**Lab2: Search: Solving a Maze Using a Goal-based Agent**

**Task 1.1**: Xác định rõ ràng các thành phần của một bài toán tìm kiếm (search problem) để có thể áp dụng các thuật toán như BFS, DFS, A\*, …. Initial state, Actions, Transition model, Goal state ,Path cost

1. Initial State (Trạng thái ban đầu)

Trạng thái ban đầu là vị trí xuất phát S trong mê cung. Biểu diễn dưới dạng tọa độ hàng – cột: S=(rs​,cs​) Và find\_positions() trả về S.

2. Actions (Tập hành động)

Tại mỗi ô, agent có thể di chuyển theo một trong bốn hướng: A={Up,Down,Left,Right} Tuy nhiên, chỉ những hành động dẫn đến ô không phải tường và nằm trong mê cung mới hợp lệ Hàm valid\_moves(arr, pos) sinh ra các hành động hợp lệ dưới dạng (Action, NewState).

3. Transition Model (Mô hình chuyển trạng thái)

Nếu ở trạng thái s=(r,c) và thực hiện hành động a, ta đến trạng thái mới s′=(r′,c′)

Công thức: Result(s,a)=s′ Với ràng buộc: arr[r′,c′]≠ # và 0≤r′<rows ,  0≤c′<cols. Được hiện thực bởi valid\_moves().

4. Goal State (Trạng thái đích)

Trạng thái đích là tọa độ G trong mê cung. Bài toán kết thúc khi agent đạt tới G=(rg,cg). Và find\_positions() trả về G.

5. Path Cost (Chi phí đường đi)

Mỗi bước di chuyển từ một ô sang ô kề cạnh có chi phí bằng **1**.Do đó, chi phí đường đi chính là tổng số bước từ S đến GĐây chính là độ dài đường đi ngắn nhất (shortest path length).Được tính bằng BFS trong bfs\_shortest\_length(arr, S, G, free\_coords)

Dựa trên việc phân tích các file mê cung trong code, ta có thể đo được Kích thước mê cung: số hàng (rows) × số cột (cols). Số lượng ô trống (N\_free): số trạng thái có thể đi qua. Độ phân nhánh trung bình (b\_avg): trung bình số hành động hợp lệ trên mỗi trạng thái. Khoảng cách Manhattan (d\_man): ước lượng khoảng cách từ S đến G. Độ dài đường đi ngắn nhất (d\_bfs): kết quả BFS trả về.

A screenshot of a black screen

AI-generated content may be incorrect.

**Task1.2**. **Cho một mê cung, hãy ước lượng kích thước bài toán tìm kiếm thông qua các tham số sau:**

**n**: số lượng trạng thái trong state space (state space size).

**d**: độ sâu của nghiệm tối ưu (depth of the optimal solution).

**m**: độ sâu cực đại của cây trạng thái (maximum depth of tree).

**b**: hệ số phân nhánh cực đại (maximum branching factor).

Mô tả cách xác định các giá trị này cho một mê cung cụ thể.

Để ước lượng kích thước bài toán tìm kiếm trong mê cung, ta có thể xác định các tham số như sau:

1. Số trạng thái n

Mỗi trạng thái tương ứng với một ô trống trong mê cung mà agent có thể đứng.

Do đó, n chính là số lượng ô khác # trong mê cung.

n=∣{(r,c)∣arr[r,c]≠#}∣

count\_free(arr) trả lời được trạng thái tương ứng với một ô trống trong mê cung

1. Độ sâu nghiệm tối ưu d

Đây là số bước đi ngắn nhất từ trạng thái xuất phát S đến đích G.

Xác định bằng cách chạy BFS từ S đến G. d=shortest\_path\_length(S,G)

bfs\_shortest(arr, S, G) bước đi ngắn nhất từ trạng thái xuất phát S đến đích G.

1. Độ sâu cực đại m

Độ sâu cực đại của cây trạng thái trong mê cung bị chặn bởi số lượng trạng thái có thể đi qua.

Một upper bound là: m≤n−1 vì không có đường đi nào dài hơn số trạng thái khả dĩ. m\_upper = n - 1.

1. Hệ số phân nhánh cực đại b

Hệ số phân nhánh là số lượng hành động hợp lệ từ một trạng thái.

b là giá trị cực đại trên toàn bộ các trạng thái.

compute\_b\_avg\_and\_bmax(arr) trả về b\_max.

