

ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

BỘ MÔN KHOA HỌC MÁY TÍNH

BÁO CÁO ĐỒ ÁN

ĐỒ ÁN 1: CÁC THUẬT TOÁN TÌM KIẾM

MÔN HỌC: CƠ SỞ TRÍ TUỆ NHÂN TẠO

Sinh viên thực hiện:

Nguyễn Hải Chấn (21120006)
Võ Trung Hoàng Hưng(21120011)
Lê Nguyễn Phương Thùy (21120146)

Giáo viên hướng dẫn:

GS. TS. Lê Hoài Bắc
Thầy Nguyễn Duy Khánh

Ngày 1 tháng 11 năm 2023



Mục lục

1 Thông tin chung về đồ án	2
1.1 Thông tin thành viên	2
1.2 Đánh giá mức độ hoàn thành	2
2 Giải thích và mô tả cách cài đặt từng thuật toán	2
2.1 Thuật toán tìm kiếm DFS (Depth First Search)	2
2.2 Thuật toán tìm kiếm BFS (Breadth First Search)	3
2.3 Thuật toán tìm kiếm UCS (Uniform-Cost Search)	4
2.4 Thuật toán tìm kiếm tham lam (Greedy Best First Search)	5
2.5 Thuật toán tìm kiếm A*	6
2.6 Các thuật toán cho bản đồ có điểm thưởng	7
2.6.1 Thuật toán Dijkstra:	7
2.6.2 Thuật toán Quy hoạch động trạng thái:	9
2.6.3 Thuật toán A* áp dụng cho Level 2 (AStart_Lv2):	10
2.7 Các thuật toán cho bản đồ có điểm đón	11
2.7.1 Thuật giải di truyền:	11
2.7.2 Thuật toán leo đồi:	13
2.8 Thuật toán cho bản đồ nâng cao - Teleport	14
3 Các kịch bản và kết quả chạy	14
3.1 Level 1: Bản đồ không có điểm thưởng	15
3.1.1 Kịch bản 1 - Bản đồ không có điểm thưởng	15
3.1.2 Kịch bản 2 - Bản đồ không có điểm thưởng	18
3.1.3 Kịch bản 3 - Bản đồ không có điểm thưởng	22
3.1.4 Kịch bản 4 - Bản đồ không có điểm thưởng	26
3.1.5 Kịch bản 5 - Bản đồ không có điểm thưởng	32
3.2 Bản đồ có điểm thưởng	37
3.2.1 Kịch bản 1 - Bản đồ có điểm thưởng:	37
3.2.2 Kịch bản 2 - Bản đồ có điểm thưởng:	39
3.2.3 Kịch bản 3 - Bản đồ có điểm thưởng:	42
3.3 Bản đồ có các điểm đón - Mức 3	45
3.3.1 Kịch bản 1 - Bản đồ có điểm đón:	45
3.3.2 Kịch bản 2 - Bản đồ có điểm đón:	46
3.3.3 Kịch bản 3 - Bản đồ có điểm đón:	47
3.4 Kịch bản nâng cấp	49
3.4.1 Thiết kế bản đồ và quy tắc về điểm dịch chuyển	49
3.4.2 Kịch bản 1 - Bản đồ có điểm dịch chuyển:	50
3.4.3 Kịch bản 2 - Bản đồ có điểm dịch chuyển:	50
3.4.4 Kịch bản 3 - Bản đồ có điểm dịch chuyển:	51
4 Video minh họa	51
5 Mô tả các câu lệnh trong file run.sh	52
Tài liệu tham khảo	53

1 Thông tin chung về đồ án

1.1 Thông tin thành viên

Mã số sinh viên	Họ và tên	Liên hệ
21120006	Nguyễn Hải Chấn	21120006@student.hcmus.edu.vn
21120011	Võ Trung Hoàng Hưng	21120011@student.hcmus.edu.vn
21120146	Lê Nguyễn Phương Thùy	21120146@student.hcmus.edu.vn

1.2 Đánh giá mức độ hoàn thành

Nội dung	Mức độ hoàn thành
Mức 1a	100%
Mức 1b	100%
Mức 2a	100%
Mức 2b	100%
Mức 3	100%
Kịch bản nâng cấp	100%
Video minh họa	100%

2 Giải thích và mô tả cách cài đặt từng thuật toán

2.1 Thuật toán tìm kiếm DFS (Depth First Search)

Thuật toán DFS (Depth First Search - Tìm kiếm theo chiều sâu) là một thuật toán tìm kiếm không có thông tin sử dụng stack hoặc đệ quy để tìm kết quả. DFS sẽ cố gắng tìm kiếm đỉnh xa nhất mà nó có thể đạt được. Khi đến đỉnh xa nhất có thể đạt được rồi, nếu như đó là điểm mà chúng ta cần tìm thì trả về đường đi từ đỉnh bắt đầu đến đỉnh đích. Nếu không phải, sẽ quay lui trở về các đỉnh trước đó để tiến hành tìm con đường khác. Thuật toán DFS đảm bảo tìm ra một đường đi tới đích (không tối ưu), nếu đường đi đó tồn tại.

Cách cài đặt:

- **Bước 1:** Khởi tạo
 - Start: trạng thái ban đầu, là ví trí xuất phát trên bản đồ
 - End: trạng thái kết thúc, là lối ra trên bản đồ
 - Matrix: Matrix là mảng hai chiều biểu diễn bản đồ, chứa các vị trí không thể đi được (tường) và các vị trí có thể đi đến được.
 - Visited: danh sách các nút đã đi qua. Ban đầu Visited là danh sách rỗng.
 - Trace: là danh sách truy vết, dùng để tìm lại đường đi khi đã duyệt tới trạng thái đích. Ban đầu Trace[Start] = None.

- **Bước 2:** Gọi hàm đệ quy DFS(current_node) với current_node là một tuple chứa thông tin tọa độ vị trí hiện tại đang đi.
- **Bước 3:** Bên trong hàm đệ quy DFS(), đầu tiên thêm ô hiện tại (current_node) vào Visited để đánh dấu là vị trí này đã đi qua rồi. Tiếp theo duyệt theo 4 hướng lên, xuống, trái, phải để đi lấy tọa độ 4 ô lân cận. Với mỗi ô lân cận, nếu tọa độ ô đang xét không vượt ra ngoài mê cung và chưa từng được đi qua thì sẽ có đường đi từ ô hiện tại đến ô lân cận đó. Vì vậy ta sẽ đánh dấu truy vết cho biết ô liền trước new_node chính là current_node và gọi đệ quy DFS(new_node) với new_node chính là tọa độ của ô lân cận để chuyển tiếp trạng thái từ ô ban đầu.
- **Bước 4:** Sau khi tìm được đường đi đến Goal thì ta sẽ cần truy vết:
 - Khởi tạo Route: danh sách đường đi
 - Thêm Current vào Route. Gán Current cho Trace[Current] (nút cha của Current). Lặp lại thao tác này cho đến khi Current = None (Đã truy vết đết trạng thái đầu).
 - Cost: chi phí đường đi: vì trong bài toán này, mỗi bước đi có chi phí như nhau và bằng 1 nên Cost được gán cho tổng số nút trong Route - 1 (không tính Start).
 - Trả về Route, Cost, và Visited. Kết thúc thuật toán

Độ phức tạp về thời gian: $O(\text{height} * \text{width})$.

Độ phức tạp về không gian: $O(\text{height} * \text{width})$.

Trong đó, height và width lần lượt chính là chiều cao và chiều rộng của mê cung.

2.2 Thuật toán tìm kiếm BFS (Breadth First Search)

Thuật toán BFS (Breadth First Search - Tìm kiếm theo chiều rộng) là một thuật toán tìm kiếm không có thông tin. Đặc điểm của nó là chuyển từ trạng thái này sang trạng thái khác theo cơ chế duyệt hết tất cả các đỉnh cùng mức trước khi chuyển qua mức kế tiếp. Thuật toán BFS đảm bảo tìm ra một đường đi ngắn nhất tới đích trong đồ thị không có trọng số, nếu đường đi đó tồn tại.

Cách cài đặt:

- **Bước 1:** Khởi tạo
 - Start: trạng thái ban đầu, là ví trí xuất phát trên bản đồ
 - End: trạng thái kết thúc, là lối ra trên bản đồ
 - Matrix: Matrix là mảng hai chiều biểu diễn bản đồ, chứa các vị trí không thể đi được (tường) và các vị trí có thể đi đến được.
 - Frontier: danh sách các trạng thái có thể đi tới trong suốt quá trình. Trong BSF, Frontier là một queue với cơ chế First-In-First-Out. Ban đầu, thêm Start vào Frontier
 - Visited: danh sách các nút đã đi qua. Ban đầu Visited là danh sách rỗng.
 - Trace: là danh sách truy vết, dùng để tìm lại đường đi khi đã duyệt tới trạng thái đích. Ban đầu Trace[Start] = None.

- **Bước 2:** Nếu Frontier rỗng, thông báo "Không có đường đi thỏa mãn". Kết thúc chương trình. Nếu không, chuyển tới **Bước 3**.
- **Bước 3:** Di tới trạng thái kế tiếp cùng mức hoặc mức tiếp theo sau khi hoàn thành mức hiện tại: lấy trạng thái cũ nhất từ Frontier ra và gán cho biến Current (Trạng thái hiện tại). Nếu Current = End, chuyển sang **Bước 4**, nếu không, thêm Current vào Visited, chuyển sang **Bước 5**.
- **Bước 4:** Truy vết:
 - Khởi tạo Route: danh sách đường đi
 - Thêm Current vào Route. Gán Current cho Trace[Current] (nút cha của Current). Lặp lại thao tác này cho đến khi Current = None (Đã truy vết đến trạng thái đầu).
 - Cost: chi phí đường đi: vì trong bài toán này, mỗi bước đi có chi phí như nhau và bằng 1 nên Cost được gán cho tổng số nút trong Route - 1 (không tính Start).
 - Trả về Route, Cost, và Visited. Kết thúc thuật toán
- **Bước 5:** Mở các trạng thái liền kề:
 - Tìm Neighbors: tất cả các trạng thái có thể đi tới được từ Current và chưa nằm trong Visited.
 - Thêm Neighbors vào Frontier.
 - Với mỗi neighbor, gán Trace[neighbor] = Current (trạng thái cha của neighbor là Current)
 - Thêm Current vào Visited. Quay lại **Bước 2**.

Độ phức tạp về thời gian: $O(\text{height} * \text{width})$.

Độ phức tạp về không gian: $O(\text{height} * \text{width})$.

Trong đó, height và width lần lượt chính là chiều cao và chiều rộng của mê cung.

2.3 Thuật toán tìm kiếm UCS (Uniform-Cost Search)

Thuật toán UCS (Uniform-Cost Search) là một thuật toán tìm kiếm không có thông tin được thiết kế để tìm đường đi ngắn nhất giữa hai đỉnh. UCS tìm kiếm đường đi có tổng trọng số nhỏ nhất bằng cách mở rộng đỉnh có tổng trọng số thấp nhất trước hết trong số các đỉnh có thể đi tới. Thuật toán UCS đảm bảo tìm ra một đường đi ngắn nhất tới đích, nếu đường đi đó tồn tại.

Cách cài đặt:

- **Bước 1:** Khởi tạo
 - Start: trạng thái ban đầu, là ví trí xuất phát trên bản đồ. Trạng thái ban đầu có chi phí là 0.
 - End: trạng thái kết thúc, là lối ra trên bản đồ
 - Matrix: Matrix là mảng hai chiều biểu diễn bản đồ, chứa các vị trí không thể đi được (tường) và các vị trí có thể đi đến được. Vì trong bài toán được đề cập trong đồ án, tất cả các bước đi đều có chi phí bằng nhau và bằng 1 nên Matrix không cần lưu trữ thêm chi phí cho các bước đi.

- Frontier: danh sách các trạng thái có thể đi tới và chi phí để đi tới trạng thái đó trong suốt quá trình. Trong UCS, Frontier là một Priority Queue với cơ chế trả về phần tử có trọng số nhỏ nhất. Ban đầu, thêm Start vào Frontier.
- Visited: danh sách các nút đã đi qua. Ban đầu Visited là danh sách rỗng.
- Trace: là danh sách truy vết, dùng để tìm lại đường đi khi đã duyệt tới trạng thái đích. Ban đầu Trace[Start] = None.

- **Bước 2:** Nếu Frontier rỗng, thông báo "Không có đường đi thỏa mãn". Kết thúc chương trình. Nếu không, chuyển tới **Bước 3**.
- **Bước 3:** Di tới trạng thái kế tiếp có chi phí nhỏ nhất: lấy trạng thái có chi phí nhỏ nhất từ Frontier ra và gán cho biến Current (Trạng thái hiện tại). Nếu Current = End, chuyển sang **Bước 4**, nếu không, thêm Current vào Visited, chuyển sang **Bước 5**.
- **Bước 4:** Truy vết:
 - Khởi tạo Route: danh sách đường đi
 - Thêm Current vào Route. Gán Current cho Trace[Current] (nút cha của Current). Lặp lại thao tác này cho đến khi Current = None (Đã truy vết đết trạng thái đầu).
 - Cost: chi phí đường đi: vì trong bài toán này, mỗi bước đi có chi phí như nhau và bằng 1 nên Cost được gán cho tổng số nút trong Route - 1 (không tính Start).
 - Trả về Route, Cost, và Visited. Kết thúc thuật toán.
- **Bước 5:** Mở các trạng thái liền kề:
 - Tìm Neighbors: tất cả các trạng thái có thể đi tới được từ Current và chưa nằm trong Visited. Với mỗi neighbor, gán chi phí bằng chi phí của Current + 1.
 - Thêm Neighbors vào Frontier.
 - Với mỗi neighbor, gán Trace[neighbor] = Current (trạng thái cha của neighbor là Current)
 - Thêm Current vào Visited. Quay lại **Bước 2**.

Lưu ý: Khi trạng thái End được thêm vào Frontier (enqueue), thuật toán vẫn chưa dừng lại. Thuật toán chỉ dừng khi End được lấy ra khỏi Frontier (dequeue), khi đó đường đi tìm được đến End là đường đi ngắn nhất.

2.4 Thuật toán tìm kiếm tham lam (Greedy Best First Search)

Thuật toán GBFS (Greedy Best First Search) là một thuật toán tìm kiếm có thông tin. Nếu như BFS, DFS và UCS (các thuật toán tìm kiếm không có thông tin) dựa vào thông tin từ thời điểm hiện tại trở về trước (backward cost) để quyết định bước đi, thì GBFS thuộc loại thuật toán tìm kiếm có thông tin, dựa vào thông tin "tương lai" (forward cost) để quyết định đường đi.

GBFS chọn bước đi tiếp theo dựa trên một hàm đánh giá (Heuristic) mà nó nghĩ sẽ đưa đến mục tiêu cuối cùng một cách nhanh nhất. GBFS có tính tham lam: nó không xem xét những bước đã đi qua và chỉ tập trung vào các bước đi lân cận và mục tiêu. Điều này có thể dẫn đến việc bỏ sót các lựa chọn tốt mà có thể dẫn đến một đường đi ngắn hơn, thậm chí có thể khiến thuật toán rơi

vào trạng thái tối ưu cục bộ và không tìm được đường đi đến đích.

Cách cài đặt:

- **Hàm đánh giá (Heuristic):** được sử dụng để tính giá trị đánh giá nhằm đưa quan đoán: nên lựa chọn bước nào là bước đi tiếp theo. Hàm heuristic có thể được xây dựng dựa trên các dạng khoảng cách. Dau đây là 2 dạng khoảng cách thường dùng trong các hàm Heuristic giá là:

1. Euclidean distance:

$$h(x, y) = \sqrt{(x[0] - y[0])^2 + (x[1] - y[1])^2}$$

2. Manhattan distance:

$$h(x, y) = |(x[0] - y[0])| + |(x[1] - y[1])|$$

- **Hàm GBFS:**

- **Bước 1:** Khởi tạo danh sách Visited để theo dõi các trạng thái đã được duyệt qua.
- **Bước 2:** Khởi tạo nút hiện tại (Current) với trạng thái ban đầu (Start)
- **Bước 3:** Bắt đầu vòng lặp chính cho đến khi tìm thấy đường đi hoặc không còn đường đi nào khả thi. Trong vòng lặp thực hiện:
 1. Kiểm tra xem trạng thái hiện tại có phải là trạng thái đích (End) không. Nếu có, tìm và trả về đường đi từ Start đến End bằng phương pháp truy vết. Chí phí của đường đi bằng tổng số các nút đã đi qua, không tính nút Start.
 2. Nếu Current không phải trạng thái đích, thêm Current vào danh sách Visited.
 3. Tìm các hàng xóm của trạng thái hiện tại (các trạng thái có thể đi tới được từ Current và chưa nằm trong Visited). Tạo một hàng đợi ưu tiên (PriorityQueue) để lưu trữ các Neighbors, sắp xếp chúng theo giá trị của hàm Heuristic (1 trong 2 hàm Heuristic nêu trên).
 4. Nếu không còn hàng xóm nào trong hàng đợi ưu tiên, thông báo rằng không có đường đi khả thi và trả về đường đi đã tìm thấy và danh sách Visited.
 5. Nếu vẫn còn hàng xóm trong hàng đợi ưu tiên, chọn hàng xóm có hàm đánh giá Heuristic thấp nhất làm trạng thái tiếp theo. Quay lại đầu vòng lặp.

2.5 Thuật toán tìm kiếm A*

Thuật toán A* thuộc loại thuật toán tìm kiếm có thông tin. Nhìn chung, A* là một tổng hợp của UCS và GBFS, có nghĩa là nó sử dụng cả thông tin backward và thông tin forward trong việc quyết định bước đi tiếp theo. Thuật toán A* có khả năng đưa ta đến một lời giải tốt trong thời gian ngắn, nhờ đó nó được ứng dụng rộng rãi trong nhiều lĩnh vực như game, map, ... Thuật toán A* thuộc loại Best-First-Search, nghĩa là nó tìm được một đường đi đầu tiên tốt nhất, nhưng không đảm bảo là đường đi ngắn nhất.

Cách cài đặt:

- **Hàm Heuristic:** Sử dụng lại hai hàm Heuristic được mô tả trong phần GBFS
- **Hàm đánh giá:** Kết hợp chi phí g và đánh giá h :

$$f(x) = g(x) + h(x),$$

trong đó $f(x)$ là hàm đánh giá của trạng thái x , $g(x)$ là chi phí để đi tới x , $h(x)$ là giá trị Heuristic của x .

- **Hàm AStar:**

- **Bước 1:** Khởi tạo Start, End, Visited để theo dõi các trạng thái đã được duyệt.
- **Bước 2:** Khởi tạo hàng đợi ưu tiên (PriorityQueue) để lưu các nút có thể đi đến. Thêm trạng thái hiện tại ban đầu (Start) vào hàng đợi. Start có giá trị $g = 0, h = 0, f = g + h = 0$.
- **Bước 3:** Bắt đầu vòng lặp chính cho đến khi tìm thấy đường đi hoặc không còn đường đi nào khả thi. Trong vòng lặp thực hiện:
 - Kiểm tra nếu hàng đợi rỗng, thông báo "Không có đường đi đến đích" và dừng chương trình.
 - Lấy trạng thái có hàm đánh giá nhỏ nhất trong hàng đợi ưu tiên ra làm trạng thái hiện tại (Current).
 - Kiểm tra xem Current có bằng End không. Nếu có, tìm và trả về đường đi từ Start đến End bằng phương pháp truy vết. Chí phí của đường đi bằng tổng số các nút đã đi qua, không tính nút Start.
 - Nếu Current không phải trạng thái đích, thêm Current vào danh sách Visited.
 - Tìm các hàng xóm (Neighbor) của trạng thái hiện tại (các trạng thái có thể đi tới được từ Current và chưa nằm trong Visited). Thêm các Neighbors vào hàng đợi ưu tiên, sắp xếp chúng theo giá trị của hàm đánh giá $f = h + g$, với h là Heuristic của Neighbor, g là chi phí để đi tới Neighbor. Quay lại đầu vòng lặp.

2.6 Các thuật toán cho bản đồ có điểm thưởng

Ở bản đồ mức Level 1, các bản đồ có 1 điểm xuất phát, 1 điểm đích và không có điểm thưởng nào. Trong Level 2, ngoài điểm xuất phát và điểm đích, ta còn có các điểm thưởng: đi qua mỗi điểm thưởng sẽ được giảm chi phí đường đi một số điểm nhất định. Mục tiêu của ta là tìm đường đi đến đích sao cho chi phí đường đi nhỏ nhất.

2.6.1 Thuật toán Dijkstra:

Ý tưởng:

Ý tưởng chính cho bản đồ này là chúng ta sẽ sử dụng thuật toán Dijkstra, một thuật toán gần giống với Uniform-Cost Search (USC là cải tiến của Dijkstra). Tuy nhiên, thuật toán Dijkstra ban đầu không thể giải quyết bài toán đồ thị có trọng số âm và bản đồ này chỉ cho phép thưởng một lần với mỗi điểm thưởng. Vì vậy ta cần cải tiến nó bằng cách thêm một thông tin trạng thái cho mỗi đỉnh, ngoài hai thông tin ban đầu là tọa độ và chi phí.

Trạng thái ở đây nghĩa là ta xem các điểm thường như các bóng đèn, có thể tắt hoặc bật hay cũng như cách hoạt động của chuỗi nhị phân, chỉ có 0 và 1. Để biểu diễn trạng thái cho n điểm thường trên bản đồ, ta dùng một số nguyên và khi số nguyên đó bằng $2^n - 1$ (n là số lượng điểm thường) nghĩa là tất cả bit trạng thái đều được bật. Nếu bit trạng thái tại vị trí index của một điểm thường được bật thì tức là đường đi hiện tại đang duyệt đã đi qua điểm thường đó và sẽ không đi qua lại lần nữa. Việc có thêm thông số trạng thái này vừa giúp tránh trường hợp đường đi lợi dụng điểm thường để đi qua lại nhiều lần vừa giúp không bị sót mất một số đường đi tối ưu mà khi dùng Dijkstra cơ bản gặp phải.

Như vậy cách làm này là một cách làm tối ưu và có thể tìm được đường đi ngắn nhất trong bản đồ có điểm thường nếu tồn tại đường đi.

