_point vào đầu sub_permutation. ▪ Thêm hoán vị mới này vào danh sách permutations.

3

trả về danh sách **permutations** chứa tất cả các hoán vị của danh sách **pickup_points**.

IV. Mức 4: Di động hình đa giác

Ý tưởng:

- Cho các đa giác di chuyển lên xuống theo chiều dọc theo vận tốc cùng với vận tốc di chuyển của vật thể (hướng di chuyển của mỗi đa giác là ngẫu nhiên trong mỗi lần chạy). Khi đường đi bị chặn bởi đa giác trong quá trình di chuyển, dừng lại để tìm đường đi mới và tiếp tục.
- Để xử lý trường hợp vật thể đang di chuyển thì đa giác tiến tới và va chạm, đánh dấu các ô phía trước đường đi của vật thể là đã bị chặn để vật thể tránh lựa chọn đường đi “nguy hiểm”.
- Cách tiếp cận này có một nhược điểm, là khi vật thể và đa giác đang tiếp giáp nhau, vì có cùng vận tốc di chuyển nên có thể dẫn đến tình huống vật thể liên tục đi theo đa giác mà không thể vượt qua. Để xử lý tình huống này, tạo một biến đếm `time_out`, nếu tiếp giáp đa giác và đi theo cùng hướng với đa giác đó trong một số frame nhất định, vật thể sẽ dừng lại vài frame để chờ đa giác đi qua, đường đi lúc này sẽ rộng mở.

Quy trình:

- 1. Tìm đường đi khởi đầu
- 2. Với mỗi frame:
 - Cập nhật vị trí mới cho vật thể theo đường đi sẵn có.
 - Cập nhật vị trí mới cho các đa giác.
 - Cập nhật vùng “nguy hiểm” mới cho đa giác.
 - Nếu vị trí tiếp theo của đường đi bị chặn:
 - Tìm lại đường đi mới.
 - Nếu đủ giá trị `time_out`, dừng vật thể lại trong 5 frame.

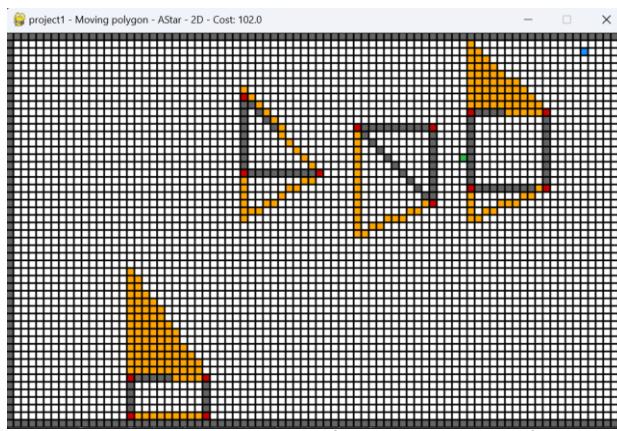
Chi tiết thuật toán:

Tên lớp	<i>Frame</i>	
Tham số (khởi tạo)		
Thuộc tính	<i>time_out</i>	Chức năng
		space: lưới ô vuông của bản đồ sc: đối tượng Surface của pygame
		Biến đếm để dừng lại chờ đa giác đi qua trong trường hợp đi cùng hướng với hướng di chuyển của đa giác