Bảng kết quả giá trị từ các mê cung:

A screenshot of a computer

AI-generated content may be incorrect.

Nhận xét:

Khi **d** tăng, chi phí và node expanded của BFS tăng *rất nhanh* (mũ theo b). Vì vậy với mazes lớn/khó, BFS rất tốn bộ nhớ.

**Task 2: Uninformed Search – Breadth-First and Depth-First**

Cần cài đặt hai chiến lược tìm kiếm không thông tin (uninformed search):

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)

Phải tuân theo pseudocode trong sách/slide.

Không lưu thông tin trực tiếp trong bản đồ (map). Chỉ sử dụng cây tìm kiếm với các cấu trúc frontier và reached

BFS: sử dụng hàng đợi (queue) làm frontier.

DFS: cần cài đặt đúng cách để tận dụng ưu điểm bộ nhớ nhỏ, không dùng reached toàn cục, chỉ kiểm tra cycle trong đường đi hiện tại.

1. Cấu trúc dữ liệu Node

Mỗi trạng thái trong cây tìm kiếm được lưu trong một đối tượng Node, gồm:

pos: tọa độ trong mê cung.

parent: tham chiếu về node cha.

action: hành động từ cha → node này.

cost: độ sâu hoặc chi phí đường đi từ gốc đến node này.

Node hỗ trợ phương thức get\_path\_from\_root() để truy vết đường đi từ trạng thái ban đầu S đến trạng thái hiện tại.

2. Thuật toán Breadth-First Search (BFS)

Frontier: **queue (FIFO)**.

Sử dụng tập reached để tránh mở rộng lại các trạng thái đã thăm.

BFS đảm bảo tìm được nghiệm tối ưu (nếu tồn tại) vì mở rộng theo độ sâu tang dần.

Các thông số đo lường:

nodes\_expanded: số node được mở rộng.

max\_depth: độ sâu lớn nhất đạt được trong quá trình tìm kiếm.

max\_frontier\_size: kích thước frontier lớn nhất.

path\_cost: số bước đi từ S đến G.

3. Thuật toán Depth-First Search (DFS)

Frontier: stack (LIFO).

Không dùng reached toàn cục (để giảm bộ nhớ), chỉ duy trì tập các trạng thái trong **đường đi hiện tại** (path\_set) để kiểm tra cycle.

Có tham số max\_depth\_limit để tránh đi quá sâu gây lặp vô hạn.

DFS có thể tìm nghiệm nhanh nhưng không đảm bảo tối ưu và có thể thất bại nếu cycle không được xử lý.

Các thông số đo lường giống BFS.

**Thực nghiệm trên các mê cung:**

A screenshot of a computer

AI-generated content may be incorrect.

Nhận xét :

Path cost DFS rất lớn so với BFS . VVì BFS tìm theo lớp (layer-by-layer) nên **đảm bảo tìm đường ngắn nhất** trong đồ thị đơn vị chi phí. DFS tìm theo chiều sâu nên thường tìm một đường “tình cờ” — có thể rất dài.

BFS mở rộng 207 node, DFS mở rộng 120 node.

Vì BFS phải khám phá toàn bộ các node ở các lớp nông trước khi tới lớp chứa goal (ở đây goal ở depth ≈14), nên tổng nodes expanded có thể lớn hơn. DFS vào sâu nhanh, nên trong trường hợp này nó đã tìm được một đường (không tối ưu) mà không cần mở quá nhiều node.

**Task 2.2: BFS và DFS (không có cấu trúc dữ liệu reached) xử lý các vòng lặp (cycles) như thế nào?**

Reached là dictionary lưu các state đã thăm

Nếu BFS không có reached, thì mỗi lần gặp lại một trạng thái cũ, nó vẫn sẽ được đưa vào frontier để mở rộng lại. Kết quả là số lượng node sẽ tăng lên cực nhanh (do lặp lại liên tục), khiến BFS tiêu tốn rất nhiều bộ nhớ và thời gian, thậm chí chạy mãi không kết thúc trong môi trường có chu trình.

Khi DFS mở rộng một node mới, nó sẽ kiểm tra xem node đó có xuất hiện trong đường đi từ gốc đến node hiện tại hay chưa. Nếu có thì bỏ qua, tránh quay lại một trạng thái đã nằm trong cùng một nhánh.