Cách cài đặt:

- **Bước 1:** Khởi tạo

- Start: trạng thái ban đầu, là ví trị xuất phát trên bản đồ.
- End: trạng thái kết thúc, là lối ra trên bản đồ.
- Matrix: Matrix là mảng hai chiều biểu diễn bản đồ, chứa các vị trí không thể đi được (tường) và các vị trí có thể đi đến được. Vì trong bài toán được đề cập trong đồ án, tất cả các bước đi đều có chi phí bằng nhau và bằng 1 nên Matrix không cần lưu trữ thêm chi phí cho các bước đi.
- Frontier: danh sách các trạng thái có thể đi tới và chi phí để đi tới trạng thái đó trong suốt quá trình. Trong Dijkstra, Frontier là một PriorityQueue với cơ chế trả về phần tử có trọng số nhỏ nhất. Trong trường hợp lần này mỗi trạng thái trong Frontier sẽ chứa 3 thông tin là tọa độ một ô, trọng số và bit trạng thái.
- Visited: danh sách các nút đã đi qua. Ban đầu Visited là danh sách rỗng.
- Trace: là danh sách truy vết, dùng để tìm lại đường đi khi đã duyệt tới trạng thái đích. Ban đầu Trace[Start] = None.
- Tạo 2 mảng lưu trữ cost[posX][posY][state] và trace[posX][posY][state] lần lượt là chi phí và điểm truy vết của ô tọa độ (posX, posY) với trạng thái state. Để giải thích rõ hơn thì state là một số nguyên biểu diễn chuỗi nhị phân n bit (n là số lượng điểm thường), trong đó nếu $bit_i = 1$ thì nghĩa là đã đi qua điểm thứ i ($0 \leq i \leq n - 1$), ngược lại là chưa đi qua. Ban đầu chi phí của ô Start với trạng thái trống (bằng 0) là 0 và điểm truy vết của ô Start là None.

- **Bước 2:** Chạy vào vòng while với điều kiện Frontier không được rỗng.

- **Bước 3:** Di tới trạng thái kế tiếp có chi phí nhỏ nhất: lấy trạng thái có chi phí nhỏ nhất từ Frontier ra, thêm tọa độ của trạng thái đó vào Visited nếu chưa đi qua lần nào.

- **Bước 4:** Kiểm tra các tọa độ liền kề:

- Duyệt theo 4 hướng lên, xuống, trái, phải để đi lấy tọa độ 4 ô lân cận. Với mỗi ô lân cận, kiểm tra có phải là 1 điểm thường hay không. Nếu ô đó là một điểm thường và kiểm tra bit trạng thái cho thấy chưa đi qua điểm thường đó thì kiểm tra chi phí ở trạng thái hiện tại cộng với chi phí để đi đến ô lân cận nếu bé hơn chi phí của ô lân cận đó lưu

trong mảng cost với trạng thái new_state thì cập nhật lại giá trị thấp nhất cho chi phí ô lân cận đó với trạng thái new_state cùng với đó là đánh dấu truy vết và cuối cùng là thêm trạng thái mới đó vào Frontier. **Lưu ý:** new_state chính là trạng thái của state hiện tại sau khi bắt bit index của điểm thường. Nếu ô lân cận không phải điểm thường thì chỉ cần thêm trạng thái mới từ ô lân cận đó nếu phát hiện có chi phí thấp hơn và giữ nguyên trạng thái state.

- **Bước 5:** Sau khi kết thúc quá trình tìm kiếm thì duyệt qua hết $2^n - 1$ trạng thái (n là số lượng điểm thường) để tìm chi phí tốt nhất đi đến Goal và thực hiện truy vết:

- Khởi tạo Route: danh sách đường đi
- Thêm Current vào Route. Gán Current cho Trace[Current][state] (nút cha của Current). Nếu ô trước đó là một điểm thường thì tắt bit index của điểm thường đó. Lặp lại thao tác này cho đến khi Current = None (Đã truy vết đến trạng thái đầu).

- **Bước 6:** Trả về Route, Visited và chi phí của điểm đích với trạng thái tốt nhất. Kết thúc thuật toán.

Độ phức tạp về thời gian: $O(2^N * \log N)$.

Độ phức tạp về không gian: $O(2^N * height * width)$.

Trong đó, N là số lượng điểm thường trên bản đồ; height và width lần lượt là chiều cao chiều rộng của mê cung.

2.6.2 Thuật toán Quy hoạch động trạng thái:

Ý tưởng:

Ý tưởng của thuật toán này là thay vì đi tìm đường đi như thông thường (đi từ từ qua những tọa độ lân cận) thì ta sẽ đi tìm thứ tự thăm các điểm thường sao cho tối ưu nhất, có thể không cần đi qua bất kỳ điểm thường nào hoặc không cần đi qua toàn bộ. Để làm được điều này thì ta sẽ dùng thuật toán quy hoạch động trạng thái với cách sử dụng trạng thái giống như thuật toán Dijkstra trên. Trong đồ án này sử dụng kỹ thuật quy hoạch động bằng đệ quy có nhớ (Memoization, một kỹ thuật giảm thời gian chạy đệ quy xuống ngang độ phức tạp thời gian với quy hoạch động bình thường bằng cách sử dụng lại kết quả của những trạng thái đã được tính toán trước đó) để tìm ra được chi phí tối ưu nhất là khi đi qua các điểm thường theo một thứ tự nào đó và truy vết lại thứ tự thăm các điểm thường sau khi đã thực hiện xong hàm đệ quy.

Đây cũng là một cách làm tối ưu, tức là sẽ hoàn toàn có thể tìm được đường đi ngắn nhất trong bản đồ có điểm thường nếu tồn tại được đường đi.

Cách cài đặt:

- Gọi số lượng điểm thường là N ($N \leq 10$).
- Tính trước khoảng cách chi phí và đường đi giữa mỗi cặp điểm thường với nhau, cũng như chi phí và đường đi đi từ Start đến các điểm thường và từ các điểm thường đến Exit bằng cách dùng thuật toán BFS từ một điểm đến điểm kia. Ở đây ta chọn BFS là vì ta đang giải quyết một bài toán tối ưu nên nếu dùng A* thì sẽ có xác suất không chính xác và UCS hay Dijkstra thì sẽ tốn chi phí về thời gian chạy hơn.

- Khởi tạo mảng $dp[idx][state][isFinish]$ sử dụng để lưu kết quả trong quy hoạch động với ý nghĩa là chi phí tối thiểu khi đi từ điểm thưởng thứ idx đến điểm kết thúc và đang ở điểm thưởng thứ idx , đã đi qua các điểm thưởng x_0, x_1, \dots với x_i là các bit đã được bật và cuối cùng $isFinish$ là 1 biến boolean để kiểm tra xem đã kết thúc việc thăm các điểm thưởng chưa hay vẫn sẽ đi tiếp.
 - $0 \leq idx < N$ là điểm hiện tại đang xét.
 - $state$ là một số nguyên biểu diễn chuỗi nhị phân N bit, trong đó nếu $bit_i = 1$ thì nghĩa là đã đi qua điểm thưởng thứ i với $0 \leq i < N$ và ngược lại thì chưa đi qua. Nhận xét rằng $0 \leq state < 2^N$.
- Quy hoạch động bằng hàm đệ quy $findMinCost(idx, state, isFinish)$ như sau:
 - Nếu $isFinish = True$ thì trả về giá trị 0 là chi phí để kết thúc.
 - Nếu $state = 2^N - 1$, tức là đã đi qua toàn bộ N điểm thưởng thì chỉ còn cần đi đến EXIT nên ta sẽ trả về giá trị $findMinCost(idx, state, True)$ cộng với chi phí đi từ điểm thưởng thứ idx đến EXIT.
 - Nếu một câu hình nào đó đã được tính toán trước đó thì ta trả về giá trị của câu hình đó thông qua mảng dp chứ không cần đệ quy lại.
 - Nếu ta quyết định không đi thăm các điểm thưởng nữa mà đi thẳng đến EXIT mặc dù chưa đi hết các điểm thưởng thì trả về giá trị $findMinCost(idx, state, True)$ cộng với chi phí để đi từ điểm thưởng idx đến EXIT.
 - Với mỗi cặp $(idx, state)$ thì ta có thể tìm được những điểm thưởng nào chưa đi qua.
 - Nếu chọn được điểm thưởng thứ X chưa đi qua thì ta làm thủ tục đi tới X bằng cách chuyển trạng thái idx trở thành điểm X và bật bit_X cho $state$ và gọi $findMinCost(X, new_state, isFinish)$. Chi phí đi từ idx đến X là giá trị điểm thưởng tại X cộng khoảng cách từ idx đến X đã tính sẵn ở trên.
 - Lưu lại giá trị vào $dp[idx][state][isFinish]$.
- Truy vết dựa trên mảng dp để tìm thứ tự đi qua các điểm thưởng nếu muốn có chi phí tối ưu nhất.

Độ phức tạp về thời gian: $O(N * 2^N)$.

Độ phức tạp về không gian: $O(N * 2^N)$.

Trong đó, N là số lượng điểm thưởng trên bản đồ.

2.6.3 Thuật toán A* áp dụng cho Level 2 (AStart_Lv2):

Ý tưởng:

Ý tưởng chính của thuật toán: trong tất cả các bước đi có thể đi tới, chọn bước đi có F nhỏ nhất:

$$F = \min\{f(x, End), f(x, p) | \forall p \in \mathcal{P} \wedge p \notin \mathcal{V}\},$$

trong đó $f(x, y) = g(x) + h(x, y)$ là tổng chi phí đi đến trạng thái x và Heuristic từ x tới y , \mathcal{P} là tập hợp các điểm thưởng (Bonus Point), \mathcal{V} là tập hợp các điểm đã đi qua (Visited). Hàm đánh giá F

nêu trên sẽ hướng ta đi đến điểm Bonus Point hoặc điểm End gần nhất so với trạng thái hiện tại.

Cách cài đặt:

- **Hàm đánh giá:** sử dụng hàm F nêu trên và hai hàm Heuristic nêu trong phần GBFS.

- **Hàm AStar_Lv2:**

- **Bước 1:** Khởi tạo Start, End, Visited để theo dõi các trạng thái đã được duyệt.
- **Bước 2:** Ứng với điểm End và mỗi Bonus Point, khởi tạo một hàng đợi ưu tiên (PriorityQueue), mỗi hàng đợi sắp xếp các nút theo đánh giá $F(x, y)$, với x là nút có thể đi đến, y là Bonus Point hoặc End tương ứng của mỗi hàng đợi. Thêm trạng thái hiện tại ban đầu (Start) vào tất cả các hàng đợi.
- **Bước 3:** Bắt đầu vòng lặp chính cho đến khi tìm thấy đường đi hoặc không còn đường đi nào khả thi. Trong vòng lặp thực hiện:
 1. Kiểm tra nếu hàng đợi ưu tiên ứng với End, nếu rỗng, thông báo "Không có đường đi đến đích" và dừng chương trình.
 2. Lấy trạng thái có hàm đánh giá nhỏ nhất trong tất cả các hàng đợi ưu tiên ra làm trạng thái hiện tại (Current).
 3. Kiểm tra xem Current có thuộc Visited hay không, nếu có thì quay lại đầu vòng lặp.
 4. Kiểm tra xem Current có bằng End không. Nếu có, tìm và trả về đường đi từ Start đến End bằng phương pháp truy vết. Chí phí của đường đi bằng chí phí của nút cuối cùng được thêm vào Visited.
 5. Kiểm tra Current có bằng một trong các Bonus Point hay không. Nếu có, giảm chí phí của Current đi bằng số điểm thưởng của Bonus Point tương ứng, cập nhật lại giá trị F của Current và xóa hàng đợi ưu tiên tương ứng với Bonus Point đó.
 6. Thêm Current vào danh sách Visited.
 7. Tìm các hàng xóm (Neighbor) của trạng thái hiện tại (các trạng thái có thể đi tới được từ Current và chưa nằm trong Visited). Thêm các Neighbors vào tất cả các hàng đợi ưu tiên, sắp xếp chúng theo giá trị của hàm đánh giá F . Quay lại đầu vòng lặp.

2.7 Các thuật toán cho bản đồ có điểm đón

Trong bản đồ Level 3, ngoài điểm xuất phát và điểm đích, ta còn có các điểm thưởng "đặc biệt" gọi là điểm đón: điểm thưởng của các điểm đón này đều sẽ bằng 0 và chúng ta buộc phải tìm đường đi qua hết tất cả các điểm đón trước khi tới đích đến. Mục tiêu của ta là tìm đường đi đến đích sao cho chi phí đường đi nhỏ nhất mà vẫn thỏa các điều kiện trên.

2.7.1 Thuật giải di truyền:

Ý tưởng:

Đặt vấn đề rằng liệu ta có thể đi tìm một thuật toán tối ưu như cách làm với bản đồ có điểm thưởng đã nói ở trên hay không thì không thể. Ở bản đồ có điểm đón theo yêu cầu của đồ án này

số lượng điểm đón tối đa có thể lên tới 25 điểm. Nếu tiếp tục dùng các bit trạng thái để biểu diễn trạng thái thì cần độ phức tạp ít nhất là $O(25 * 2^{25})$, quá lớn cho cả về thời gian lẫn bộ nhớ.

Vậy nếu không thể tìm thuật toán tối ưu thì chúng ta sẽ tìm hướng những hướng đi mới. Và thuật toán đầu tiên để giải quyết bản đồ có điểm đón chính là thuật giải di truyền.

Ý tưởng đầu tiên để áp dụng thuật giải di truyền đó là ta cảm thấy yêu cầu của bản đồ này có vẻ giống với một bài toán nổi tiếng nào đó. Có vẻ là bài toán TSP (Travelling Salesman Problem). Bài toán người bán hàng là bài toán cho trước danh sách các thành phố và khoảng cách giữa chúng, tìm chu trình đi qua tất cả thành phố sao cho chi phí khoảng cách là ngắn nhất. Nhưng đây lại là tìm đường đi trên một ma trận 2 chiều và ngoài các điểm đón còn có những điểm bình thường khác. Tuy nhiên ta hoàn toàn có thể đưa bài toán này về bài toán TSP bằng cách đi tìm chi phí đường đi giữa các điểm đón trước, khi đó ta coi các điểm đón như một đỉnh trên đồ thị bình thường và cần đi qua mọi đỉnh trước khi tới EXIT.

Sau khi trở thành bài toán TSP thì ta áp dụng giải thuật di truyền như mong muốn. Giải thuật di truyền gồm 4 bước chính: Mã hóa lời giải, khởi tạo quần thể, sử dụng các phép toán di truyền như sinh sản, chọn lọc tự nhiên hay hiếm hơn là đột biến và cuối cùng là đánh giá độ thích nghi.

Cách cài đặt:

- Đầu tiên khởi tạo một class đại diện cho cá thể trong thuật toán, gồm các phương thức lai tạo, đột biến, tính độ thích nghi (ở đây độ thích nghi là chi phí hoàn thành đường đi, chi phí càng nhỏ thì càng tốt).
- Gọi N là số lượng điểm đón ($0 \leq N \leq 25$).
- Tính trước khoảng cách chi phí và đường đi giữa mỗi cặp điểm đón với nhau, cũng như chi phí và đường đi đi từ điểm bắt đầu đến các điểm đón và từ các điểm đón đến Exit bằng cách dùng thuật toán A* từ một điểm đến điểm kia. Ở đây ta chọn A* là vì ta đang giải quyết một bài toán không cần quá khắt khe về độ tối ưu nên nếu dùng A* thì sẽ tiết kiệm được thời gian chạy hơn rất nhiều.
- Mã hóa chu trình (cá thể - gen): Chu trình được mã hóa thành mảng có thứ tự các số hiệu index của các điểm đón. Ví dụ của 1 chu trình 10 điểm đón: [8, 9, 0, 1, 2, 7, 5, 6, 3, 2]. Mỗi chu trình cần phải có thêm thông số về chi phí của toàn bộ chu trình đó. Chi phí này được tính bằng tổng độ dài các cạnh trong chu trình và cộng thêm độ dài đường đi từ điểm bắt đầu đến đỉnh đầu tiên và từ đỉnh cuối cùng đến điểm EXIT thông qua các khoảng cách đã tính trước ở trên. Mỗi chu trình sẽ là một lời giải, trong giải thuật di truyền coi đó như 1 cá thể. Việc tiến hóa về sau sẽ dựa trên tập chu trình khởi tạo ban đầu và tìm ra kết quả tốt nhất sau một số thế hệ.
- Khởi tạo quần thể: Quần thể ban đầu được khởi tạo bằng cách sinh ngẫu nhiên các chu trình. Số lượng chu trình khởi tạo cũng như số lượng cá thể tối đa được em chọn là 100 cá thể. Việc sinh ngẫu nhiên sẽ sử dụng hàm đột biến (giải thích bên dưới).
- Tạo phương thức đột biến: Phương thức đột biến được thực hiện dựa trên 1 cá thể đầu vào. Ta thực hiện đột biến bằng cách tráo đổi các điểm trên gen cho nhau. Số lần tráo đổi được sinh ngẫu nhiên từ trong khoảng 5% chiều dài chu trình (tức là có tối đa 10% vị trí trên 1 gen có thể bị đột biến), vị trí được tráo đổi cũng sẽ được sinh ngẫu nhiên trong quá trình chạy.

- Tạo phương thức lai ghép: Phương thức lai ghép được thực hiện dựa trên 2 cá thể đầu vào. Thực hiện lai ghép 1 điểm cắt với vị trí cắt là ngẫu nhiên. Sau đó lấy từ vị trí cắt đến hết chu trình đầu tiên và nạp dần theo thứ tự những điểm đón còn sót lại trong chu trình thứ hai và có thể làm ngược lại. Ngoài ra sẽ có tỉ lệ 10% đột biến dựa trên hai cá thể đầu vào.
- Chọn lọc tự nhiên và tiến hóa: Qua nhiều thế hệ, số cá thể trong quần thể vẫn luôn là 100 cố định. Ở mỗi thế hệ, ta giữ lại 10% cá thể tốt nhất cho tới thế hệ tiếp theo, 90% cá thể còn lại sẽ có được quyền phép lai ghép từ 50 cá thể tốt nhất trong mỗi thế hệ. Ngoài ra cứ mỗi 100 thế hệ thì sẽ tiến hành đột biến trên toàn bộ cá thể để làm đa dạng gen.
- Sau khi trải qua 1000 thế hệ thì ta lấy ra cá thể có chi phí tối ưu nhất để làm chu trình cho bài toán.

Lưu ý: Mặc định là input của bản đồ mê cung phải luôn có đường đi giữa các điểm đón.

2.7.2 Thuật toán leo đồi:

Ý tưởng:

Một cách giải quyết bản đồ có điểm đón gần giống với thuật giải di truyền là sử dụng thuật toán leo đồi (Hill Climbing). Chúng ta cũng sẽ cần đưa bài toán về dạng bài toán người đưa hàng. Sau đó áp dụng thuật toán leo đồi để lấy chu trình tối ưu nhất tìm được.

Cách cài đặt:

- Tính trước khoảng cách chi phí và đường đi giữa mỗi cặp điểm đón với nhau, cũng như chi phí và đường đi đi từ điểm bắt đầu đến các điểm đón và từ các điểm đón đến Exit bằng cách dùng thuật toán A* từ một điểm đến điểm kia. Ở đây ta chọn A* là vì ta đang giải quyết một bài toán không cần quá khắt khe về độ tối ưu nên nếu dùng A* thì sẽ tiết kiệm được thời gian chạy hơn rất nhiều.
- Mã hóa chu trình: Chu trình được mã hóa thành mảng có thứ tự các số hiệu index của các điểm đón. Ví dụ của 1 chu trình 10 điểm đón: [8, 7, 0, 1, 2, 9, 5, 6, 3, 2].
- Sinh ngẫu nhiên một chu trình đầu tiên, sau đó sinh ra các chu trình "hàng xóm" của chu trình đó.
- Các chu trình "hàng xóm" được tạo ra dựa trên 1 chu trình đầu vào, bằng cách lần lượt hoán đổi giá trị thứ tự các cặp đỉnh trong chu trình. Tìm ra chu trình hàng xóm tạo ra đường đi tốt nhất và so sánh với chu trình ban đầu, nếu chu trình hàng xóm tốt hơn thì từ chu trình đó sẽ tiếp tục được sinh ra những chu trình hàng xóm mới. Ngược lại nếu chu trình ban đầu tốt hơn mọi chu trình hàng xóm thì sẽ lấy kết quả cho bài toán chính là chu trình ban đầu và kết thúc thuật toán.

Lưu ý: Mặc định là input của bản đồ mê cung phải luôn có đường đi giữa các điểm đón.

2.8 Thuật toán cho bản đồ nâng cao - Teleport

Ý tưởng:

Ở bản đồ nâng cao này, nhóm sẽ làm một bản đồ dịch chuyển 1 chiều, và từ cổng dịch chuyển đầu vào sẽ bắt buộc phải đi đến cổng dịch chuyển đầu ra chứ không được tùy ý đi 4 hướng lân cận như bình thường.

Ta sẽ sử dụng thuật toán tìm kiếm BFS có cách cài đặt như phần 2.2 nhưng có một thay đổi nhỏ. Đó là khi duyệt tới 1 cổng dịch chuyển đầu vào, thay vì push điểm dịch chuyển đó vào hàng đợi thì ta cần phải push điểm dịch chuyển đầu ra vào hàng đợi. Mục đích là để từ cổng đầu vào không thể tiếp tục di chuyển theo 4 hướng lân cận như bình thường mà chỉ có thể đi tới cổng đầu ra.

Sử dụng thuật toán BFS cho ra kết quả tối ưu bởi vì ở bản đồ này bản chất là 4 điểm lân cận với cổng đầu vào sẽ có thêm 1 đường đi trực tiếp có trọng số bằng 1 tới cổng đầu ra. Như vậy vẫn giữ được thứ tự duyệt từ bé đến lớn như bình thường nên cách làm này sẽ cho ra kết quả tối ưu nếu tồn tại đường đi.

Độ phức tạp về thời gian: $O(\text{height} * \text{width})$.