	<i>previous_position</i>	Vị trí trước đó của vật thể (dùng để kiểm tra có đang đi cùng hướng với một đa giác đang chặn đường hay không)
Thực hiện	Hàm update_new_frame()	<p>Cập nhật vị trí mới cho các đa giác khi di chuyển</p> <ul style="list-style-type: none"> Xóa đa giác cũ bằng phương thức reset_polygon() của lớp Polygon. Xóa vùng nguy hiểm của đa giác cũ bằng phương thức reset_unsafe_points() lớp Polygon. Kiểm tra đa giác có đang nằm ở rìa bản đồ hay không, nếu có thì cập nhật lại vị trí các đỉnh cho đa giác để đa giác bật lại về hướng ngược lại (chỉ xử lý cho đa giác có 3 hoặc 4 đỉnh). Cập nhật lại vị trí cho các đỉnh của đa giác theo vận tốc của đa giác (các đa giác đang dùng để mô phỏng chỉ di chuyển lên xuống theo chiều dọc với vận tốc là 1) Gọi các phương thức init_edge(), set_passable_polygon() và update_unsafe_points() của lớp Polygon để khởi tạo đa giác mới. Vẽ đa giác mới bằng phương thức draw() của lớp Grid.
	Hàm update_agent()	<p>Cập nhật vị trí mới cho vật thể tìm đường</p> <ul style="list-style-type: none"> Kiểm tra đường đi đường đi có tồn tại hay không (trong trường hợp đang đợi tìm đường đi mới). Lưu lại vị trí hiện tại vào biến previous_position. Lấy ô vuông tiếp theo từ đường đi. Kiểm tra đã đến đích hay chưa, nếu đã đến thì cập nhật vị trí mới, vẽ lại và trả về chuỗi “arrived”. Kiểm tra ô vuông tiếp theo trong đường đi có khả dụng hay không, nếu có thì cập nhật vị trí mới, xóa ô vuông đó khỏi đường đi, vẽ lại và trả về True. Nếu ô vuông tiếp theo trong đường đi không khả dụng, và nếu vị trí trước đó và vị trí hiện tại có cùng tọa độ x (tức là đang bị đa giác chặn và đang đi cùng hướng với đa giác đó), tăng giá trị time_out lên 1 đơn vị. Nếu time_out đạt đến giá trị 3, đặt lại time_out bằng 0 và trả về chuỗi “delay”.

Tên lớp	Agent
Tham số	space: lưới ô vuông của bản đồ

(khởi tạo)	sc: đối tượng Surface của pygame frame: đối tượng Frame path: mảng chứa đường đi queue_request: hàng chờ dùng để gửi yêu cầu tìm đường queue_response: hàng chờ dùng để nhận đường đi mới
Thuộc tính	Chức năng
	delay Biến dùng để đếm ngược thời gian thời gian dừng lại chờ đa giác đang chặn đi qua (nếu vật thể đi cùng hướng). is_arrived Biến kiểm tra đã đến đích hay chưa
Thực hiện	Hàm update () Cập nhật khung hình mới cho đến khi vật thể đến đích. <ul style="list-style-type: none"> Cập nhật vị trí cho các đa giác bằng phương thức update_new_frame() của lớp Frame. Cập nhật vị trí cho vật thể bằng phương thức update_agent() của lớp Frame. Nếu kết quả trả về là False, tạo yêu cầu tìm đường mới đến đa nhiệm tìm đường. Nếu kết quả trả về là “delay”, gán cho biến delay giá trị là 5 và giảm dần 1 đơn vị sau mỗi vòng lặp, khi đang trong trạng thái delay, update_agent() sẽ không được gọi. Các khung hình cách nhau 100ms bằng cách sử dụng phương thức delay() của pygame.time. Kiểm tra hàng chờ queue_response có gửi đến đường đi mới hay không, nếu có thì lưu lại đường đi mới và hiển thị lên màn hình.

Ví dụ:Link demo: [Link](#)**Phân tích:**

Khung hình chạy thử có hiển thị vùng “nguy hiểm”

- Hình ảnh phía trên là một khung hình từ chương trình có hiển thị vùng “nguy hiểm” (vùng màu cam). Mức độ tối ưu của thuật toán này sẽ phụ thuộc hoàn toàn vào cách xác định vùng này, nếu quá nhỏ sẽ xảy ra nhiều tinh huống va chạm, nhưng nếu quá lớn sẽ làm tăng chi phí di chuyển và tìm đường. Phương án hiện tại đang sử dụng chiều cao của vùng này vào khoảng 1.25 lần chiều ngang của đa giác.
- Vùng “nguy hiểm” ngoài được tạo ra theo hướng di chuyển của đa giác, cũng sẽ được chuyển đổi dần dần theo hướng ngược lại khi đa giác tới gần tường, để đảm bảo khi bật ngược lại sẽ không va chạm với vật thể.