Cách làm này giúp DFS tránh vòng lặp trực tiếp (quay lại cha hoặc tổ tiên gần nhất), nhưng DFS vẫn có thể duyệt nhiều nhánh khác nhau dẫn đến cùng một trạng thái (ví dụ đi vòng qua một đường khác). Do không có reached, các trạng thái lặp từ nhánh khác nhau vẫn được duyệt lại → dẫn đến nhiều node thừa, nhưng vẫn đảm bảo thuật toán không bị kẹt trong vòng lặp vô hạn.

**Task 2.3: Việc triển khai của bạn đã hoàn chỉnh và tối ưu chưa? Hãy giải thích lý do. Độ phức tạp về thời gian và không gian của từng triển khai của bạn là bao nhiêu? Đặc biệt, hãy thảo luận về sự khác biệt về độ phức tạp không gian giữa BFS và DFS.**

    BFS (Breadth-First Search):

BFS trong cài đặt của mình là hoàn chỉnh (complete) vì nếu có lời giải thì chắc chắn tìm thấy (do duyệt tuần tự theo độ sâu tăng dần).

BFS cũng là tối ưu (optimal) với chi phí đường đi bằng nhau (mỗi bước đi có chi phí = 1), vì nó luôn tìm được đường đi ngắn nhất từ start → goal.

DFS (Depth-First Search, không dùng reached, chỉ cycle checking):

DFS không hoàn chỉnh trong môi trường vô hạn hoặc đồ thị có nhiều chu trình phức tạp (nếu không có cycle checking, nó có thể lặp vô tận). Với cycle checking, DFS sẽ tránh được lặp trực tiếp nhưng vẫn có nguy cơ duyệt nhiều trạng thái trùng nhau qua nhánh khác → không đảm bảo sẽ tìm được lời giải trong mọi trường hợp.

DFS cũng không tối ưu, vì nó có thể tìm thấy một lời giải nhưng không đảm bảo là đường đi ngắn nhất.

**Task 3: Informed search: Implement greedy best-first search and A search**

Trong bài toán tìm đường đi trong mê cung (maze), bạn cần cài đặt hai thuật toán tìm kiếm có thông tin (informed search):

1. Greedy Best-First Search (GBFS)
2. *A Search*\*

Yêu cầu:

Sử dụng **khoảng cách Manhattan** làm hàm heuristic (ước lượng chi phí còn lại từ vị trí hiện tại đến đích).

Cả hai thuật toán dựa trên Best-First Search, chỉ thay đổi cách đánh giá node trong frontier.

Cần trả về:

**Path length**: độ dài đường đi tìm được (số bước).

**Path cost**: tổng chi phí (số bước đi).

**Nodes expanded**: số lượng node đã mở rộng.

**Max frontier size**: kích thước lớn nhất của frontier trong quá trình tìm kiếm.

**Giải quyết task 3:**

**Hàm heuristic (Manhattan distance, manhattan\_distance())**

Hàm này tính khoảng cách Manhattan giữa một trạng thái hiện tại và trạng thái đích. Nó ước lượng chi phí còn lại để đi đến đích, giúp thuật toán có định hướng tốt hơn thay vì mở rộng một cách mù quáng.

Hàm Manhattan phù hợp cho mê cung vì ta chỉ có thể di chuyển theo 4 hướng (lên, xuống, trái, phải), nên ước lượng này admissible (không bao giờ vượt quá chi phí thật) và consistent (thỏa mãn điều kiện tam giác).

**Greedy Best-First Search (greedy\_best\_first\_search())**

Sử dụng giá trị heuristic h(n) để mở rộng node có ước lượng gần đích nhất. Tác dụng: giúp tìm đường nhanh trong nhiều trường hợp, vì luôn “lao” về phía goal. Hạn chế: không đảm bảo tìm được đường tối ưu, vì không xét đến chi phí thực g(n).

**A\* Search (a\_star\_search())**

Kết hợp cả chi phí đã đi g(n) và ước lượng còn lại h(n): f(n)=g(n)+h(n). Tác dụng: vừa mở rộng hướng về goal (nhờ h(n)), vừa đảm bảo chọn đường ít tốn kém nhất (nhờ g(n)). Ưu điểm: nếu heuristic admissible, A\* luôn tìm ra đường đi ngắn nhất (tối ưu).

A screenshot of a computer

AI-generated content may be incorrect.