Độ phức tạp về không gian: $O(\text{height} * \text{width})$.

Trong đó, height và width lần lượt chính là chiều cao và chiều rộng của mê cung.

3 Các kịch bản và kết quả chạy

Trong các bản đồ có các thành phần sau:



Tường của mê cung: thể hiện các bức tường và các chướng ngại vật, tác nhân sẽ không thể di chuyển lên các vị trí này



Điểm bắt đầu: đại diện cho vị trí xuất phát của tác nhân.



Điểm kết thúc: đại diện cho vị trí đích mà tác nhân cần di chuyển đến, là vị trí duy nhất trên biên của các bức tường mà tác nhân có thể thoát khỏi mê cung.



Một bước đi trong đường đi tìm được dẫn đến đích



Những vị trí đã mở ra



Một bước đi lặp lại vị trí đã đi trước đó trong đường đi tới đích



Điểm thưởng chưa được đi qua, điểm đón



Điểm thưởng, điểm đón đã đi qua



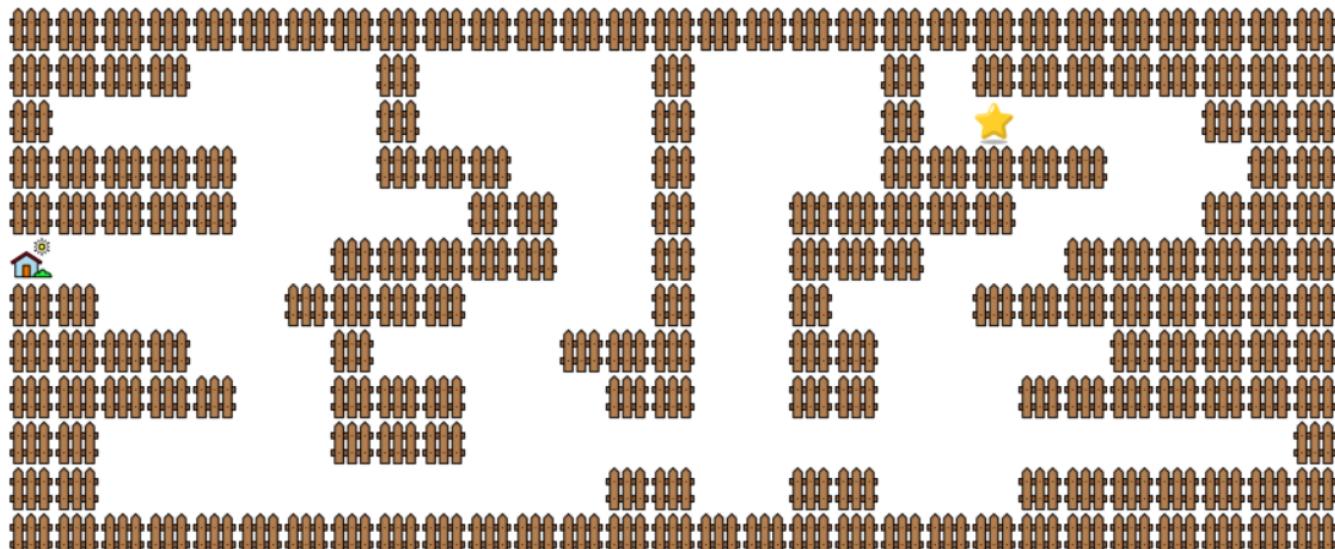
Cổng dịch chuyển đầu vào



Cổng dịch chuyển đầu ra

3.1 Level 1: Bản đồ không có điểm thưởng

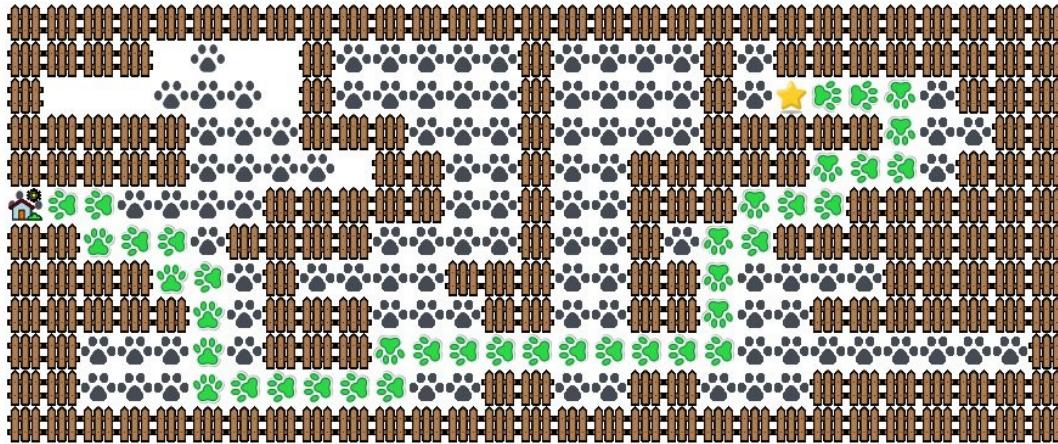
3.1.1 Kịch bản 1 - Bản đồ không có điểm thưởng



Hình 1: Kịch bản 1, Level 1

Kết quả chạy trên các thuật toán BFS, DFS, UCS, GBFS, A*:

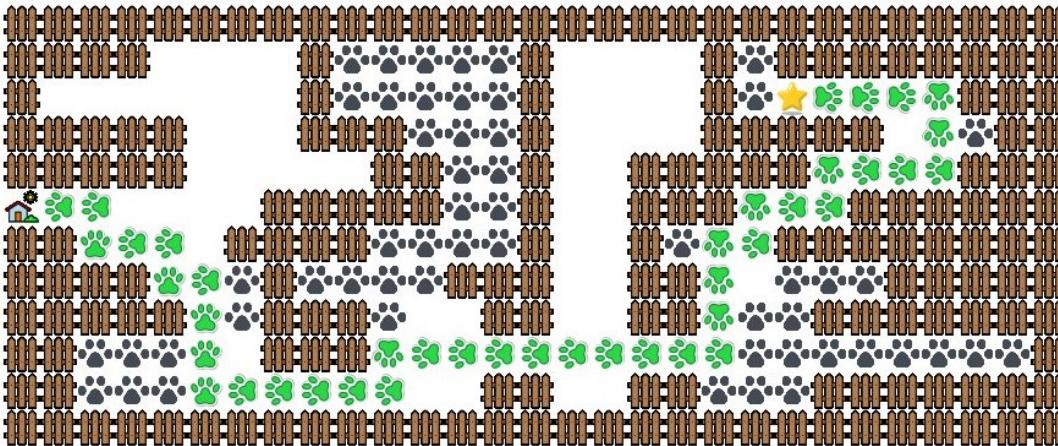
1. BFS



Hình 2: Kết quả chạy BFS trên kịch bản 1, Level 1

- Chi phí đường đi tìm được: 40
- Thời gian chạy: 0.001105 (s).
- Tổng số nút mở: 148

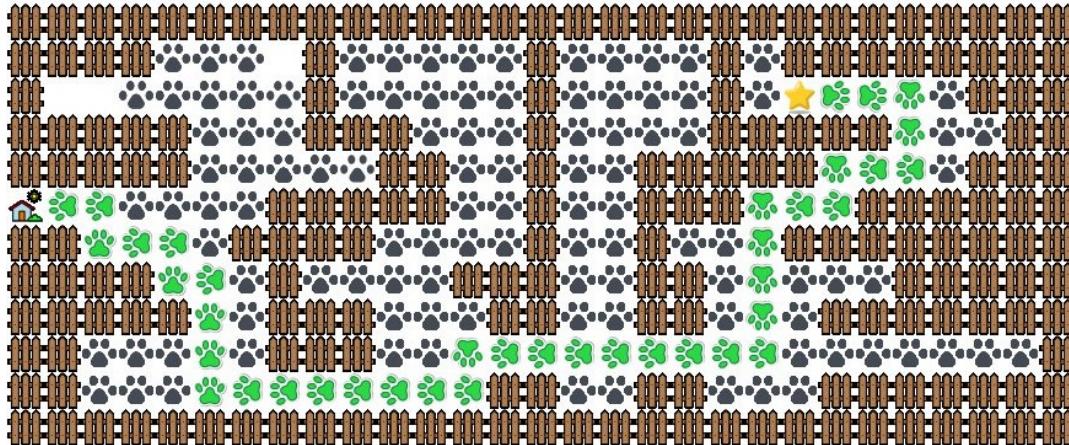
2. DFS



Hình 3: Kết quả chạy DFS trên kịch bản 1, Level 1

- Chi phí đường đi tìm được: 42
- Thời gian chạy: 0.000919(s)
- Tổng số nút mở: 95

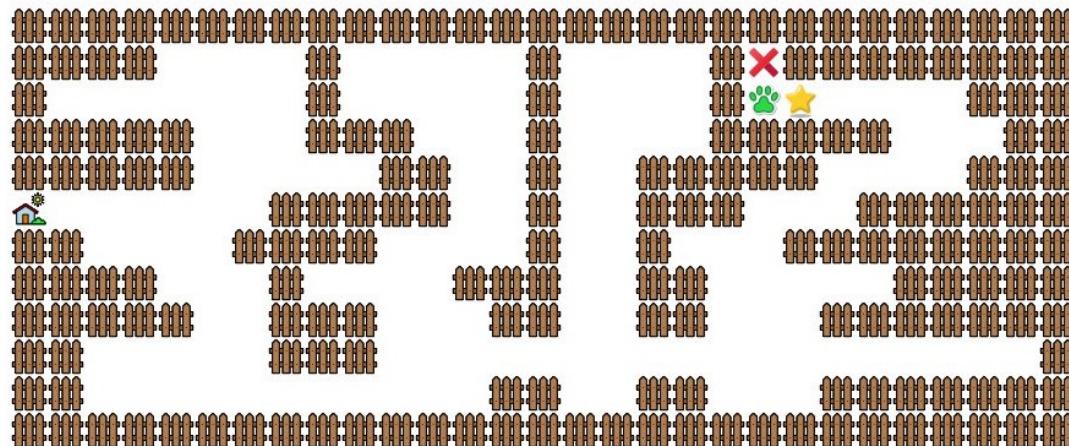
3. UCS



Hình 4: Kết quả chạy UCS trên kịch bản 1, Level 1

- Chi phí đường đi tìm được: 40
- Thời gian chạy: 0.002025 (s).
- Tổng số nút mở: 147

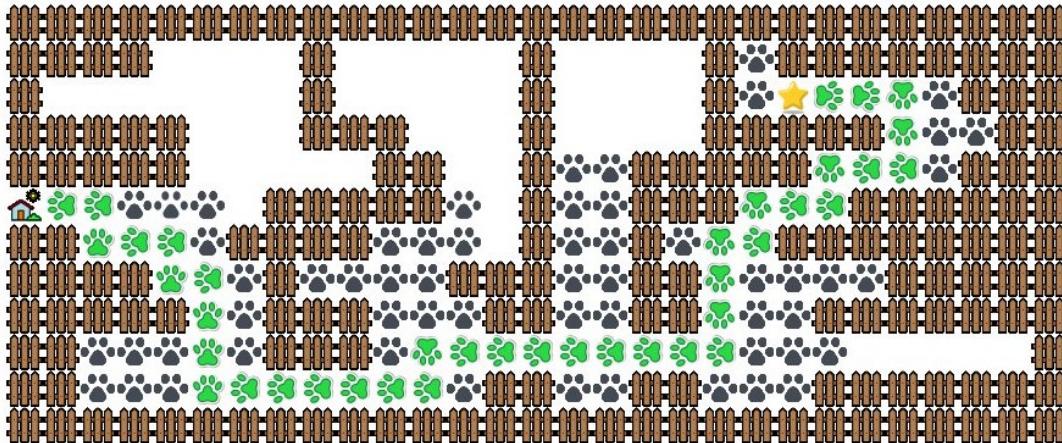
4. GBFS



Hình 5: Kết quả chạy GBFS trên kịch bản 1, Level 1

- Không tìm được đường đi tới đích.

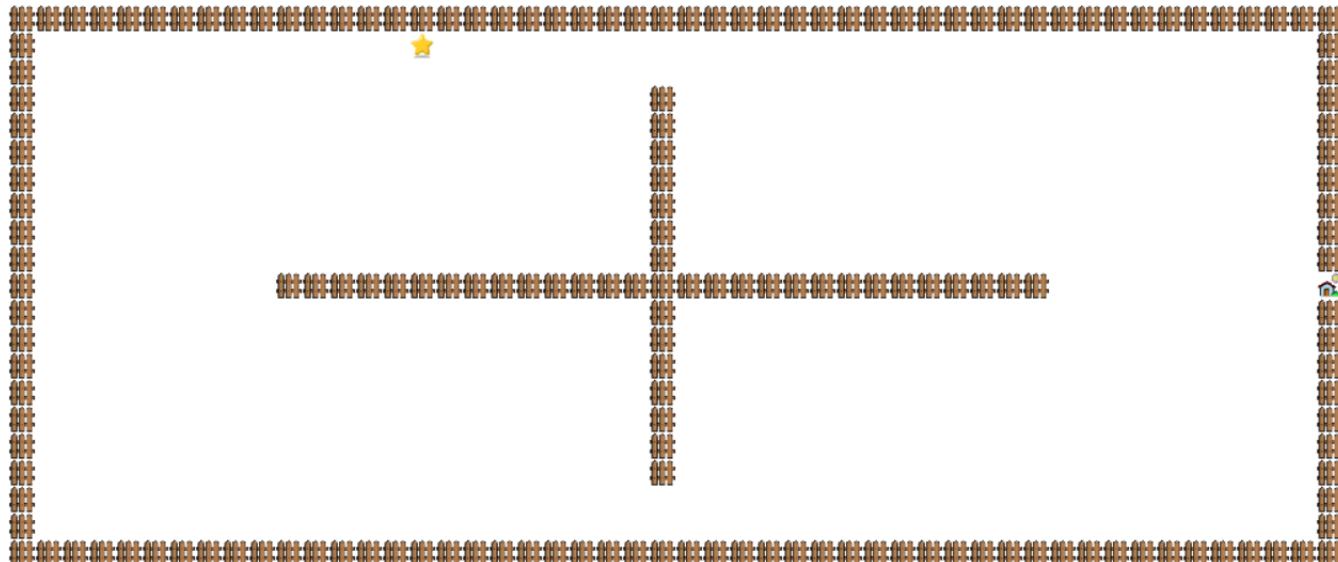
5. A*



Hình 6: Kết quả chạy A* trên kinh bản 1, Level 1

- Chi phí đường đi tìm được: 40
- Thời gian chạy: 0.001 (s)
- Tổng số nút mở: 96.

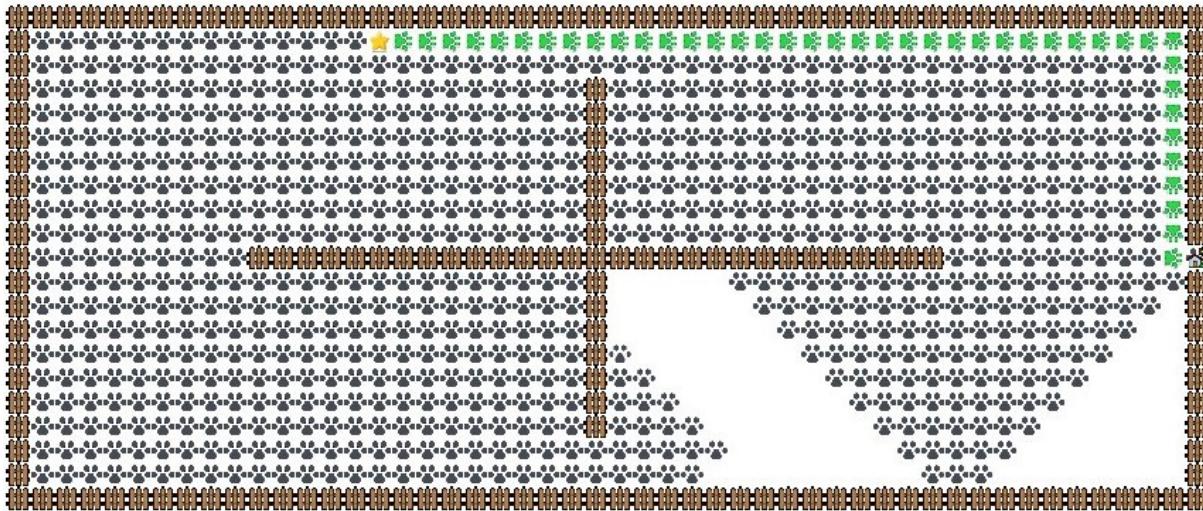
3.1.2 Kịch bản 2 - Bản đồ không có điểm thưởng



Hình 7: Kịch bản 2, Level 1

Kết quả chạy trên các thuật toán BFS, DFS, UCS, GBFS, A*:

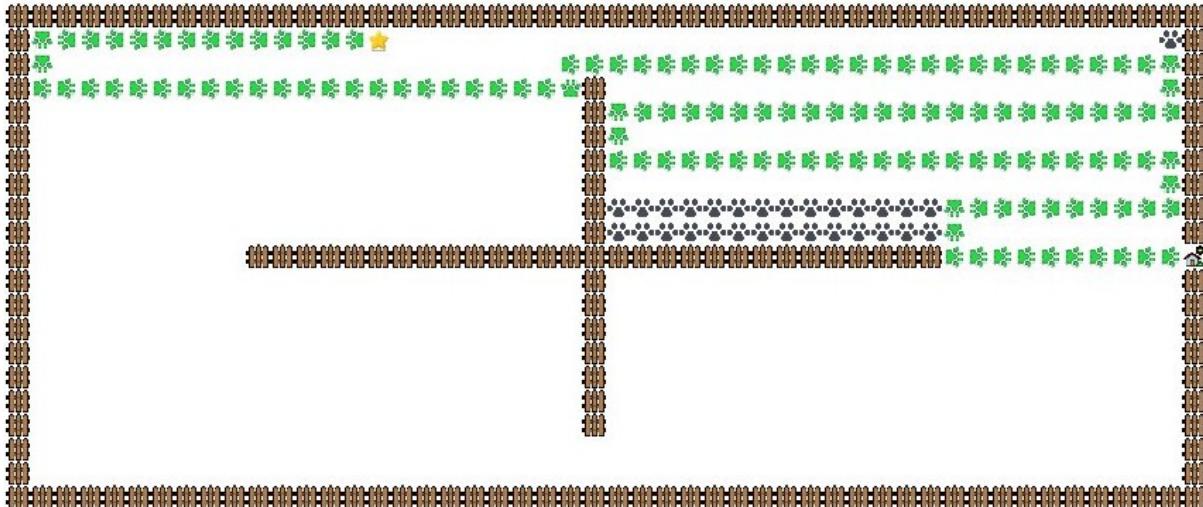
1. BFS



Hình 8: Kết quả chạy BFS trên kinh bản 2, Level 1

- Chi phí đường đi tìm được: 43
- Thời gian chạy: 0.017336 (s).
- Tổng số nút mở: 771

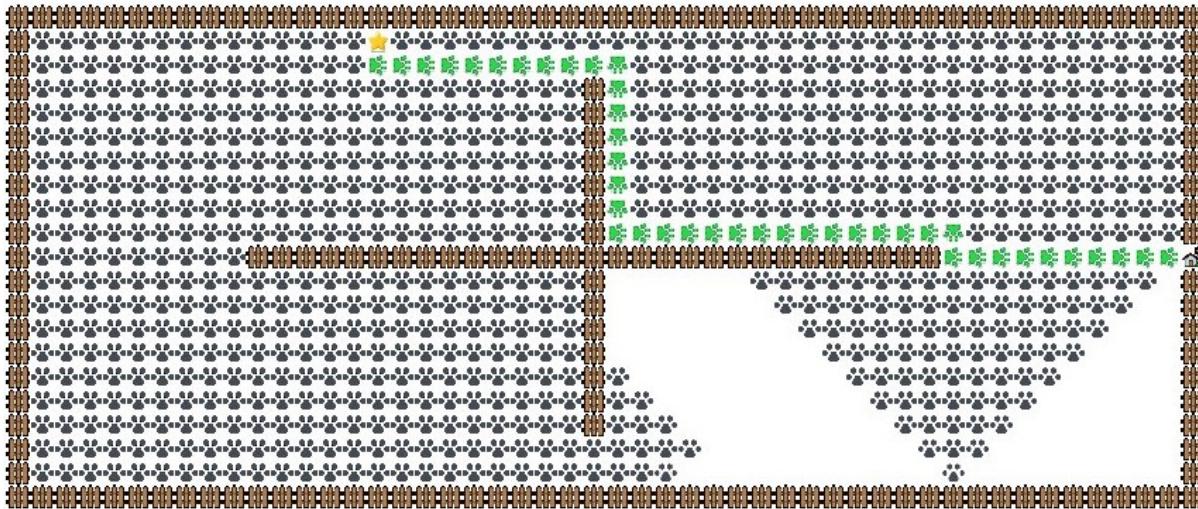
2. DFS



Hình 9: Kết quả chạy DFS trên kinh bản 2, Level 1

- Chi phí đường đi tìm được: 137
- Thời gian chạy: 0.004089(s)
- Tổng số nút mở: 165