V. Mức 5: Mô hình trên không gian 3D

Ý tưởng:

Raycast là một kỹ thuật dùng để mô phỏng va chạm giữa tia sáng từ 1 điểm đến một vật thể trong môi trường 2D, nhằm tạo ra hiệu ứng 3D trên màn hình. Đầu tiên, ta xác định điểm bắt đầu của tia và hướng của tia. Tiếp theo, ta lan truyền tia bằng cách mở rộng tia từ điểm bắt đầu theo hướng đã xác định qua một vòng lặp. Trong mỗi vòng lặp, tia sẽ kiểm tra xem liệu nó có gặp phải ô không đi qua được hay không. Nếu có, ta vẽ một đường đại diện cho tường lên trên màn hình. Quá trình này được lặp lại cho mỗi tia được phát ra từ điểm quan sát, tạo ra một loạt các đường va chạm, từ đó tạo ra một bức tranh toàn cảnh về môi trường xung quanh.

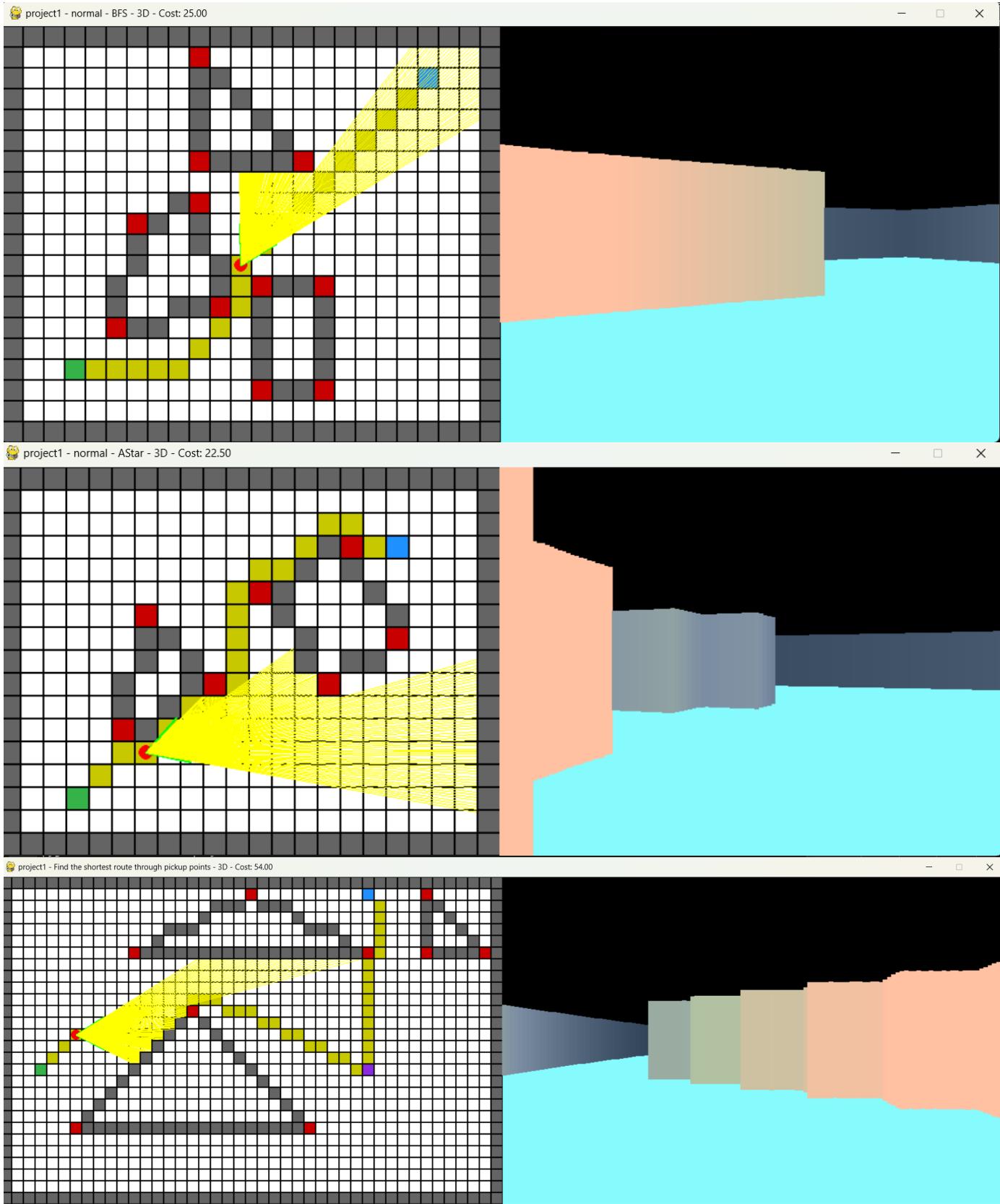
Chi tiết thuật toán:

Tên lớp		Player
Tên biến		Chức năng
	<i>angle</i>	góc quay hiện tại của người chơi
	<i>angle_delta</i>	Số độ mà góc nhìn của người chơi phải thay đổi sau mỗi bước cập nhật
	<i>angle_times</i>	Số lần cần thay đổi góc nhìn khi đang trong quá trình xoay để hướng về ô tiếp theo
	<i>position_delta</i>	Tuple chứa khoảng cách di chuyển theo chiều ngang và chiều dọc mà người chơi cần thực hiện trong mỗi chu kỳ cập nhật để đến ô đích
	<i>position_times</i>	Số chu kỳ cần cập nhật để người chơi hoàn thành chuyển động tới ô tiếp theo
	Hàm <i>Update()</i>	Kiểm tra xoay và chuyển động, sửa đổi vị trí, góc nhìn của người chơi. <ul style="list-style-type: none"> Xoay: Nếu người chơi đang xoay (<i>angle_times</i> != 0 và

	<p>angle_delta != 0), góc của người chơi được cập nhật dần bằng cách sử dụng angle_delta. Biến đếm angle_times được giảm cho đến khi bằng 0, tương đương với việc hoàn thành xoay.</p> <ul style="list-style-type: none"> Chuyển động: Nếu người chơi đang di chuyển (position_times != 0), vị trí của người chơi (x và y) được cập nhật dựa trên các giá trị trong position_delta. Position_times sẽ giảm đến khi bằng 0. Nếu vẫn còn ô trong đường dẫn (<i>dest_cell</i> < len(<i>self.path</i>) - 1), tính toán ô tiếp theo. cell được sử dụng lưu chỉ số cột và hàng của ô đích trong grid. Tính toán bằng cách sử dụng <i>divmod(self.grid.get_cell(self.x, self.y), self.grid.width)</i> và sau đó được đảo ngược để khớp với quy ước (cột, hàng). dest_angle (góc tới cell người chơi cần nhìn tiếp theo) được tính bằng cách sử dụng math.atan2() để xác định góc giữa vị trí hiện tại của người chơi và ô tới. Sau đó, góc được chuyển đổi thành độ bằng cách sử dụng math.degrees(). angle_delta là độ khác biệt giữa góc hiện tại và góc mong muốn để hướng về ô đích. Giới hạn ở mức thay đổi tối đa 4 độ/chu kỳ cập nhật để tránh việc xoay đột ngột. position_dest là tọa độ của ô đích. position_delta được tính toán để đại diện cho sự khác biệt giữa vị trí hiện tại của người chơi và vị trí đích. Nó được chia cho position_times để trải đều chuyển động qua nhiều chu kỳ cập nhật. position_times được đặt thành một giá trị (ví dụ: 30) để xác định số chu kỳ cập nhật cho hoạt ảnh chuyển động.
2	<p>Vẽ người chơi và các tia quét trên màn hình game</p> <ul style="list-style-type: none"> - Vẽ hình tròn đại diện cho người chơi: <ul style="list-style-type: none"> Sử dụng pygame.draw.circle để vẽ một hình tròn có màu đỏ tại vị trí hiện tại của người chơi trên màn hình.