Nhận xét:

Kết quả này cho thấy: GBFS nhanh nhưng không tối ưu. A\* có thể chậm hơn chút, nhưng luôn tối ưu và đáng tin cậy hơn dù ví dụ trên path cost giống nhau.

A\* tối ưu nhưng phải lưu nodes expanded và frontier rất lớn nên tốn bộ nhớ

**Task 3.2: Việc triển khai của bạn đã hoàn thiện và tối ưu chưa? Độ phức tạp về thời gian và không gian là bao nhiêu?**

**Greedy Best-First Search (GBFS)**

Đầy đủ:

GBFS không đảm bảo đầy đủ nếu có chu trình (loop) hoặc mê cung vô hạn, vì nó chỉ chọn nút “gần goal nhất” theo heuristic mà không quan tâm đến chi phí đường đi. Trong bài toán mê cung hữu hạn và có kiểm tra visited/reached, nó đầy đủ (sẽ tìm ra đường đi nếu có).

Tối ưu:

GBFS không tối ưu vì chỉ chọn đường đi “trông có vẻ gần” goal nhất, có thể đi vòng vèo hơn so với đường đi ngắn nhất.

**A\* Search**

Đầy đủ:

Nếu heuristic admissible (không bao giờ đánh giá quá cao chi phí thật sự) thì A\* luôn đầy đủ trên không gian hữu hạn. Heuristic Manhattan trong mê cung (chỉ di chuyển 4 hướng) là admissible → đảm bảo tìm thấy lời giải nếu tồn tại.

Tối ưu:

A\* sẽ tối ưu (tìm đường đi ngắn nhất) khi heuristic admissible và consistent.

Manhattan distance vừa admissible vừa consistent → A\* của bạn tối ưu.

**Task 4: Comparison and Discussion**

Trong task này, ta chạy **bốn thuật toán**:

* **BFS (Breadth-First Search)**
* **DFS (Depth-First Search)**
* **Greedy Best-First Search (GBFS)**
* **A\* Search**

Mỗi lần chạy, ta ghi lại các thông số:

**Path Length**: độ dài đường đi tìm được. **Path Cost**: chi phí (số bước di chuyển) **Nodes Expanded**: số lượng node mở rộng. **Max Frontier**: kích thước frontier lớn nhất

Kết quả:

A black and white screen with white text

AI-generated content may be incorrect.

**Nhận xét:**

**BFS**: Luôn tìm được đường **ngắn nhất**. Nhược điểm: **bộ nhớ lớn**, frontier phình to ở maze lớn.

**DFS**: Bộ nhớ nhỏ (dùng stack). Có thể tìm ra đích nhưng không đảm bảo tối ưu. Nếu không có cycle checking thì có thể **vòng lặp vô hạn** trong maze có chu trình.

**Greedy Best-First Search (GBFS)**: Chạy rất nhanh vì chỉ dựa vào heuristic h(n). Nhưng không đảm bảo tìm ra đường đi tối ưu. Trong không gian mở, dễ “lao thẳng” về phía goal.

**A\***: Kết hợp chi phí thật g(n) và heuristic h(n).Luôn tìm được đường đi **tối ưu** (nếu h admissible). Hiệu quả nhất về cân bằng giữa tốc độ và chất lượng kết quả.

**Task 4.2: Trình bày kết quả bằng cách sử dụng biểu đồ**

Các thuật toán với độ đo là path length

A screenshot of a graph

AI-generated content may be incorrect.

Nhận xét:

BFS, GBFS, A\* đều tìm đường tối ưu nên path cost ngắn còn DFS rất dài.

A graph of different colored bars

AI-generated content may be incorrect.

Nhận xét:

BFS thường mở rộng số lượng nút nhiều nhất trong hầu hết các trường hợp. Điều này đúng vì BFS duyệt theo tầng, mở rộng toàn bộ không gian tìm kiếm cho đến khi tìm được lời giải nên rất tốn tài nguyên.

DFS số lượng nút mở rộng ít hơn BFS trong một số mê cung, nhưng có trường hợp lại khá lớn. Nguyên nhân: DFS phụ thuộc đường đi, nếu đi sai nhánh có thể mở rộng nhiều hơn trước khi quay lại.

A\* mở rộng ít hơn BFS và DFS, nhưng nhiều hơn GBFS. Vì A\* vừa dùng heuristic (như GBFS) vừa đảm bảo tìm đường tối ưu, nên phải mở rộng nhiều hơn GBFS nhưng ít hơn BFS/DFS.