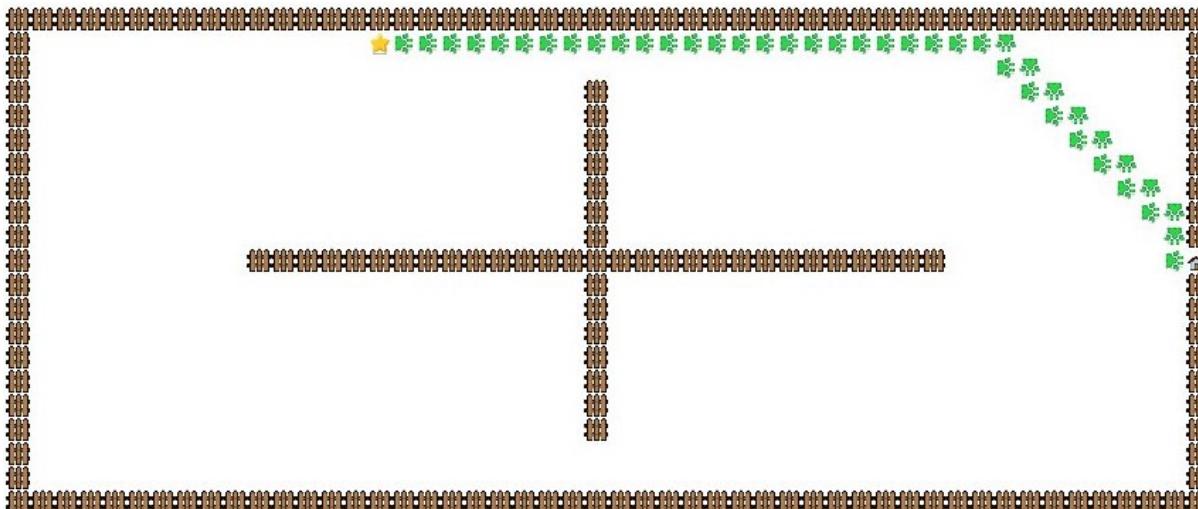
3. UCS



Hình 10: Kết quả chạy UCS trên kịch bản 2, Level 1

- Chi phí đường đi tìm được: 43
- Thời gian chạy: 0.030889 (s).
- Tổng số nút mở: 746

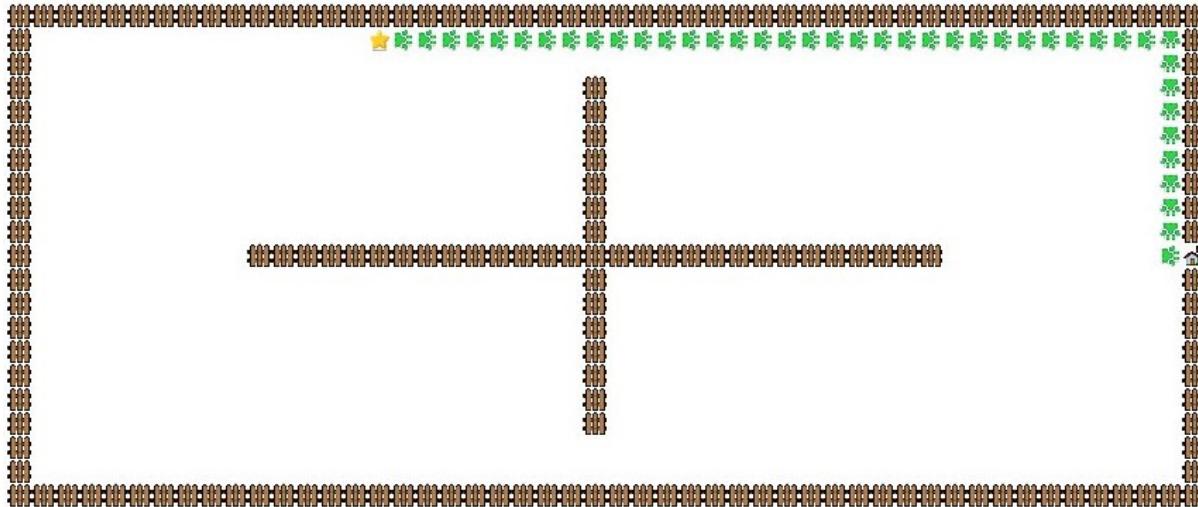
4. GBFS sử dụng Ecludiean distance



Hình 11: Kết quả chạy GBFS sử dụng Ecludiean distance trên kịch bản 2, Level 1

- Chi phí đường đi tìm được: 43
- Thời gian chạy: 0.000996 (s).
- Tổng số nút mở: 42

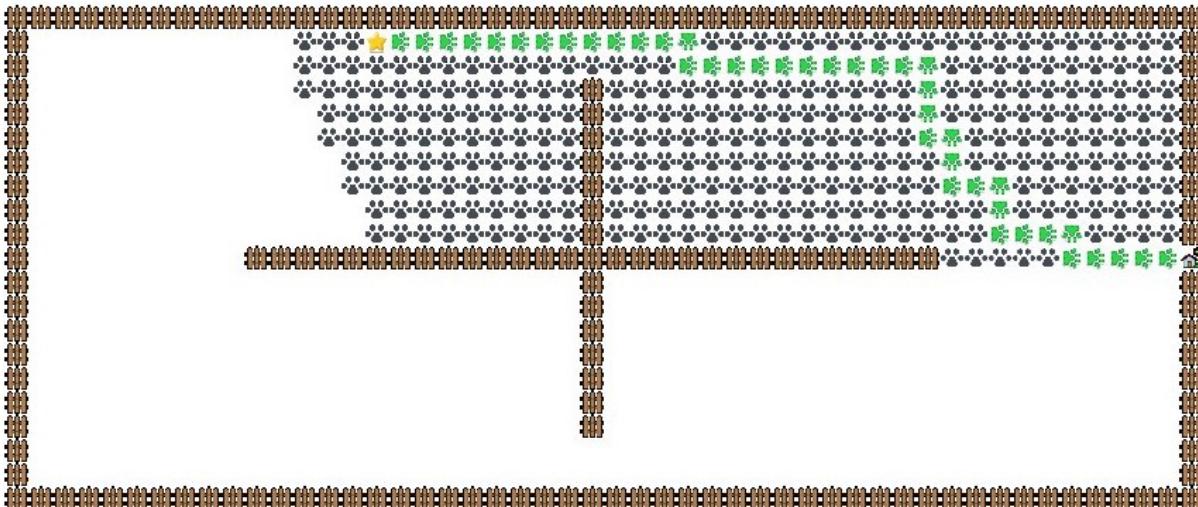
GBFS sử dụng Manhattan Distance



Hình 12: Kết quả chạy GBFS sử dụng Manhattan distance trên kịch bản 2, Level 1

- Chi phí đường đi tìm được: 43
- Thời gian chạy: 0.000999 (s).
- Tổng số nút mở: 42

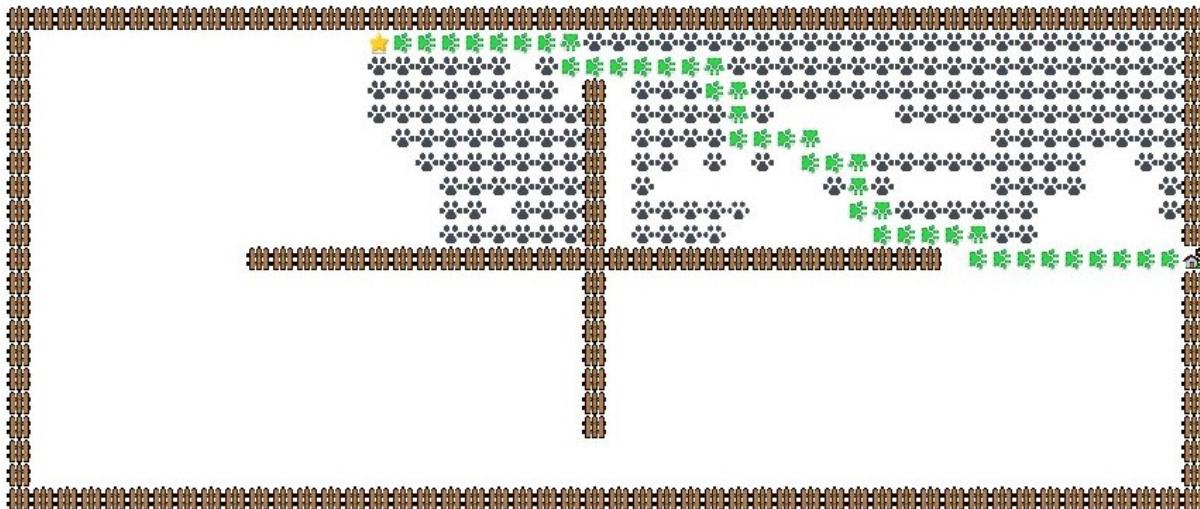
5. A* sử dụng Euclidean distance



Hình 13: Kết quả chạy A* sử dụng Euclidean distance trên kịch bản 2, Level 1

- Chi phí đường đi tìm được: 43
- Thời gian chạy: 0.015058 (s)
- Tổng số nút mở: 317.

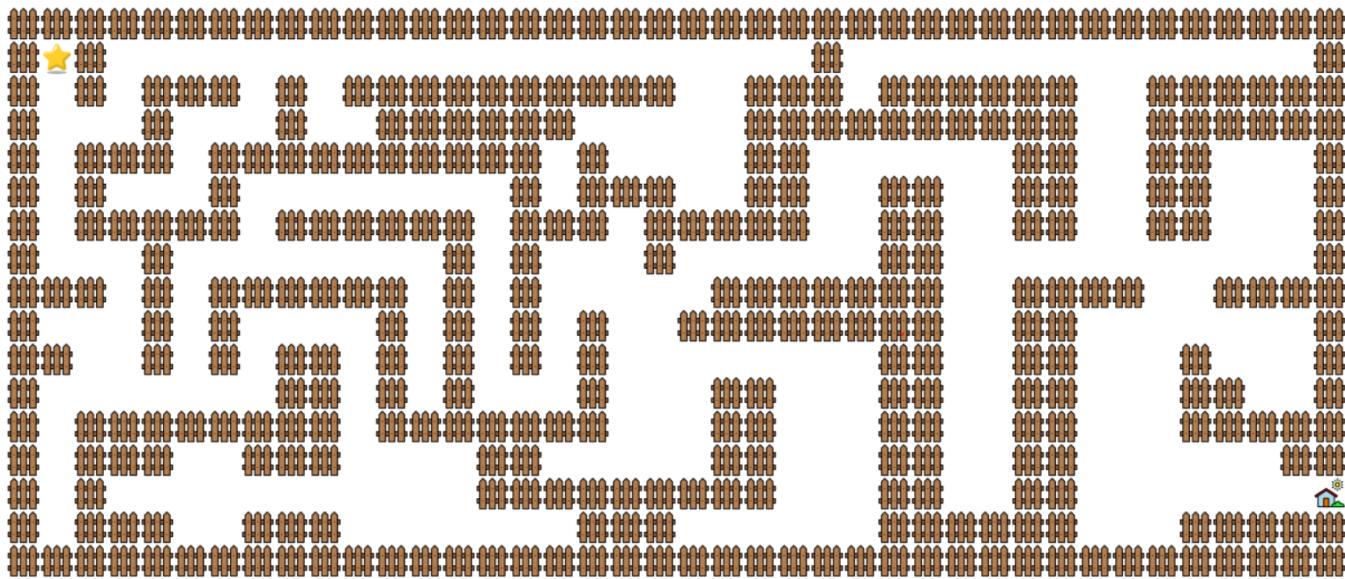
A* sử dụng Manhattan distance



Hình 14: Kết quả chạy A* sử dụng Manhattan distance trên kịch bản 2, Level 1

- Chi phí đường đi tìm được: 43
- Thời gian chạy: 0.008002 (s)
- Tổng số nút mở: 230.

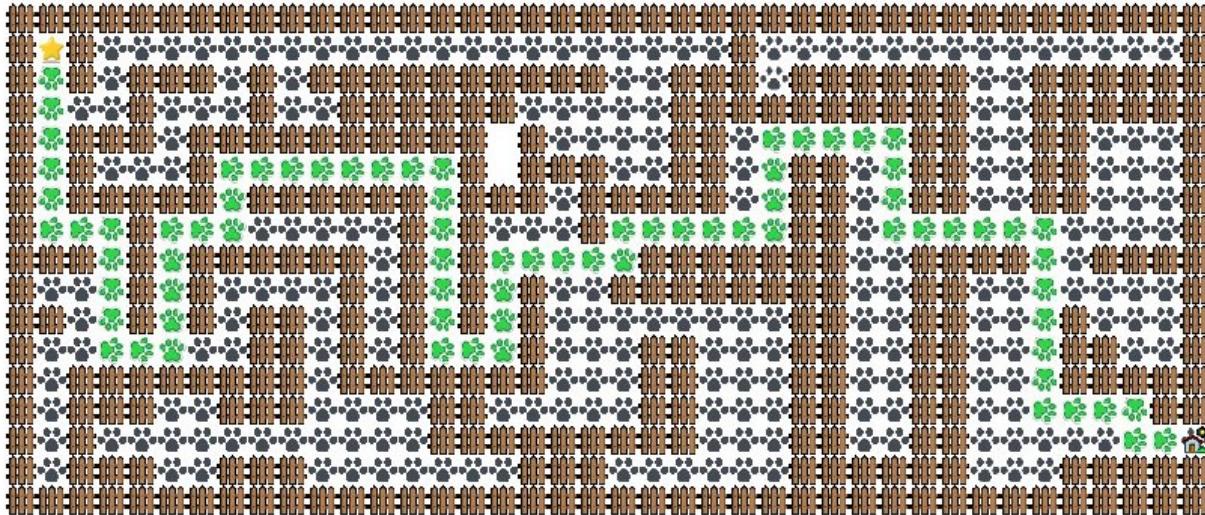
3.1.3 Kịch bản 3 - Bản đồ không có điểm thường



Hình 15: Kịch bản 3, Level 1

Kết quả chạy trên các thuật toán BFS, DFS, UCS, GBFS, A*:

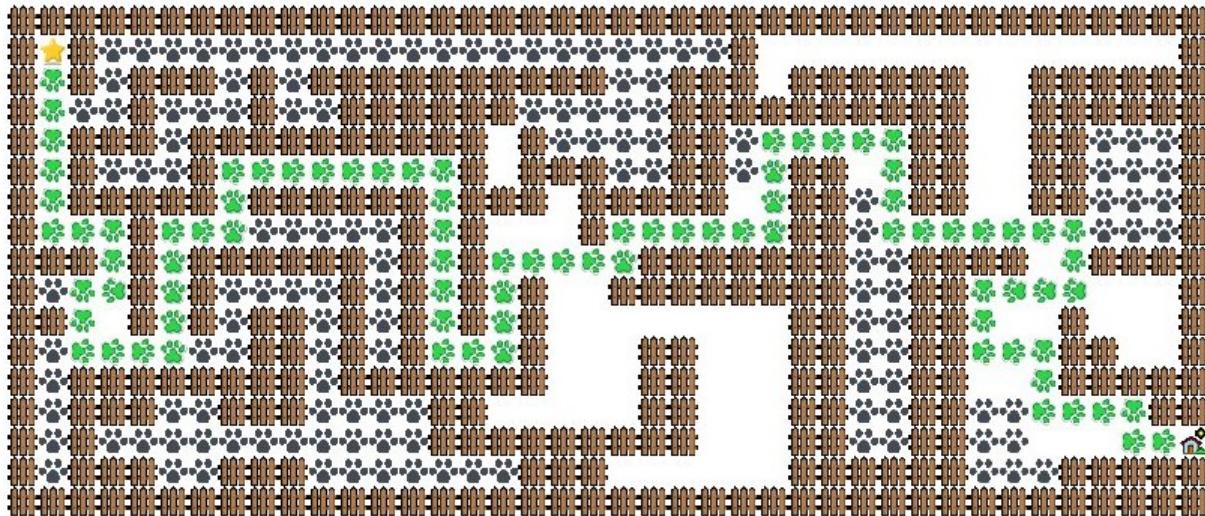
1. BFS



Hình 16: Kết quả chạy BFS trên kinh bản 3, Level 1

- Chi phí đường đi tìm được: 77
- Thời gian chạy: 0.004087 (s).
- Tổng số nút mở: 308

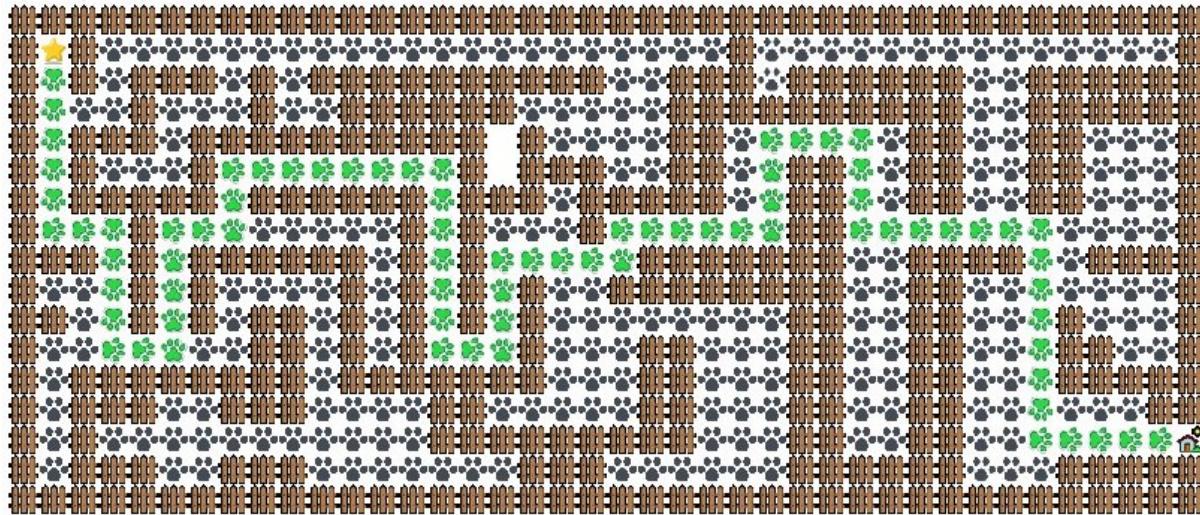
2. DFS



Hình 17: Kết quả chạy DFS trên kinh bản 3, Level 1

- Chi phí đường đi tìm được: 85
- Thời gian chạy: 0.002093(s)
- Tổng số nút mở: 220

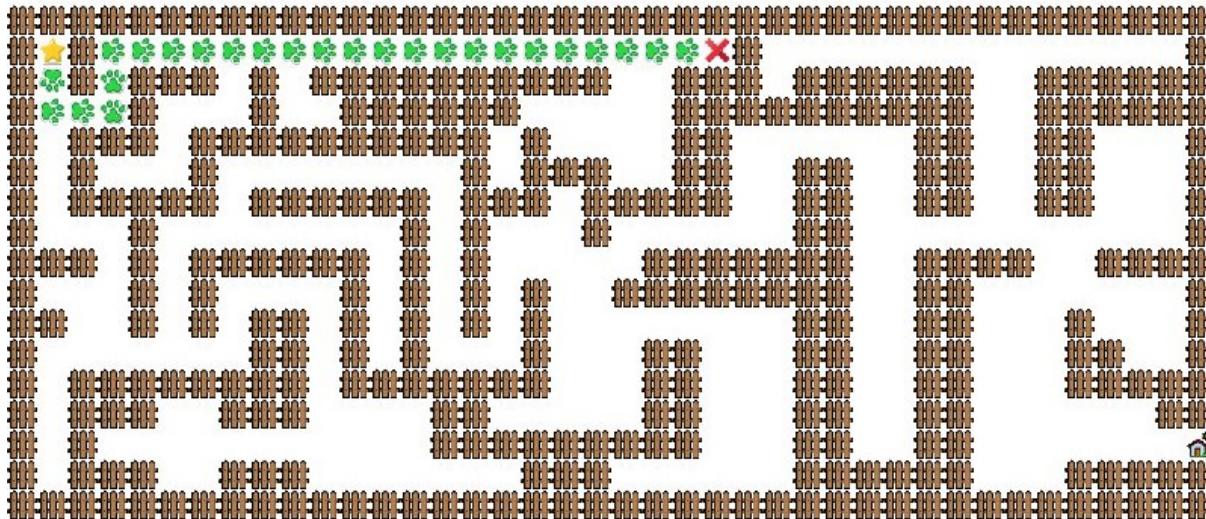
3. UCS



Hình 18: Kết quả chạy UCS trên kịch bản 3, Level 1

- Chi phí đường đi tìm được: 77
- Thời gian chạy: 0.005091 (s).
- Tổng số nút mở: 307

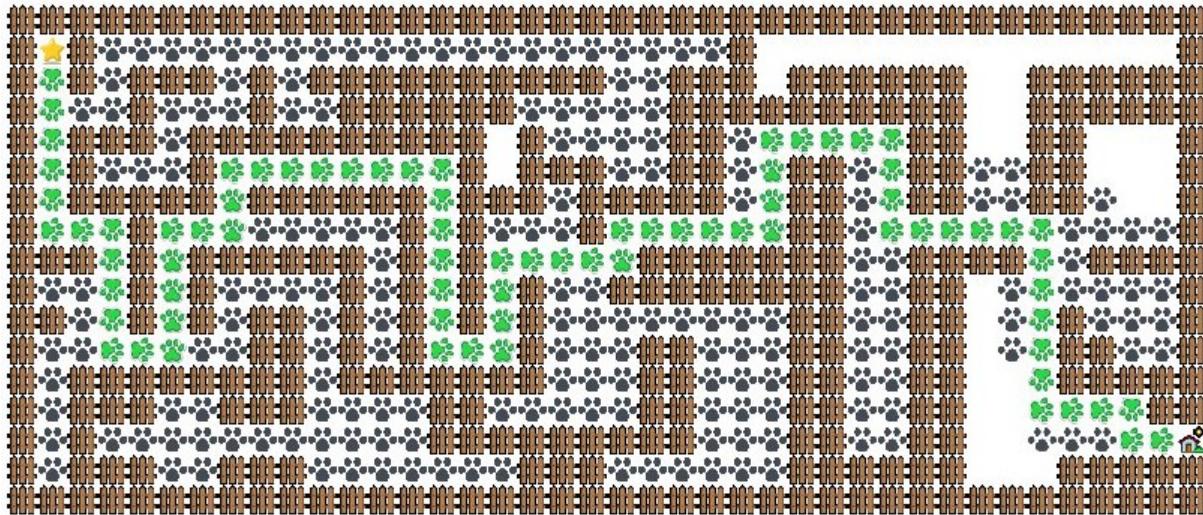
4. GBFS



Hình 19: Kết quả chạy GBFS trên kịch bản 3, Level 1

- Không tìm được đường đi đến đích.