		<ul style="list-style-type: none"> • Hình tròn đại diện cho vị trí của người chơi trong không gian game. - Vẽ các đường thẳng đại diện cho hướng nhìn của người chơi: <p>Sử dụng <code>pygame.draw.line</code> để vẽ ba đoạn thẳng màu xanh lá cây để đại diện cho hướng nhìn của người chơi và phạm vi tầm nhìn.</p>
3		<ul style="list-style-type: none"> - Duyệt qua tia quét: • Sử dụng vòng lặp <code>for ray in range(CASTED_RAYS)</code> để lặp qua từng tia quét. • Trong mỗi tia quét, sử dụng vòng lặp <code>for depth in range(MAX_DEPTH)</code> để tăng chiều dài tia qua mỗi lần lặp - Kiểm tra va chạm: <ul style="list-style-type: none"> • Tính toán tọa độ x và y của điểm cuối mà tia quét được bằng cách nhân cos và sin với chiều dài tia quét (depth). • Kiểm tra xem điểm này có nằm trong một ô hợp lệ hay không bằng cách sử dụng <code>self.grid.get_cell()</code>. • Nếu điểm đó không hợp lệ hoặc ô không đi qua được, có nghĩa là tia quét đã va chạm với tường. - Vẽ tường: <ul style="list-style-type: none"> • Sử dụng <code>pygame.draw.line()</code> để vẽ tia quét từ vị trí của người chơi đến vị trí va chạm. • Tính toán độ sâu thực tế của điểm va chạm và tính toán màu sắc và chiều cao của vật thể được hiển thị. - Vẽ một hình chữ nhật để đại diện cho tường.

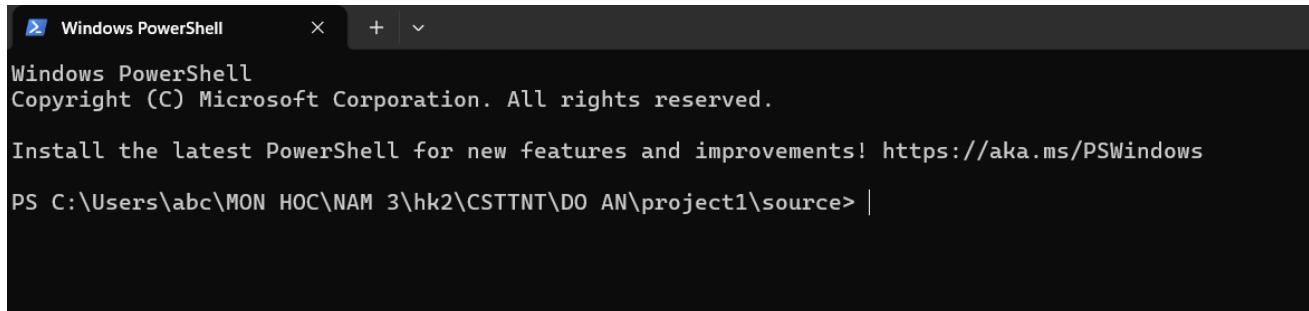
Ví dụ:



VI. Hướng dẫn sử dụng chương trình

⇒ Xem hướng dẫn chi tiết trong file Instructions for using the program.pdf

Để chạy chương trình ta vào thư mục source của dự án và mở cmd



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

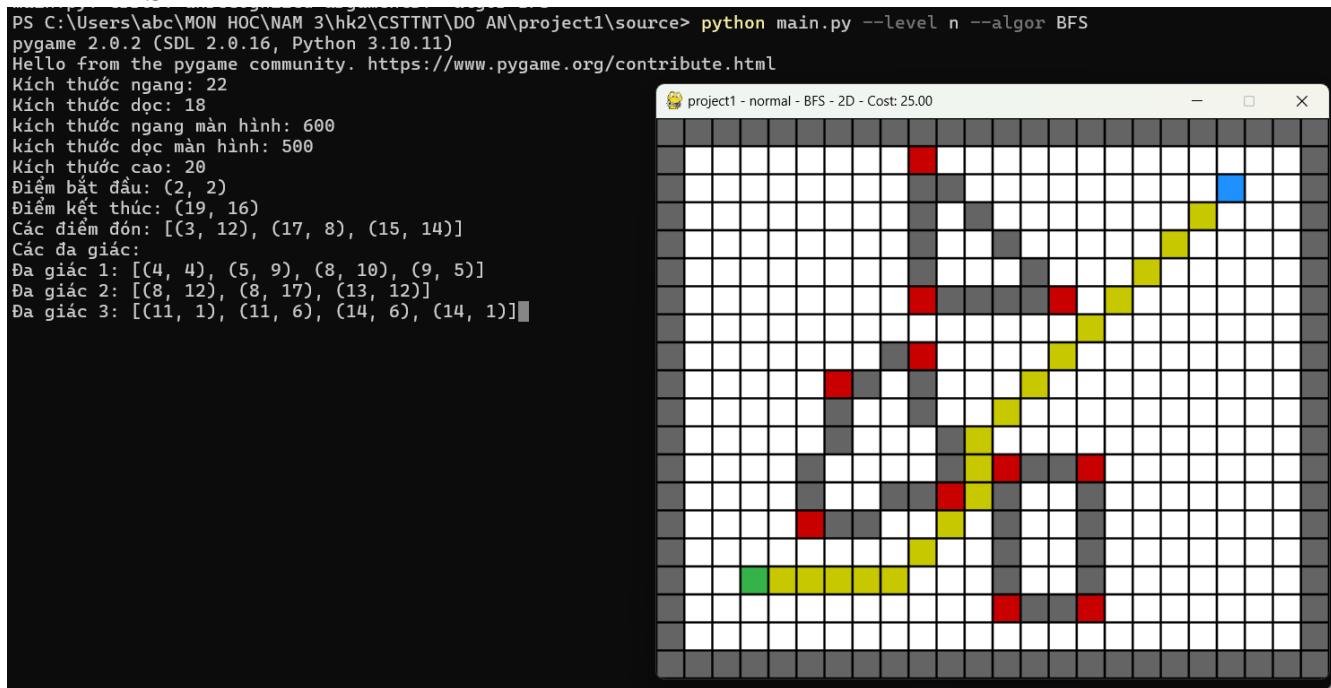
PS C:\Users\abc\MON HOC\NAM 3\hk2\CSTTNT\DO AN\project1\source> |
```

Ta chạy theo cú pháp sau:

python main.py --level n/pk/mp [--algor BFS/DFS/UCS/AStar] [-space 2D/3D]

Ví dụ:

- **python main.py --level n --algor BFS**: chạy ở mức cơ bản (mức 1,2) với thuật toán BFS

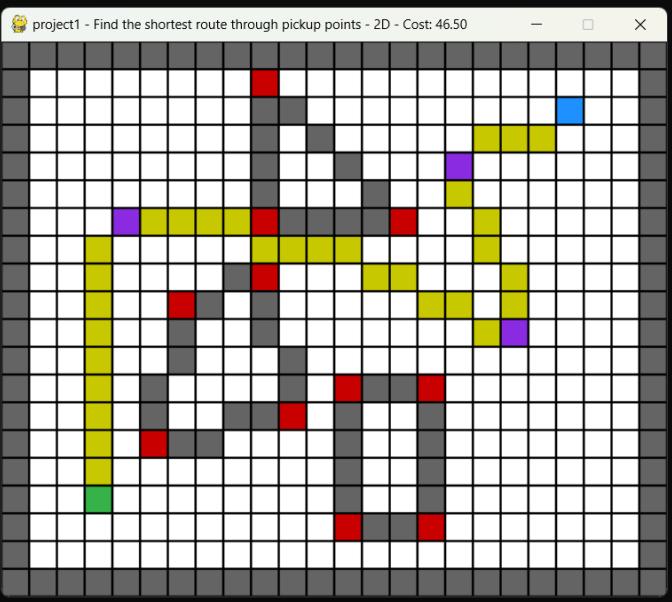