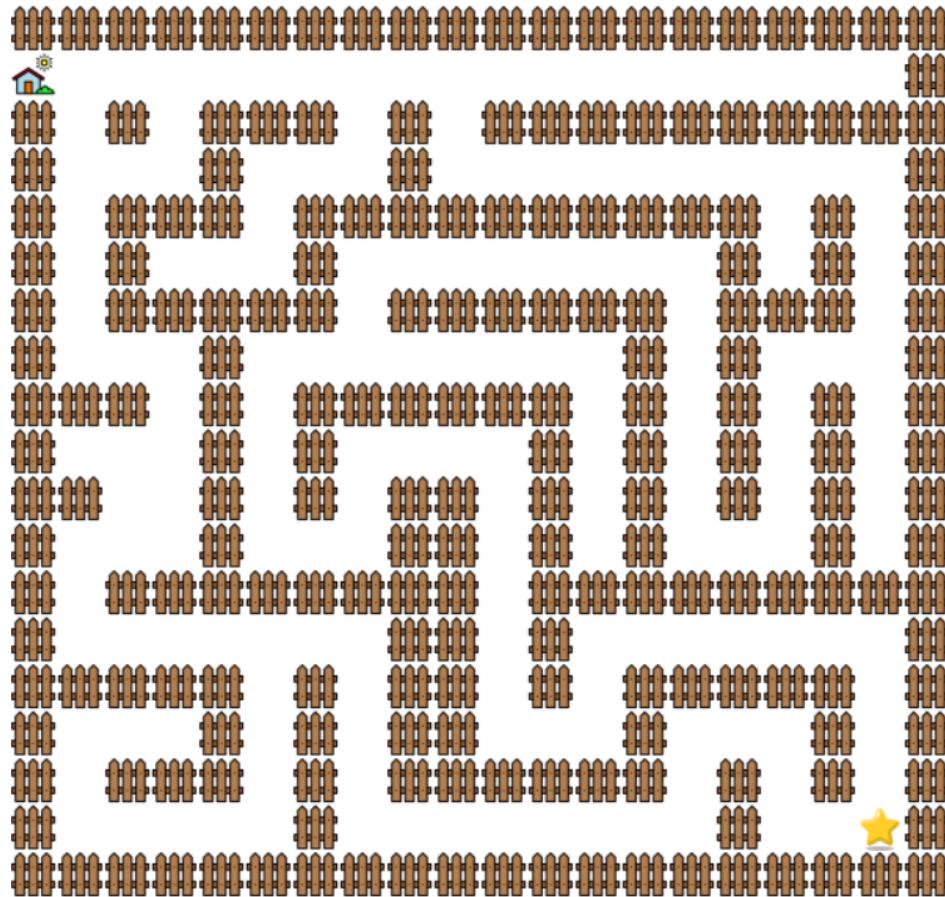
5. A*



Hình 20: Kết quả chạy A* trên kịch bản 3, Level 1

- Chi phí đường đi tìm được: 77
- Thời gian chạy: 0.004097 (s)
- Tổng số nút mở: 266.

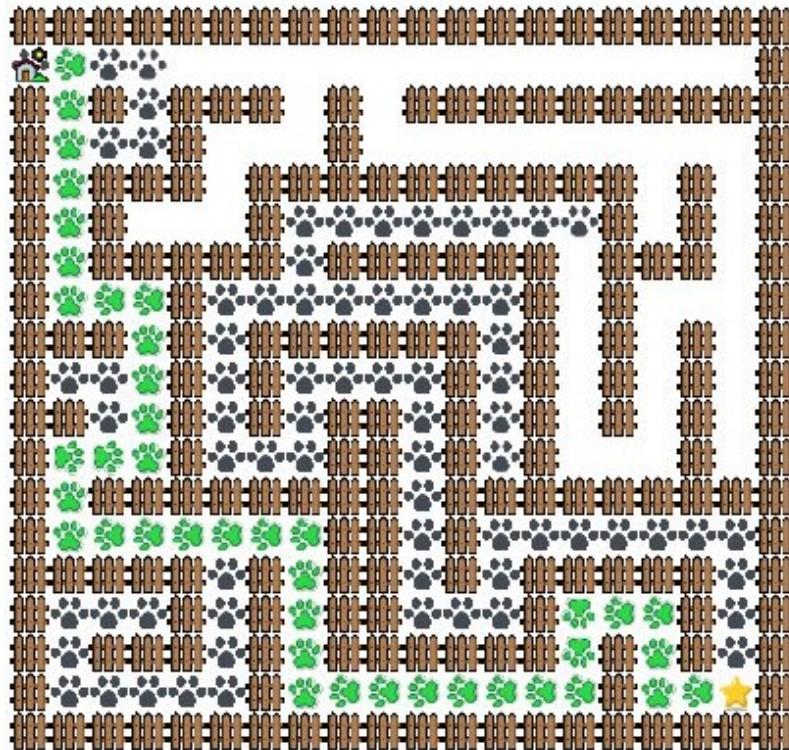
3.1.4 Kịch bản 4 - Bản đồ không có điểm thường



Hình 21: Kịch bản 4, Level 1

Kết quả chạy trên các thuật toán BFS, DFS, UCS, GBFS, A*:

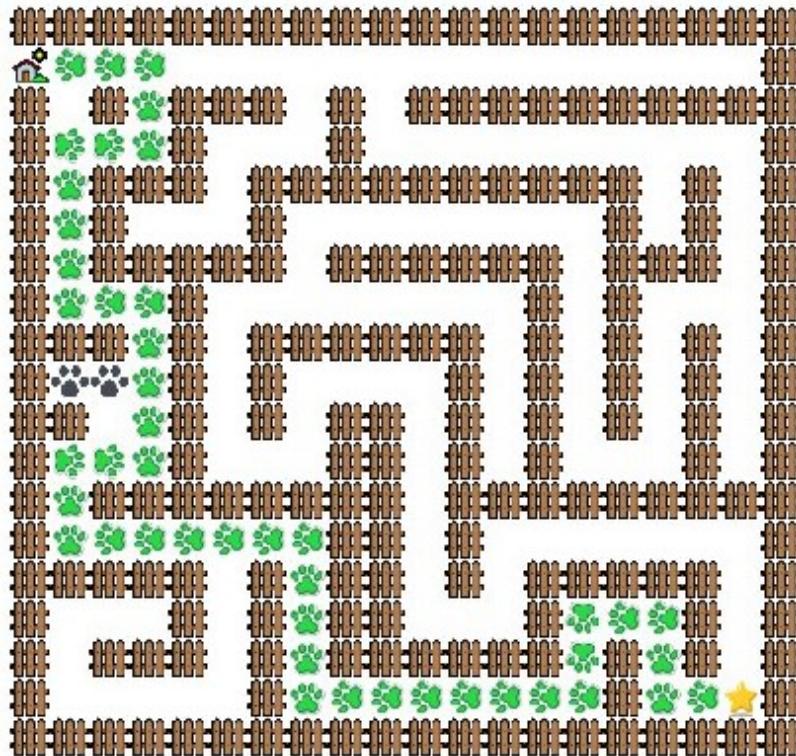
1. BFS



Hình 22: Kết quả chạy BFS trên kích thước 4, Level 1

- Chi phí đường đi tìm được: 42
- Thời gian chạy: 0.0 (s).
- Tổng số nút mở: 113

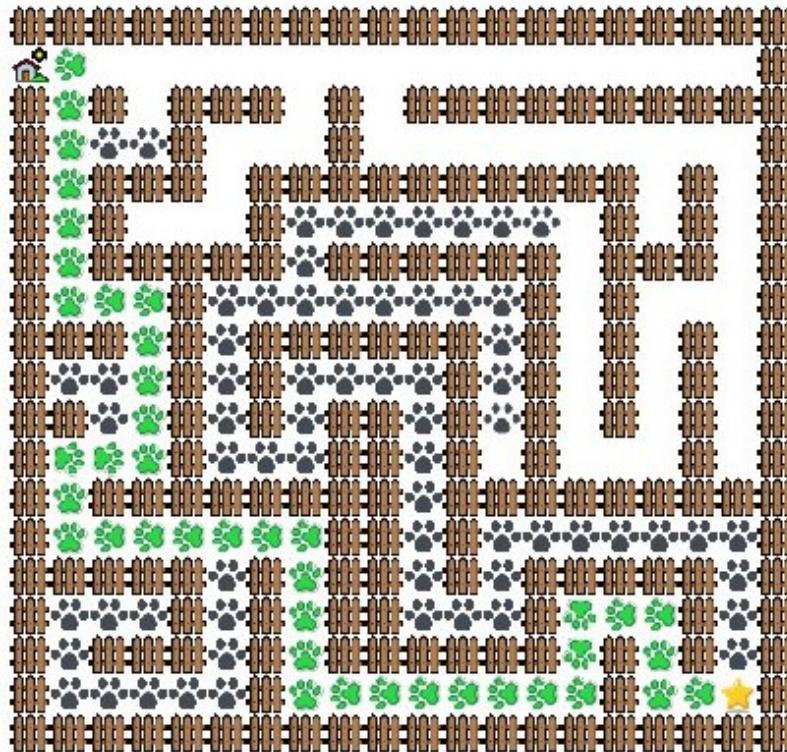
2. DFS



Hình 23: Kết quả chạy DFS trên kinh bắn 4, Level 1

- Chi phí đường đi tìm được: 46
- Thời gian chạy: 0.000900(s)
- Tổng số nút mở: 47

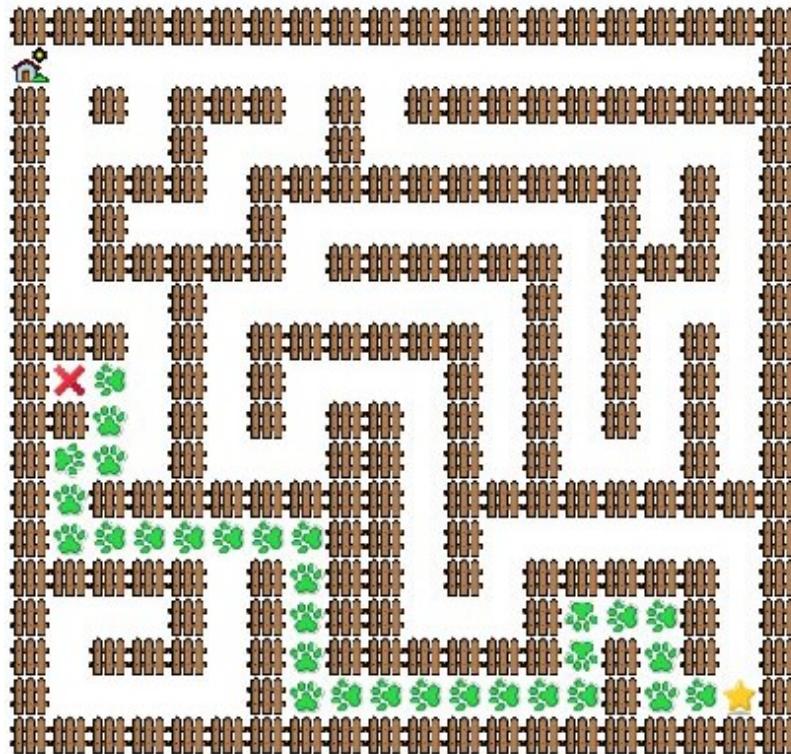
3. UCS



Hình 24: Kết quả chạy UCS trên kịch bản 4, Level 1

- Chi phí đường đi tìm được: 42
- Thời gian chạy: 0.000997 (s).
- Tổng số nút mở: 107

4. GBFS sử dụng Heuristic Euclidean distance

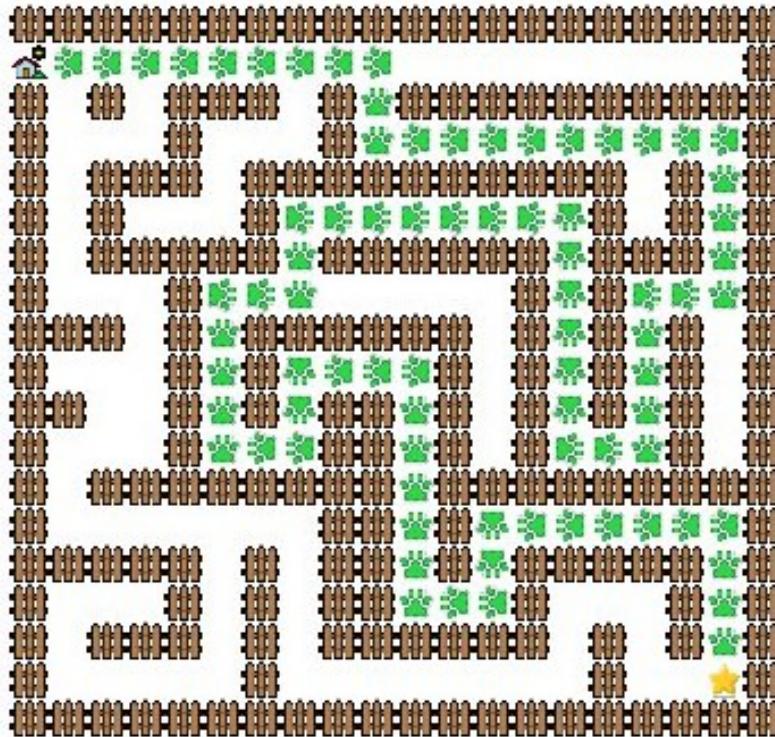


Hình 25: Kết quả chạy GBFS trên kịch bản 4, Level 1

- Không tìm được đường đi đến đích.

GBFS sử dụng Heuristic Manhattan distance

Các phiên bản GBFS ở trên sử dụng Heuristic Euclidean distance. Ở kịch bản này, chúng ta sẽ xem xét thêm một Heuristic nữa vì nó mang lại sự khác biệt so với Heuristic ban đầu.

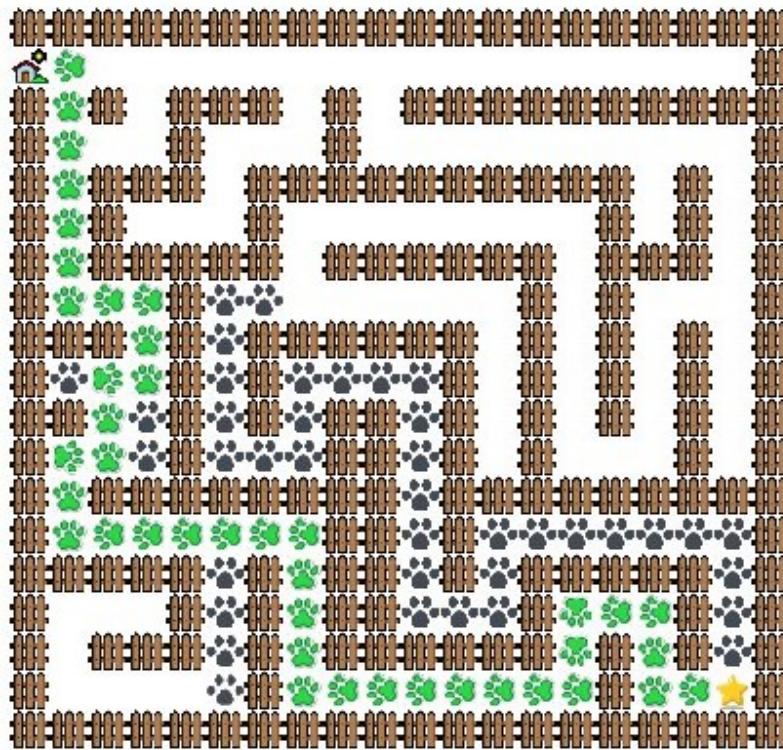


Hình 26: Kết quả chạy GBFS sử dụng Manhattan distance trên kích thước 4, Level 1

- Chi phí đường đi: 80
- Tổng số nút mở: 80

Việc thay đổi Heuristic đã giúp chúng ta đi từ kết quả thất bại đến kết quả tìm được đường đi đến đích.

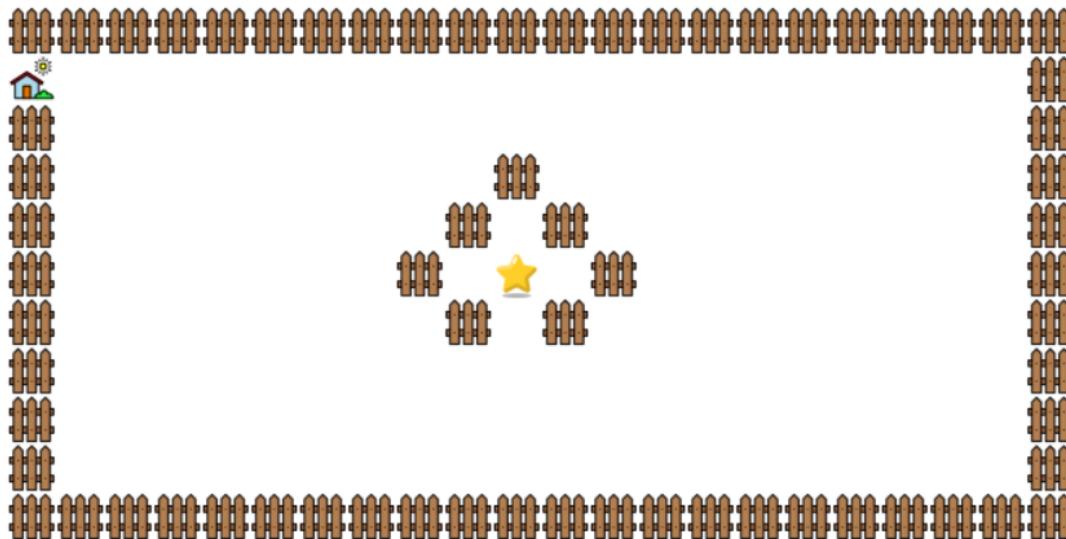
5. A*



Hình 27: Kết quả chạy A* trên kịch bản 4, Level 1

- Chi phí đường đi tìm được: 42
- Thời gian chạy: 0.001 (s)
- Tổng số nút mở: 80.

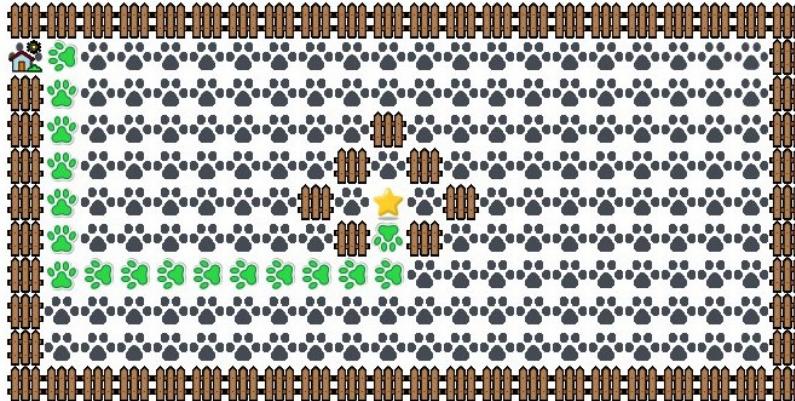
3.1.5 Kịch bản 5 - Bản đồ không có điểm thưởng



Hình 28: Kịch bản 5, Level 1

Kết quả chạy trên các thuật toán BFS, DFS, UCS, GBFS, A*:

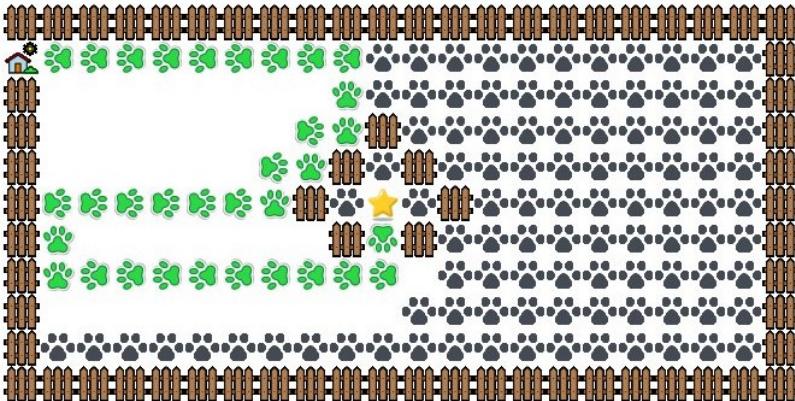
1. BFS



Hình 29: Kết quả chạy BFS trên kịch bản 5, Level 1

- Chi phí đường đi tìm được: 18
- Thời gian chạy: 0.000999 (s).
- Tổng số nút mở: 173

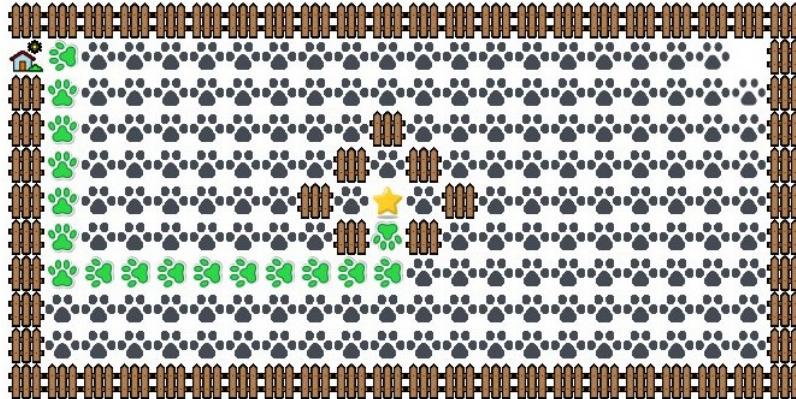
2. DFS



Hình 30: Kết quả chạy DFS trên kịch bản 5, Level 1

- Chi phí đường đi tìm được: 34
- Thời gian chạy: 0.001997(s)
- Tổng số nút mở: 133

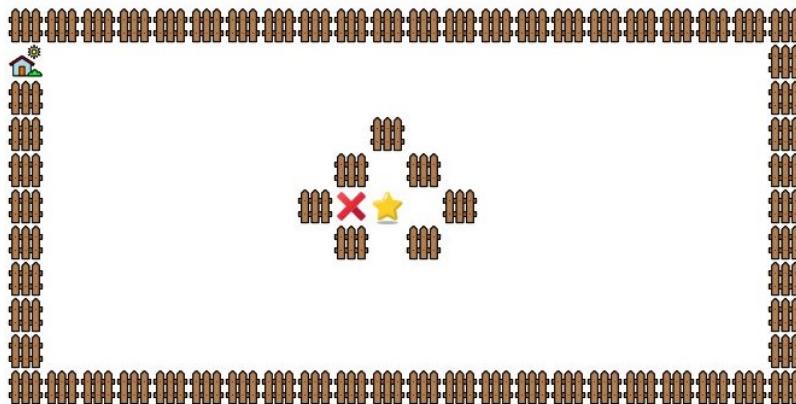
3. UCS



Hình 31: Kết quả chạy UCS trên kịch bản 5, Level 1

- Chi phí đường đi tìm được: 18
- Thời gian chạy: 0.002950 (s).
- Tổng số nút mở: 171

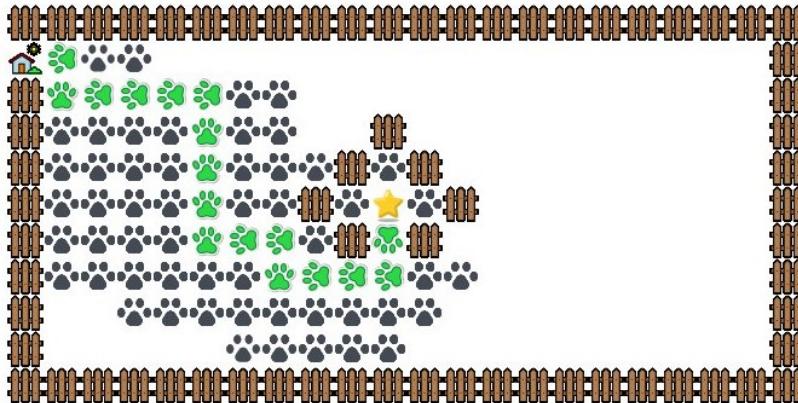
4. GBFS



Hình 32: Kết quả chạy GBFS trên kịch bản 5, Level 1

- Không tìm được đường đi đến đích.

5. A*



Hình 33: Kết quả chạy A* trên kịch bản 5, Level 1

- Chi phí đường đi tìm được: 18
- Thời gian chạy: 0.001084 (s)
- Tổng số nút mở: 70.

Nhận xét:

1. **BFS:** Vì BFS duyệt hết tất cả các bước đi cùng mức rồi mới duyệt tới mức kế tiếp nên BFS cần phải duyệt một lúc tất cả các con đường có thể trước khi có một con đường dẫn tới đích. Điều này dẫn tới BFS có các đặc tính sau:

- Tính đầy đủ: BFS đảm bảo tìm ra được đường đi đúng, nếu đường đi đó tồn tại. Trong 5 kịch bản của Level 1, BFS luôn tìm được đường đi tới đích.
- Tính tối ưu: Vì một bước đi cần một chi phí như nhau và việc duyệt tất cả các con đường cùng lúc nên tất cả các con đường có chi phí như nhau. Vậy đường đi đầu tiên tìm được là đường đi tốt nhất. Trong mỗi kịch bản của Level 1, chi phí của đường đi tìm được bởi BFS là đều là nhỏ nhất.
- Độ mở các nút: BFS có độ mở các nút cao, trong trường hợp xấu nhất có thể phải mở hết tất cả các ô trống mới tìm đường đe dọa ra. Trong mỗi kịch bản của Level 1, tổng số nút mở của BFS đều là số nút mở lớn nhất so với các thuật toán khác..

2. **DFS:** DFS chọn một con đường và duyệt hết con đường đó trước khi quay lại các lựa chọn khác. Điều này dẫn tới DFS có các đặc tính sau:

- Tính đầy đủ: DFS có khả năng quay lại duyệt các phương án khác khi một phương án thất bại, điều này đảm bảo tìm ra được đường đi đúng, nếu đường đi đó tồn tại. Trong 5 kịch bản trên, DFS luôn tìm được đường đi.
- Tính tối ưu: Nếu con đường DFS chọn dẫn tới đích thì DFS sẽ dừng lại. Điều này không đảm bảo tính tối ưu vì vẫn còn những con đường khác chưa được duyệt. Trong tất cả các kịch bản của Level 1, chi phí đường đi tìm được bởi DFS đều lớn hơn chi phí của BFS.
- Độ mở các nút: Trong trường hợp tốt nhất, DFS không cần phải mở thêm một nút nào ngoài con đường mình đã chọn (con đường đầu tiên dẫn tới đích). Trong trường hợp xấu nhất, DFS phải mở tất cả các nút trống để tìm tới đích. Trong 5 kịch bản trên, tổng số

nút mở của DFS luôn nhỏ hơn BFS, có nghĩa là DFS có khả năng tìm thấy đường đi sớm hơn BFS. Điều này có khả năng xảy ra vì DFS tìm thấy một con đường tới đích nào đó trước khi duyệt hết các con đường còn lại.

3. **UCS:** Vì UCS chọn con đường đi có chi phí nhỏ nhất trong tất cả các con đường có thể để đi tiếp nên có các đặc tính sau đây:

- Tính đầy đủ: UCS đảm bảo tìm ra được đường đi đúng, nếu đường đi đó tồn tại
- Tính tối ưu: UCS đảm bảo tìm ra được đường đi tới đích có chi phí nhỏ nhất. Trong tất cả các kịch bản trên, chi phí đường đi của UCS bằng chi phí đường đi của BFS và là nhỏ nhất. Tuy nhiên cách đi có thể khác với BFS.
- Độ mở các nút: BFS có độ mở các nút cao, trong trường hợp xấu nhất có thể phải mở hết tất cả các ô trống mới tìm đường đường ra. Trong hầu hết các kịch bản, tổng số nút mở của UCS là xấp xỉ so với BFS.

4. **GBFS:** Vì GBFS lựa chọn phương án tiếp theo hoàn toàn dựa vào Heuristic và GBFS "tham lam" chỉ quan tâm đến mục tiêu và các vị trí xung quanh mình nên nó có các đặc điểm sau đây:

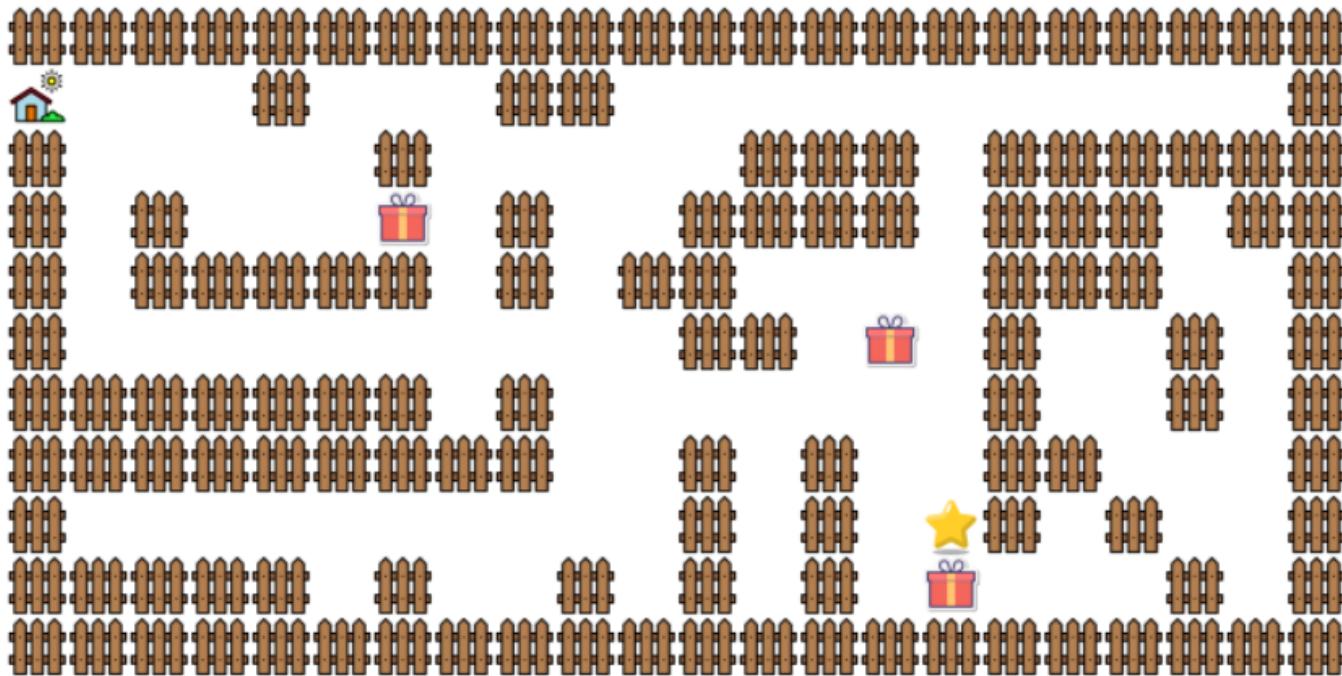
- Tính đầy đủ: GBFS không đảm bảo tìm được đường đi tới đích. Trong 5 kịch bản trên, GBFS đã bị thất bại trong 4 kịch bản. Điều này xảy ra khi Heuristic được sử dụng là không đủ tốt và GBFS bị kẹt trong trạng thái cục bộ.
- Tính tối ưu: GBFS không đảm bảo tính tối ưu.
- Độ mở các nút: Trong trường hợp GBFS tìm ra được đường đi, độ mở các nút của nó thường thấp hơn BFS hay DFS. Lý do là bởi GBFS dựa vào Heuristic để di chuyển theo hướng mà nó nghĩ là gần đích nhất, chứ nó không di chuyển theo các hướng còn lại như BFS hay DFS.

5. **A*:** A* có thể xem là thuật toán tìm kiếm thông minh vì nó áp dụng thêm tri thức Heuristic để làm quá trình tìm kiếm diễn ra nhanh hơn nhưng vẫn không làm mất tính đầy đủ của thuật toán và không làm giảm nhiều tính tối ưu của thuật toán:

- Tính đầy đủ: A* đảm bảo tìm được đường đi tới đích.
- Tính tối ưu: Con đường tìm ra bởi A* không đảm bảo là con đường tối ưu. Nhưng trong nhiều trường hợp, độ chênh lệnh chi phí so với chi phí tối ưu là không cao. Trong 5 kịch bản của Level 1, chi phí của A* là ngang bằng so với chi phí tối ưu của BFS hay UCS.
- Độ mở các nút: Vì áp dụng tri thức Heuristic khiến cho đường đi của A* ưu tiên con đường hướng về đích. Trong tất cả các kịch bản của Level1, A* cho độ mở các nút tốt hơn BFS hay UCS.

3.2 Bản đồ có điểm thưởng

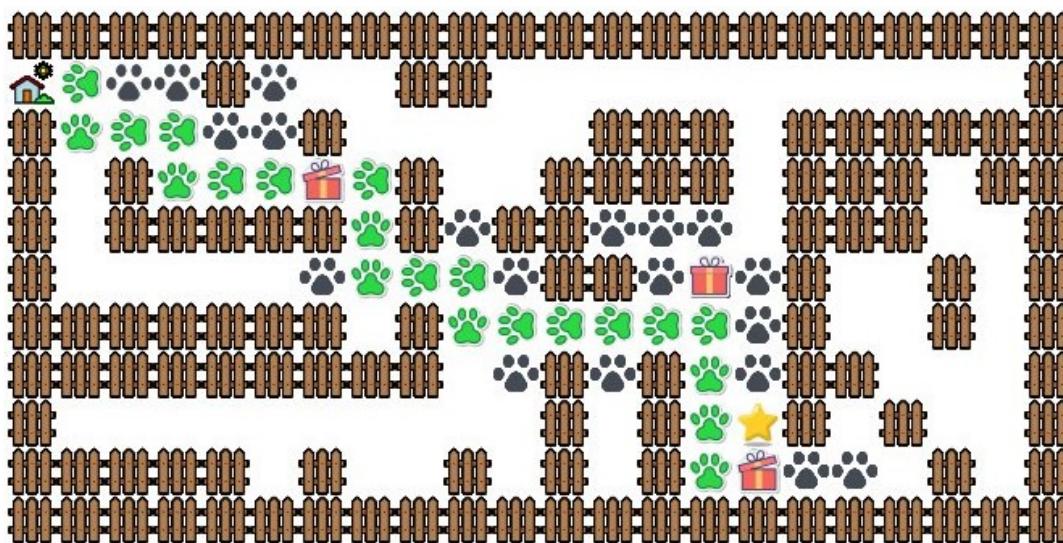
3.2.1 Kịch bản 1 - Bản đồ có điểm thưởng:



Hình 34: Kịch bản 1, Level 2

Kết quả chạy trên các thuật toán Astar _ Lv2, Dijktra, Quy hoạch động:

- Astar _ Lv2:

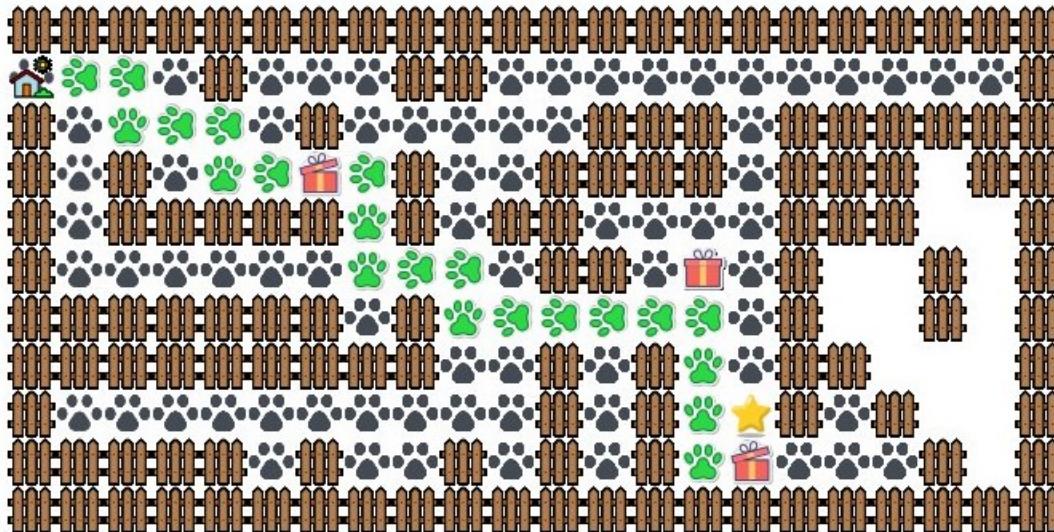


Hình 35: Kết quả chạy A* _ Lv2 trên kịch bản 1, Level 2

- Chi phí đường đi tìm được: 16

- Các điểm thưởng đã ăn: (3, 6) với -3 điểm, (9, 15) với -5 điểm,
- Thời gian chạy: 0.002007 (s)
- Tổng số nút mở: 43.

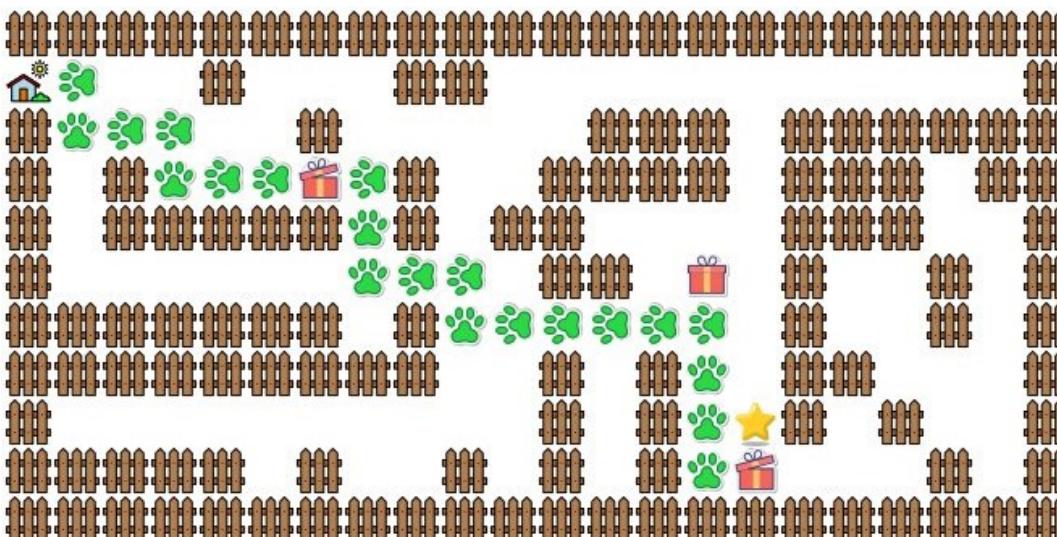
- Dijktra:



Hình 36: Kết quả chạy Dijkstra trên kịch bản 1, Level 2

- Chi phí đường đi tìm được: 16
- Các điểm thưởng đã ăn: (3, 6) với -3 điểm, (9, 15) với -5 điểm,
- Thời gian chạy: 0.151842 (s)
- Tổng số nút mở: 94.

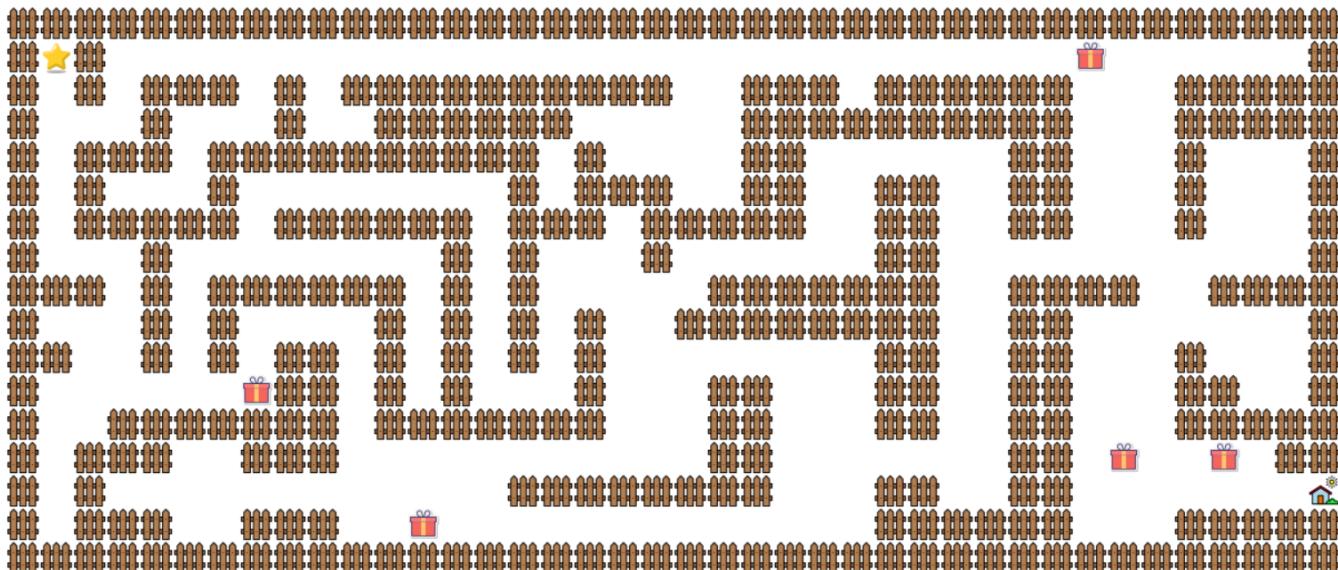
- Quy hoạch động:



Hình 37: Kết quả chạy Quy hoạch động trên kịch bản 1, Level 2

- Chi phí đường đi tìm được: 16
- Các điểm thưởng đã ăn: (3, 6) với -3 điểm, (9, 15) với -5 điểm,
- Thời gian chạy: 0.006998 (s)

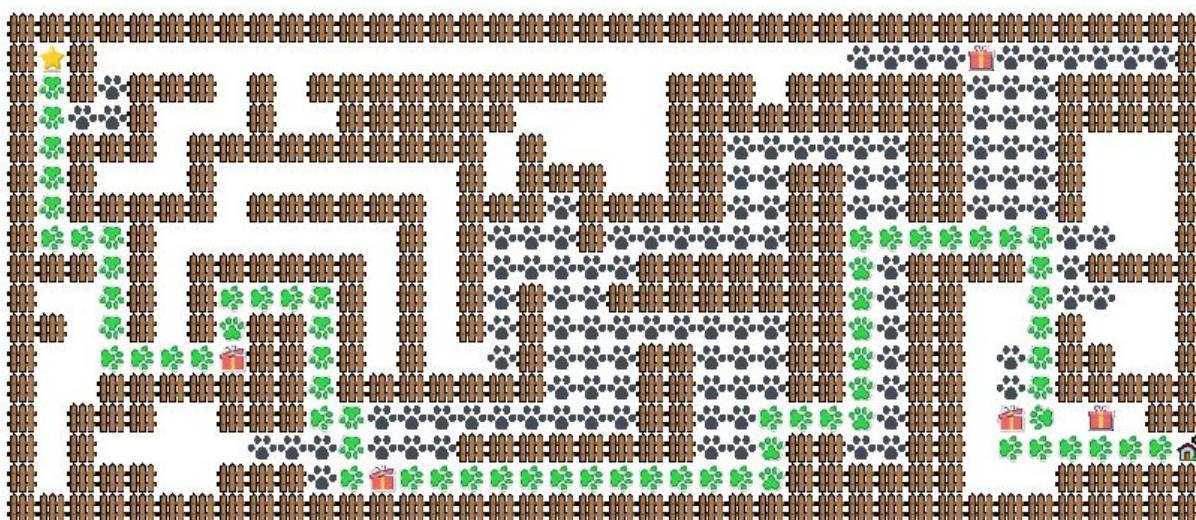
3.2.2 Kịch bản 2 - Bản đồ có điểm thưởng:



Hình 38: Kịch bản 1, Level 2

Kết quả chạy trên các thuật toán Astar_Lv2, Dijktra, Quy hoạch động:

- Astar_Lv2 sử dụng Heuristic Euclidean distance:



Hình 39: Kết quả chạy A*_Lv2 trên kịch bản 2, Level 2

- Chi phí đường đi tìm được: 31

- Các điểm thưởng đã ăn: (11, 7) với -20 điểm, (13, 33) với -10 điểm, (15, 12) với -12 điểm.
- Thời gian chạy: 0.013097 (s)
- Tổng số nút mở: 191.