```
PS C:\Users\abc\MON HOC\NAM 3\hk2\CSTTNT\DO AN\project1\source> python main.py --level n --algor BFS
pygame 2.0.2 (SDL 2.0.16, Python 3.10.11)
Hello from the pygame community. https://www.pygame.org/contribute.html
Kích thước ngang: 22
Kích thước dọc: 18
kích thước ngang màn hình: 600
kích thước dọc màn hình: 500
Kích thước cao: 20
Điểm bắt đầu: (2, 2)
Điểm kết thúc: (19, 16)
Các điểm đón: [(3, 12), (17, 8), (15, 14)]
Các đà giác:
Đà giác 1: [(4, 4), (5, 9), (8, 10), (9, 5)]
Đà giác 2: [(8, 12), (8, 17), (13, 12)]
Đà giác 3: [(11, 1), (11, 6), (14, 6), (14, 1)]
```

The screenshot shows a 2D grid search visualization titled "project1 - normal - BFS - 2D - Cost: 25.00". The grid is 22x20. It contains several obstacles represented by red and grey blocks. A path is highlighted in yellow, starting from the bottom-left corner (11, 1) and ending at the top-right corner (19, 16). The path consists of several segments, including a vertical segment on the left, a horizontal segment at the bottom, and a diagonal segment in the middle-right area.

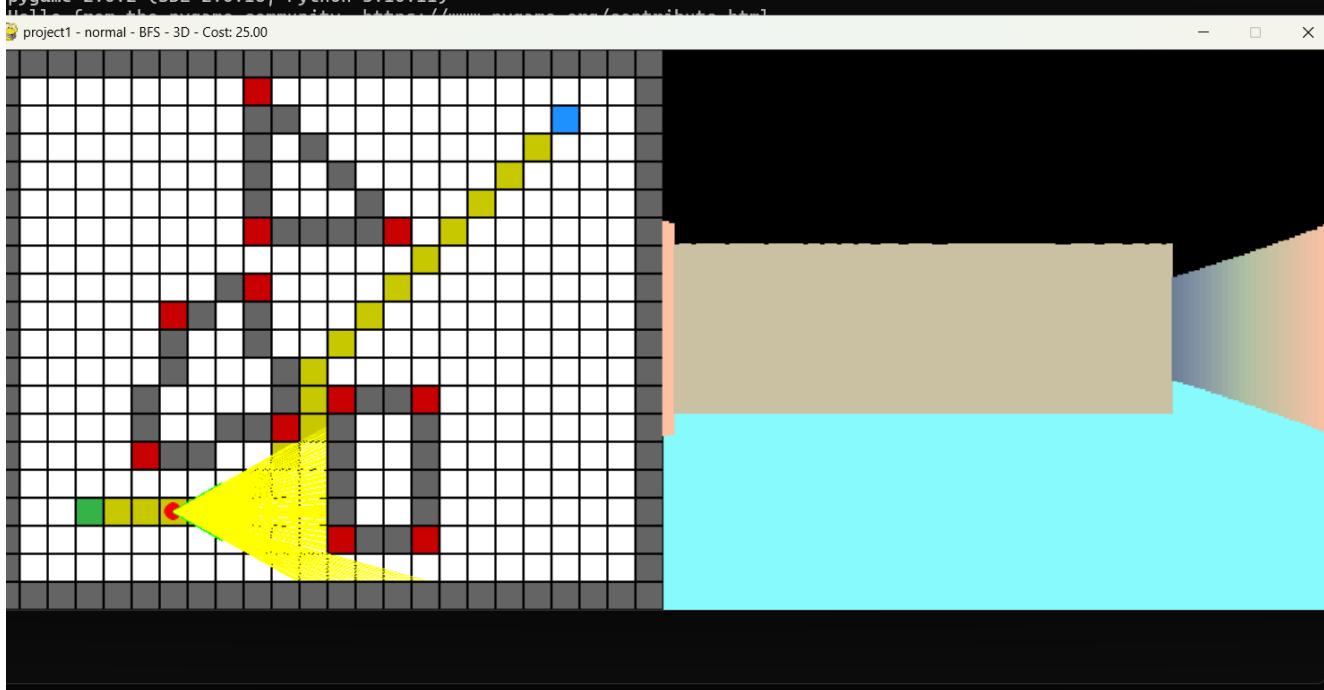
- **python main.py --level pk**: tìm đường đi ngắn nhất từ điểm đầu đến các điểm đón sau đó đến điểm cuối