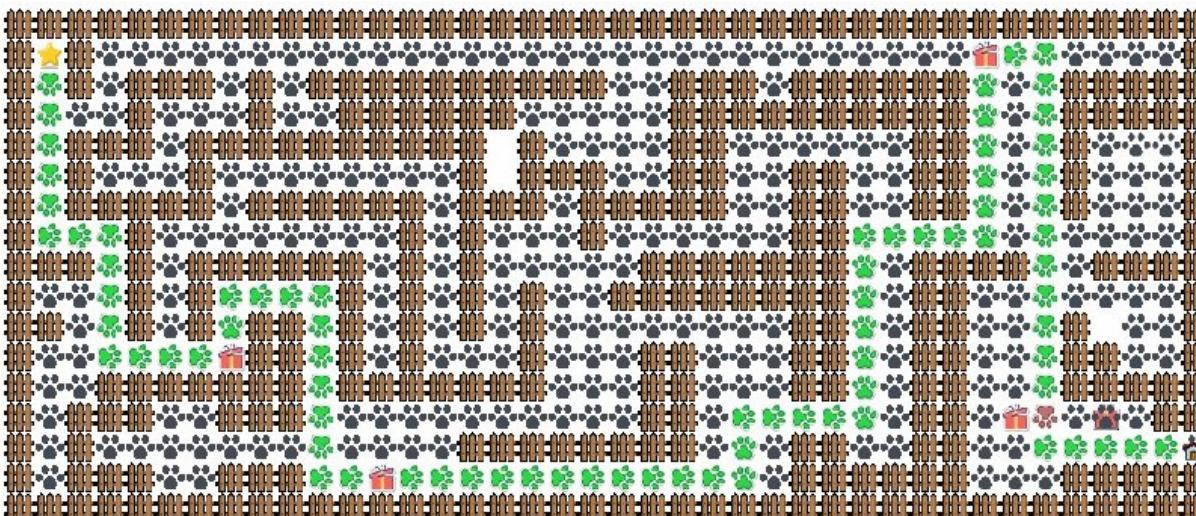
- Astar_Lv2 sử dụng Heuristic Euclidean distance:



Hình 40: Kết quả chạy A*_Lv2 sử dụng Heuristic Manhattan distance trên kịch bản 2, Level 2

- Chi phí đường đi tìm được: 28
- Các điểm thưởng đã ăn: (11, 7) với -20 điểm, (13, 33) với -10 điểm, (15, 12) với -12 điểm, (1,32) với -15 điểm.

- Dijkstra:

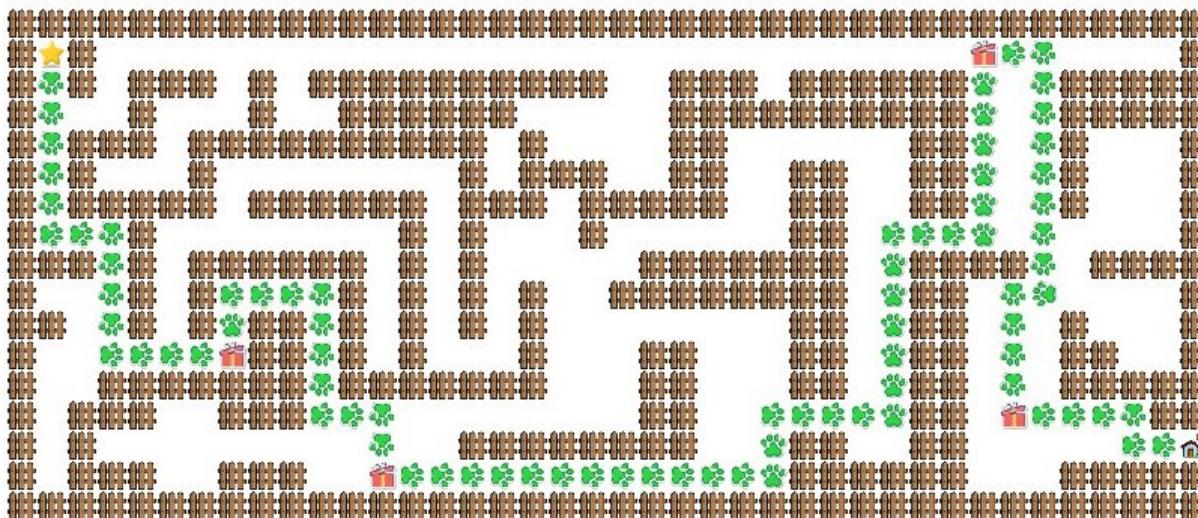


Hình 41: Kết quả chạy Dijkstra trên kịch bản 2, Level 2

- Chi phí đường đi tìm được: 28

- Các điểm thưởng đã ăn: (11, 7) với -20 điểm, (13, 33) với -10 điểm, (15, 12) với -12 điểm, (1,32) với -15 điểm.
- Thời gian chạy: 0.259053 (s)
- Tổng số nút mở: 323.

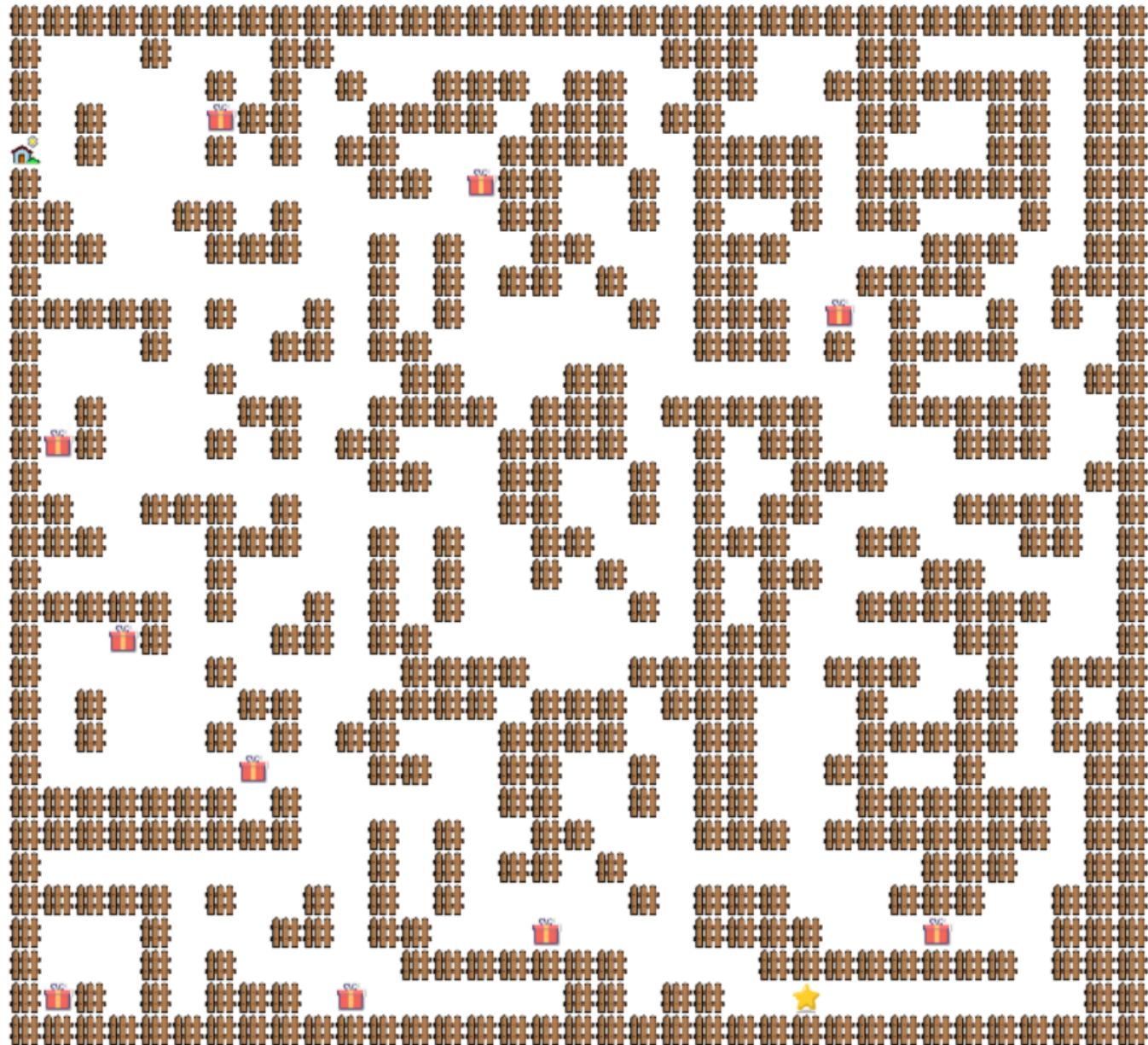
• **Quy hoạch động:**



Hình 42: Kết quả chạy Quy hoạch động trên kịch bản 2, Level 2

- Chi phí đường đi tìm được: 28
- Các điểm thưởng đã ăn: (11, 7) với -20 điểm, (13, 33) với -10 điểm, (15, 12) với -12 điểm, (1,32) với -15 điểm.
- Thời gian chạy: 0.051256 (s)

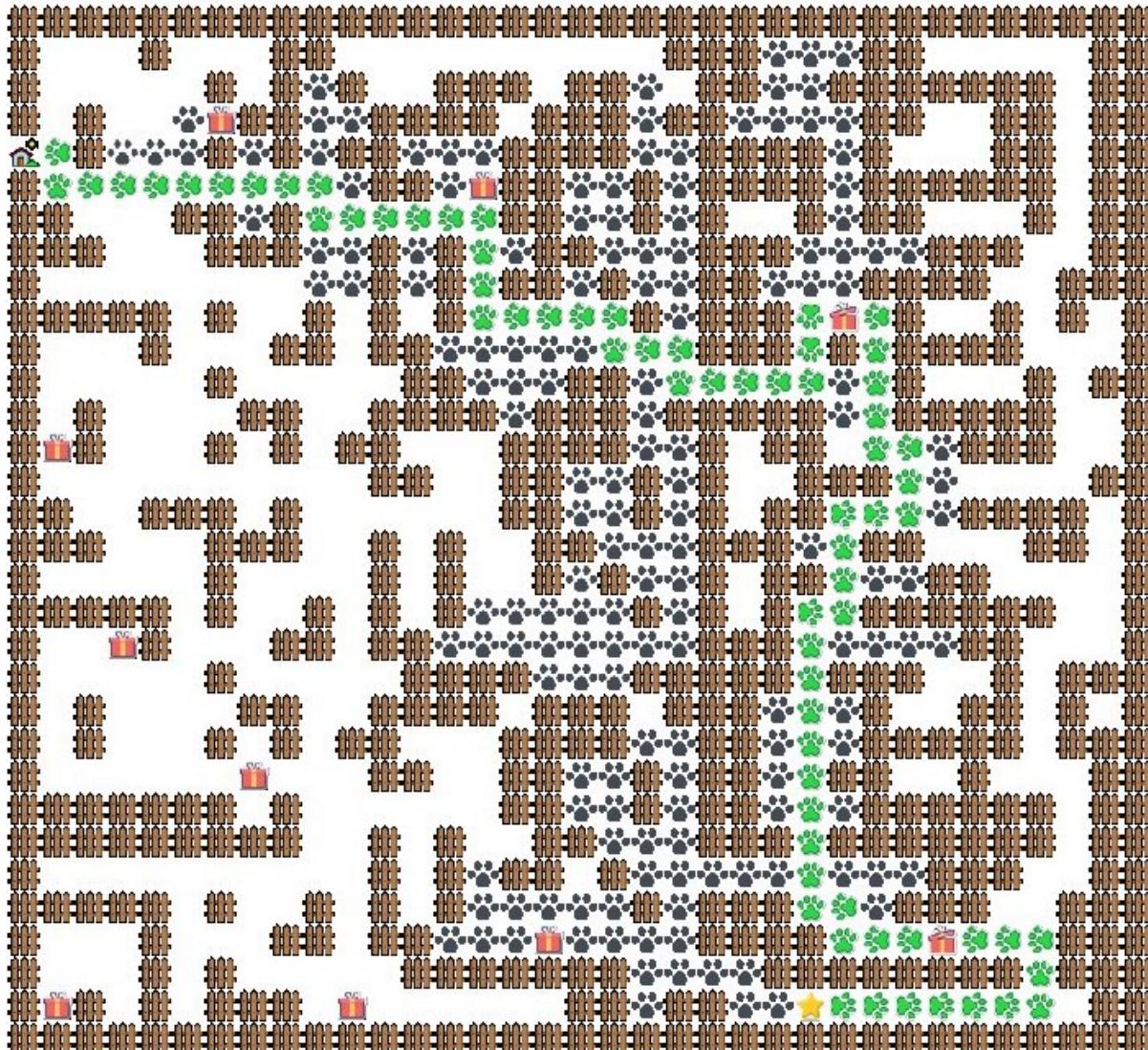
3.2.3 Kịch bản 3 - Bản đồ có điểm thưởng:



Hình 43: Kịch bản 3, Level 2

Kết quả chạy trên các thuật toán Astar_Lv2, Dijkstra, Quy hoạch động:

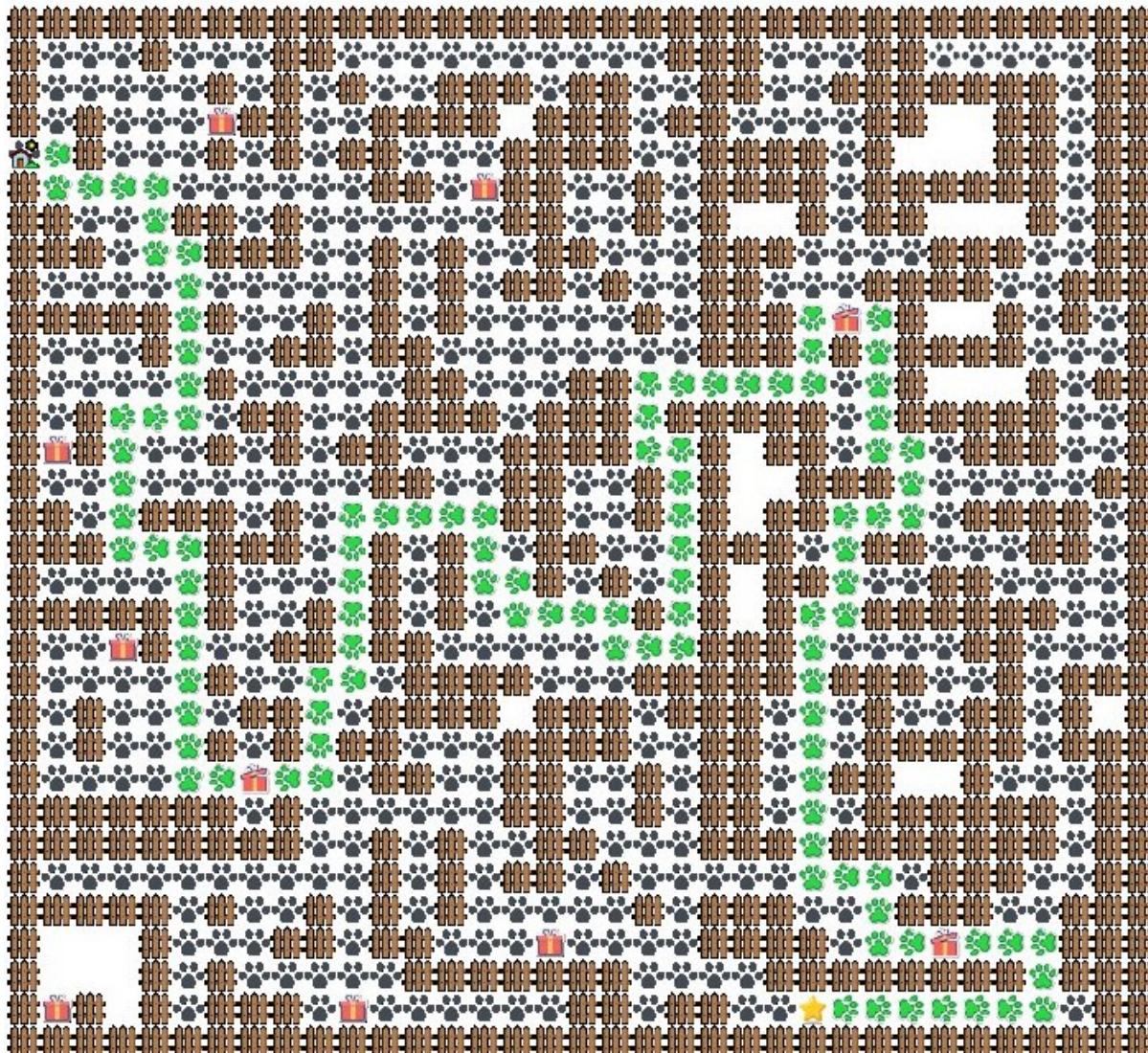
- Astar_Lv2:



Hình 44: Kết quả chạy A* _ Lv2 trên kịch bản 3, Level 2

- Chi phí đường đi tìm được: 2
- Các điểm thưởng đã ăn: (9, 25) với -50 điểm, (28, 28) với -22 điểm.
- Thời gian chạy: 0.035007 (s)
- Tổng số nút mở: 236.

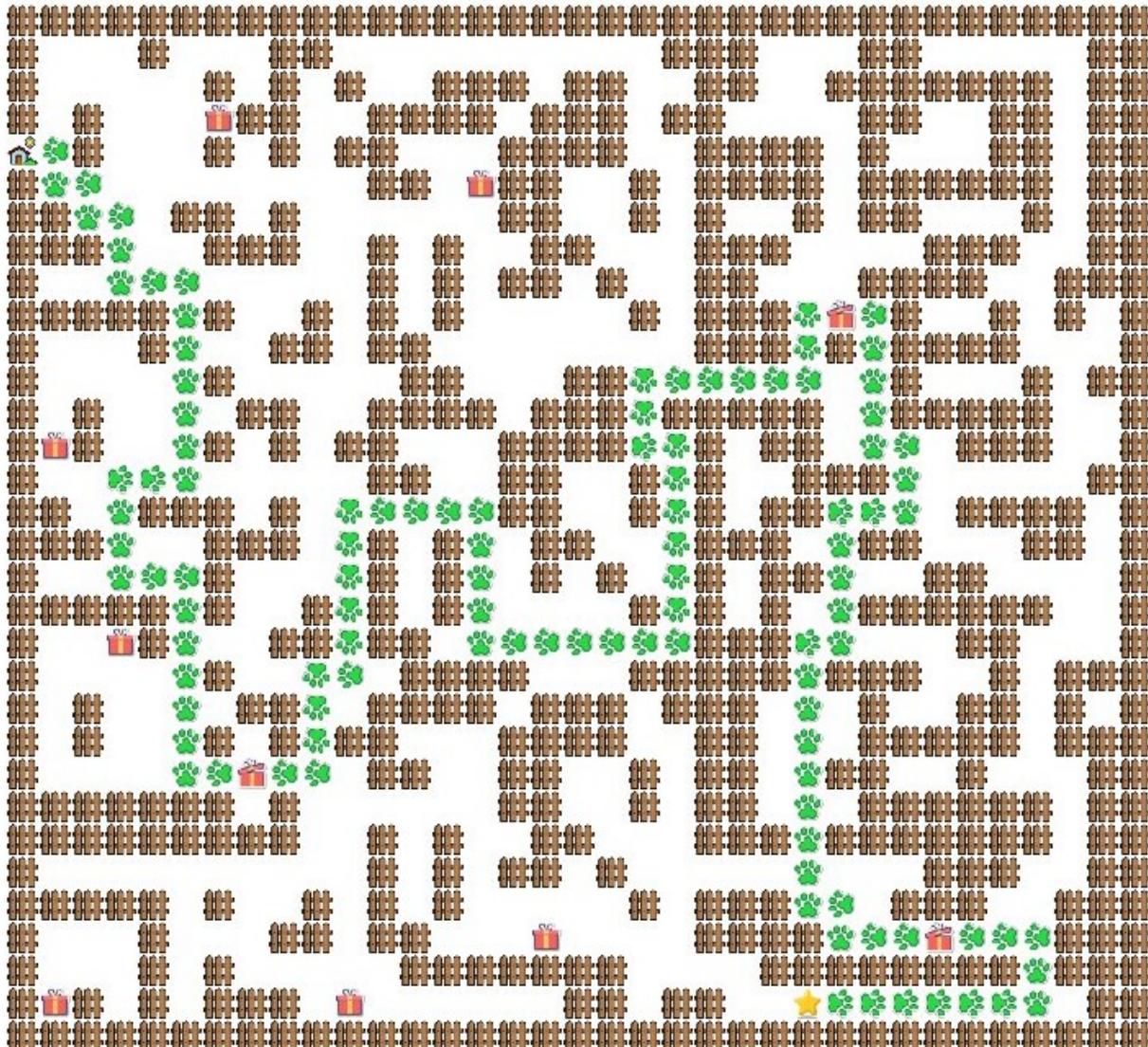
- **Dijkstra:**



Hình 45: Kết quả chạy Dijkstra trên kịch bản 3, Level 2

- Chi phí đường đi tìm được: -5
- Các điểm thưởng đã ăn: (9, 25) với -50 điểm, (28, 28) với -22 điểm, (23, 7) với -45 điểm.
- Thời gian chạy: 3.022606 (s)
- Tổng số nút mở: 518.

- **Quy hoạch động:**



Hình 46: Kết quả chạy Quy hoạch động trên kinh bản 3, Level 2

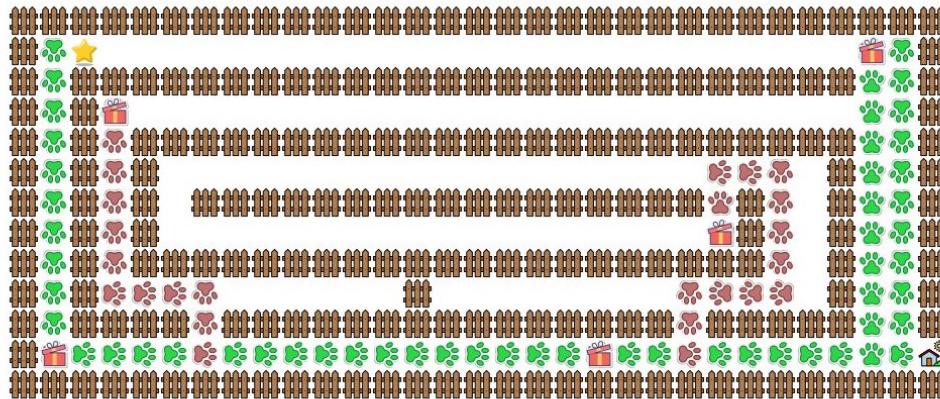
- Chi phí đường đi tìm được: -5
- Các điểm thưởng đã ăn: (9, 25) với -50 điểm, (28, 28) với -22 điểm, (23, 7) với -45 điểm.
- Thời gian chạy: 0.318470 (s)

3.3 Bản đồ có các điểm đón - Mức 3

3.3.1 Kịch bản 1 - Bản đồ có điểm đón:

Kết quả chạy trên các thuật toán Genetic, Hill Climbing:

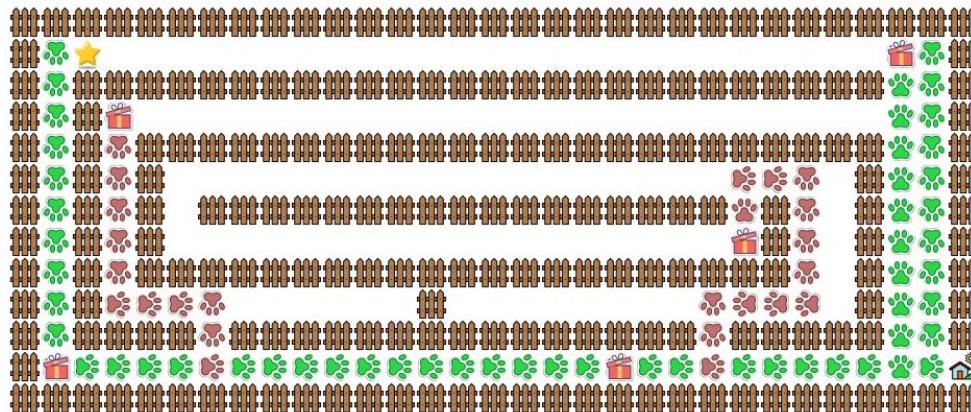
- Genetic:



Hình 47: Kết quả chạy Genetic trên kịch bản 1, Level 3

– Chi phí đường đi tìm được: 43

- Hill Climbing:



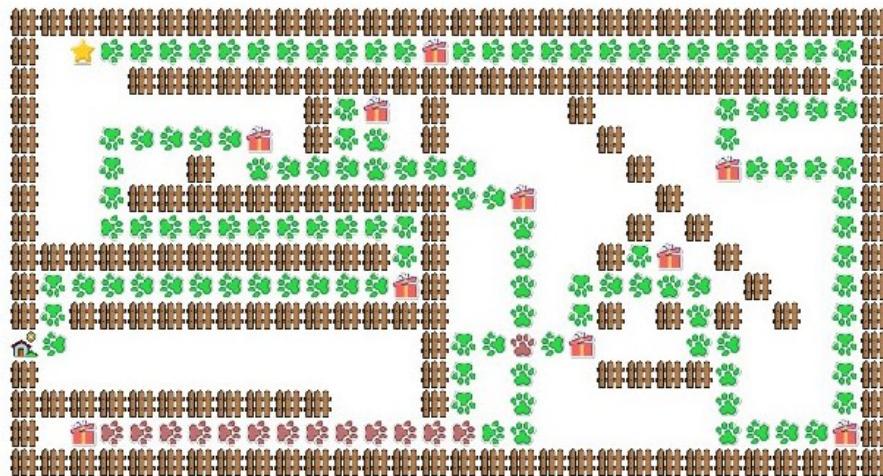
Hình 48: Kết quả chạy Hill Climbing trên kịch bản 1, Level 3

– Chi phí đường đi tìm được: 43

3.3.2 Kịch bản 2 - Bản đồ có điểm đón:

Kết quả chạy trên các thuật toán Genetic, Hill Climbing:

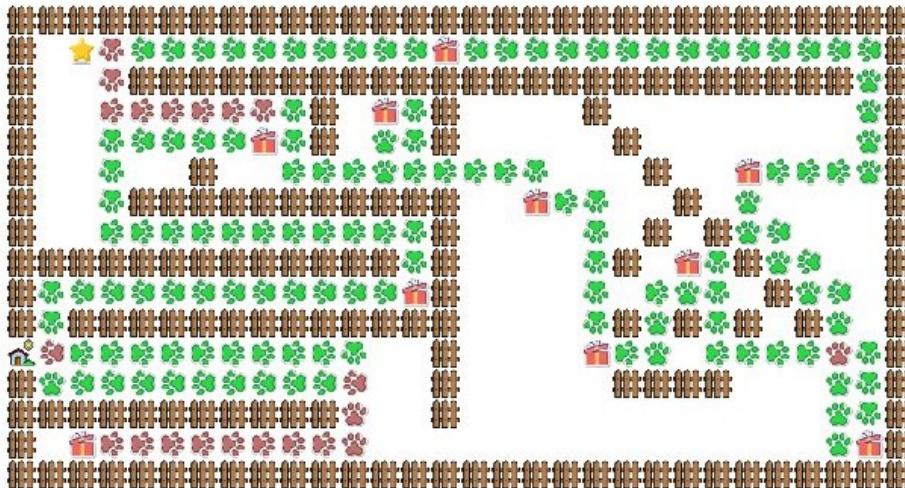
- Genetic:



Hình 49: Kết quả chạy Genetic trên kịch bản 2, Level 3

– Chi phí đường đi tìm được: 67

- Hill Climbing:



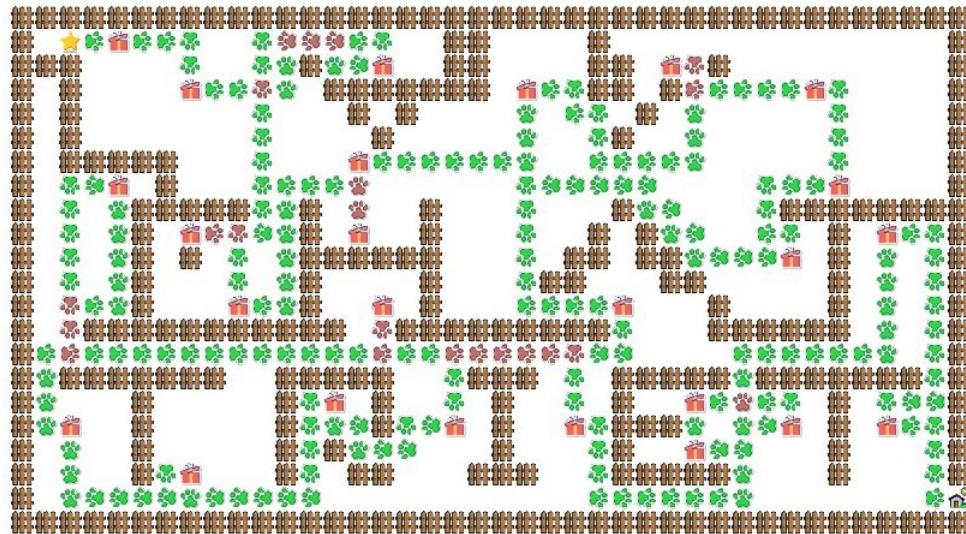
Hình 50: Kết quả chạy Hill Climbing trên kịch bản 2, Level 3

– Chi phí đường đi tìm được: 92

3.3.3 Kịch bản 3 - Bản đồ có điểm đón:

Kết quả chạy trên các thuật toán Genetic, Hill Climbing:

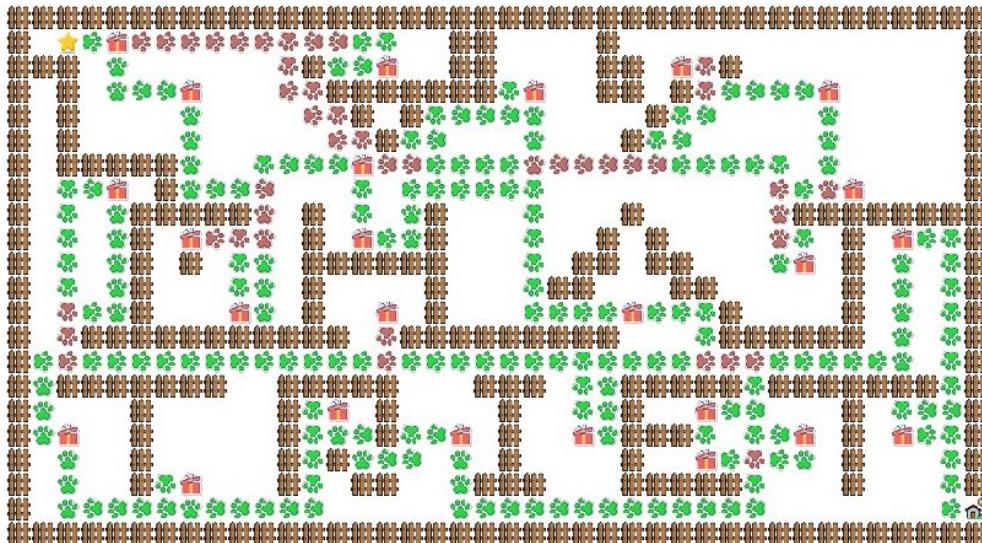
- **Genetic:**



Hình 51: Kết quả chạy Genetic trên kịch bản 3, Level 3

– Chi phí đường đi tìm được: 155

- **Hill Climbing:**



Hình 52: Kết quả chạy Hill Climbing trên kịch bản 3, Level 3

– Chi phí đường đi tìm được: 207

3.4 Kịch bản nâng cấp

3.4.1 Thiết kế bản đồ và quy tắc về điểm dịch chuyển

Bản đồ teleport được thiết kế như sau:

- Dòng đầu tiên chứa số nguyên n là số cổng dịch chuyển.
- Trên n dòng tiếp theo, mỗi dòng chứa 4 số nguyên x_1, y_1, x_2, y_2 với (x_1, y_1) là tọa độ của cổng dịch chuyển đầu vào và (x_2, y_2) là tọa độ của cổng đầu ra.
- Các dòng tiếp theo mô tả bản đồ mê cung. Bản đồ ở kịch bản sẽ giống hoàn toàn các bản đồ cũ, chỉ có một điểm khác biệt là các cổng dịch chuyển. Các cổng dịch chuyển đầu vào sẽ được ký hiệu bởi ký tự **o** và các cổng dịch chuyển đầu ra sẽ được ký hiệu bởi ký tự **O**.

Lưu ý: Ở bản đồ nâng cao này, nhóm sẽ làm một bản đồ dịch chuyển 1 chiều, và từ cổng dịch chuyển đầu vào sẽ bắt buộc phải đi đến cổng dịch chuyển đầu ra chứ không được tùy ý đi 4 hướng lân cận như bình thường.

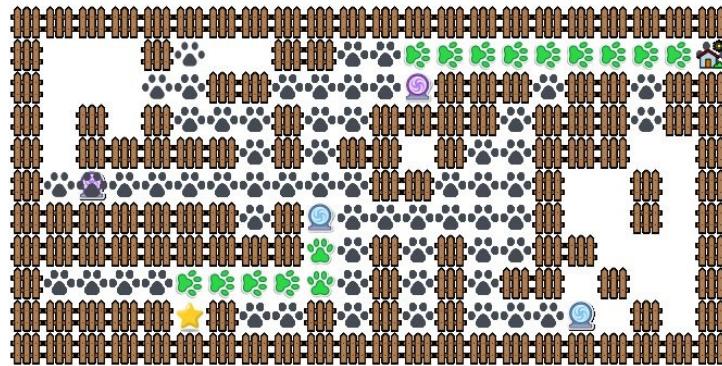
The screenshot shows a text editor interface with a toolbar at the top. The menu bar includes 'File', 'Edit', and 'View'. Below the menu is a tab bar with several tabs: 'xt', 'UCS.txt', 'BFS.txt', 'UCS.txt', and 'input1.txt'. The 'input1.txt' tab is currently active. The main text area displays the following content:

```

2
6 9 2 12
9 17 5 2
XXXXXXXXXXXXXXXXXXXXXX
X   X   XX
X       XX      OXXX  XX  XX
X  X  X   X   XXXX  XXX  XX
X  XXXXX  X  XX  X   XXX   X
X  0       XX   X   X  X
XXXXXXX  XO       X   X  X
XXXXXXXXXX  X  X   XX   X
X           X  X  XX  X   X
XXXXxxSx  X  X  X   o  X  X
XXXXXXXXXXXXXXXXXXXXXX

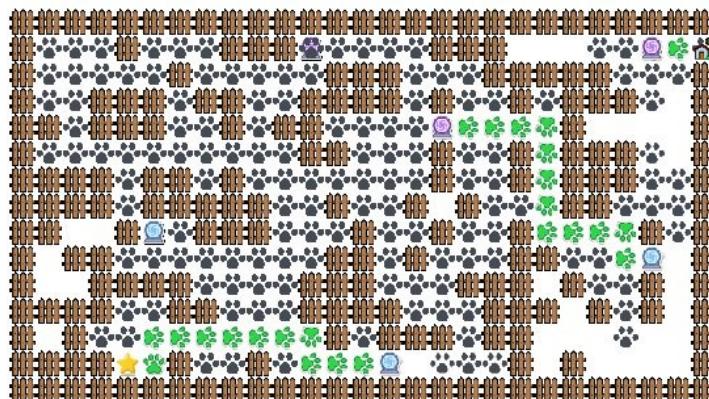
```

Hình 53: Minh họa input của bản đồ nâng cao - Teleport

3.4.2 Kịch bản 1 - Bản đồ có điểm dịch chuyển:

Hình 54: Kết quả chạy BFS trên kịch bản 1, bản đồ nâng cao

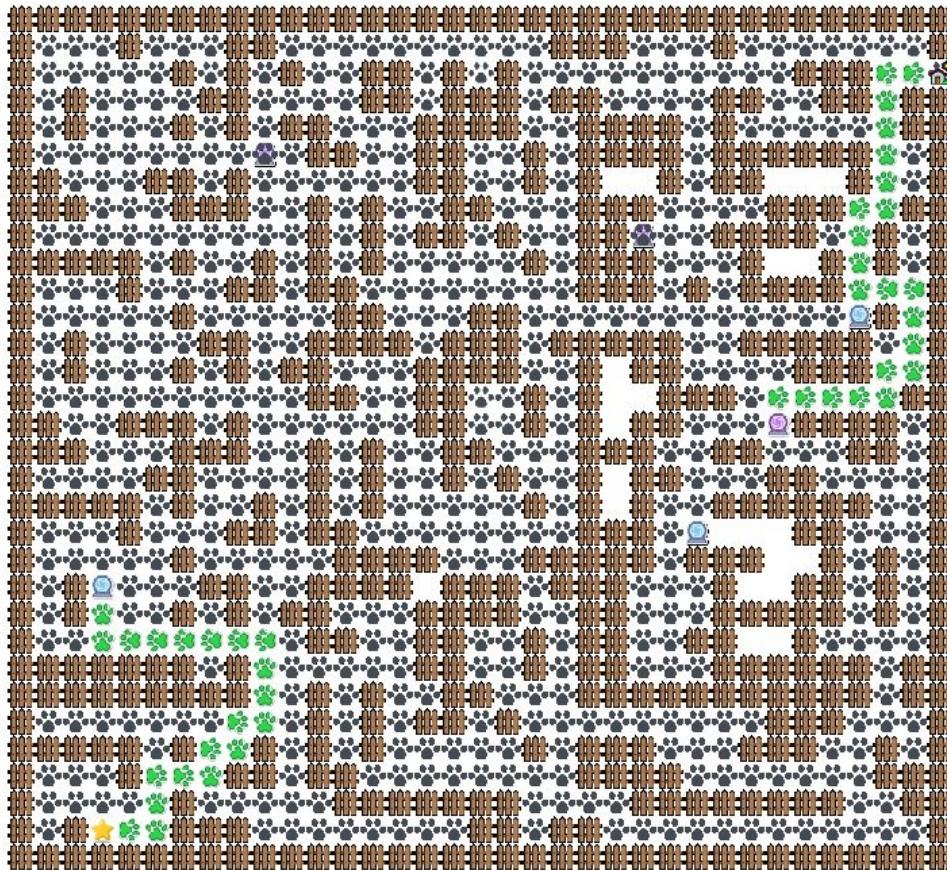
- Chi phí đường đi tìm được: 17
- Các điểm dịch chuyển đã đi qua: (6, 9) dịch chuyển tới (2, 12)
- Thời gian chạy: 0.002007(s)

3.4.3 Kịch bản 2 - Bản đồ có điểm dịch chuyển:

Hình 55: Kết quả chạy BFS trên kịch bản 2, bản đồ nâng cao

- Chi phí đường đi tìm được: 27
- Các điểm dịch chuyển đã đi qua: (13, 14) dịch chuyển tới (4, 16) và từ (8, 5) dịch chuyển tới (1, 11)
- Thời gian chạy: 0.002983(s)

3.4.4 Kịch bản 3 - Bản đồ có điểm dịch chuyển:



Hình 56: Kết quả chạy BFS trên kịch bản 3, bản đồ nâng cao

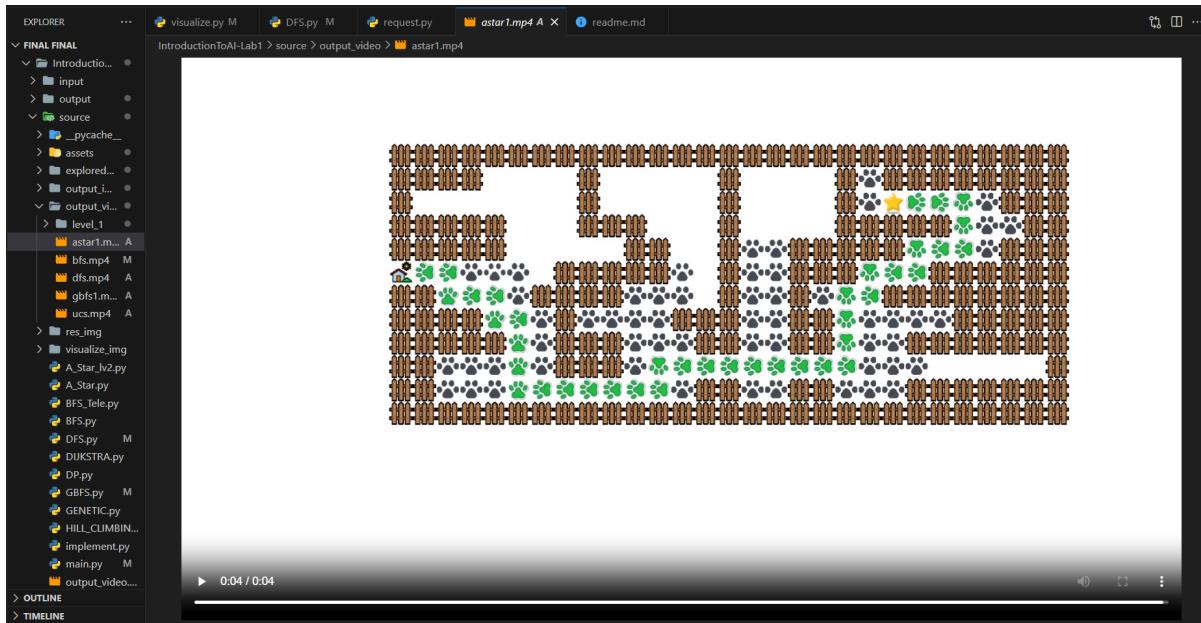
- Chi phí đường đi tìm được: 44
- Các điểm dịch chuyển đã đi qua: (21, 3) dịch chuyển tới (15, 28)
- Thời gian chạy: 0.032983(s)

4 Video minh họa

Thay vì chỉ output ra đường đi thoát khỏi mê cung, nhóm đã output ra video thể hiện quá trình tìm kiếm (bao gồm các nút được mở, đường đi cuối cùng,...)

BÁO CÁO ĐỒ ÁN

- Import thêm các thư viện và tài liệu cần thiết: pygame, imageio, numpy, các thuật toán đã cài đặt.
- Load ảnh cho các phần tử trên bản đồ: Sử dụng thư viện Pygame để tải các hình ảnh cho các phần tử như tường, điểm xuất phát, điểm kết thúc, vật phẩm, vị trí của người chơi, v.v.
- Lưu mô phỏng Pygame thành một tệp mp4: tạo một biến writer để ghi video, sau đó ghi từng khung hình của mô phỏng vào biến này.



Hình 57: Video minh họa được lưu trong thư mục *output_video*

5 Mô tả các câu lệnh trong file run.sh

File run.sh chứa toàn bộ các câu lệnh dùng để chạy tất cả các thuật toán cho tất cả level 1, 2, 3 và advance.

Cú pháp tổng quát:

```
python ./source/main.py <level> <input> <algorithm_name>
```

Trong đó

- level thuộc {1,2,3, advance}
- input thuộc {1,2,3,4,5} (Lưu ý input4,5 chỉ có trong level 1)
- algorithm_name thuộc [dfs, bfs, ucs, gbfs, dijkstra, astar, astar_lv2 , hill_climbing, genetic, bfs_tele, dp]

Ví dụ:

```
python ./source/main.py 1 3 bfs
```

Code cho tất cả các yêu cầu của đồ án được hoàn thành khi chạy file run.sh từ terminal:

```
bash run.sh
```

Tài liệu

- Lê Hoài Bắc, Tô Hoài Việt, Cơ sở Trí tuệ nhân tạo, Nhà xuất bản Khoa học và Kỹ Thuật, 2014.
- Save Pygame simulation as an mp4 file
- Thư viện Pygame
- Hàm vẽ bản đồ
- Search Algorithms in AI