```
PS C:\Users\abc\MON HOC\NAM 3\hk2\CSTTNT\DO AN\project1\source> python main.py --level pk
pygame 2.0.2 (SDL 2.0.16, Python 3.10.11)
Hello from the pygame community. https://www.pygame.org/contribute.html
Kích thước ngang: 22
Kích thước dọc: 18
kích thước ngang màn hình: 600
kích thước dọc màn hình: 500
Kích thước cao: 20
Điểm bắt đầu: (2, 2)
Điểm kết thúc: (19, 16)
Các điểm đón: [(3, 12), (17, 8), (15, 14)]
Các đa giác:
Đa giác 1: [(4, 4), (5, 9), (8, 10), (9, 5)]
Đa giác 2: [(8, 12), (8, 17), (13, 12)]
Đa giác 3: [(11, 1), (11, 6), (14, 6), (14, 1)]
```

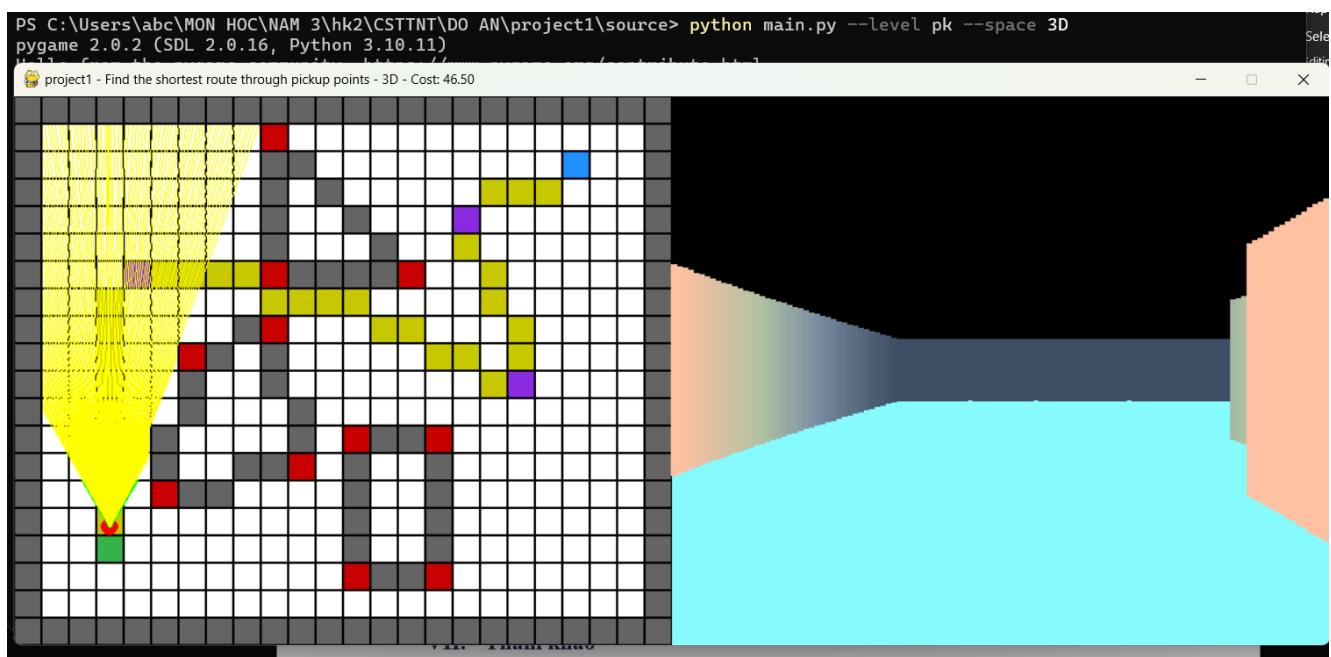


- **python main.py --level mp --algor UCS:** chạy thuật toán UCS trong trường hợp các đa giác di chuyển
- Link demo: [Link](#)
- **python main.py --level n --algor BFS --space 3D:** chạy thuật toán BFS ở mức độ cơ bản(mức 1, 2) trong không gian 3 chiều

```
PS C:\Users\abc\MON HOC\NAM 3\hk2\CSTTNT\DO AN\project1\source> python main.py --level n --algor BFS --space 3D
pygame 2.0.2 (SDL 2.0.16, Python 3.10.11)
```



- **python main.py --level pk --space 3D:** chạy thuật toán tìm đường đi ngắn nhất qua các điểm đón trong không gian 3 chiều



VII. Tham khảo

1. [Search Algorithms in AI - GeeksforGeeks](#)
2. <https://www.baeldung.com/cs/uniform-cost-search-vs-best-first-search#:~:text=Completeness%20of%20UCS,UCS%20will%20also%20be%20complete.>
3. <https://killerrobotics.me/2021/08/13/raycasting-game-in-python-and-pygame-part-1/>