GBFS ít nút mở rộng nhất vì heuristic Manhattan giúp GBFS đi thẳng về đích nhanh hơn, nhưng không đảm bảo tìm đường tối ưu.

A graph of different sizes and colors

AI-generated content may be incorrect.

Nhận xét:

DFS Có kích thước frontier lớn nhất trong nhiều trường hợp. Vì đặc tính của DFS: trong một số mê cung rộng và mở, DFS có thể lưu trữ rất nhiều nhánh trước khi tìm được đường đi.

BFS Frontier nhỏ hơn nhiều so với DFS. BFS lưu trữ theo tầng, frontier ở từng thời điểm lại không phình to bằng DFS.

GBFS Frontier ở mức trung bình (thường cao hơn BFS, thấp hơn DFS). Do GBFS chọn mở rộng nút có heuristic tốt nhất, nên frontier thường tập trung quanh đường đến đích thay vì trải rộng khắp mê cung.

A\* Frontier khá cân bằng, thường gần bằng hoặc nhỉnh hơn GBFS một chút. Lý do là A\* vừa xét heuristic (như GBFS) vừa xét cost (giống BFS), nên frontier giữ nhiều ứng viên hơn GBFS, nhưng không dài quá mức như DFS.

**Task 4.3: Thảo luận về những bài học quan trọng nhất mà bạn đã học được từ việc thực hiện các chiến lược tìm kiếm khác nhau.**

Những bài học quan trọng khi triển khai các thuật toán tìm đường khác nhau. Bài học từ các thuật toán tìm đường BFS (Breadth-First Search)

Luôn tìm được đường đi ngắn nhất về số bước nếu tất cả bước đi có cùng chi phí. Sử dụng bộ nhớ lớn khi mê cung rộng hoặc nhiều nhánh vì lưu toàn bộ các node ở cùng mức sâu.Thích hợp khi ưu tiên độ chính xác về đường đi, nhưng kém hiệu quả với maze lớn. DFS (Depth-First Search)Có thể nhanh chóng tìm được một đường đi, nhưng không đảm bảo ngắn nhất. Sử dụng bộ nhớ ít vì chỉ lưu đường đi hiện tại và các nhánh dọc theo nó.Dễ gặp vấn đề với các mê cung có vòng lặp nếu không kiểm tra các node đã thăm. GBFS (Greedy Best-First Search) Dựa vào heuristic để ưu tiên các node gần goal. Có thể rất nhanh nếu heuristic tốt, nhưng không đảm bảo tìm đường ngắn nhất. Dễ bị “kẹt” nếu heuristic sai lệch so với thực tế. A\*Kết hợp chi phí đường đi hiện tại + heuristic đến goal.Với heuristic hợp lệ, đảm bảo tìm đường tối ưu và hoàn thành.Hiệu quả cao nếu heuristic được thiết kế tốt, cân bằng giữa tốc độ, bộ nhớ và chất lượng đường đi

**Task 4.4: Advanced Task: IDS and Multiple Goals**

Trong phần này, sinh viên cần cài đặt thuật toán **Iterative Deepening Search (IDS)** dựa trên DFS giới hạn độ sâu. Sau đó, thử nghiệm IDS trên các maze đã cho và quan sát kết quả. Ngoài ra, cần phân tích một số vấn đề có thể xảy ra khi áp dụng IDS trên các maze lớn hoặc có không gian mở, giải thích nguyên nhân.

**DFS giới hạn độ sâu (dfs\_limited)**

Hàm này thực hiện tìm kiếm theo chiều sâu nhưng có **giới hạn độ sâu** (limit). Input: start, goal, maze, depth limit.Output: nếu tìm thấy đường đi thì trả về đường đi, số node mở rộng, chi phí (cost) và kích thước frontier cực đại. **Tác dụng**: Cho phép kiểm soát độ sâu tìm kiếm, từ đó có thể lồng vào IDS.

**IDS (iterative\_deepening\_search)**

Thuật toán IDS chạy DFS nhiều lần với giới hạn độ sâu tăng dần từ 0 đến max\_depth. Ở mỗi lần lặp, gọi dfs\_limited để tìm kiếm đường đi. Nếu tìm thấy goal, dừng lại và trả về kết quả. Nếu sau khi thử hết đến max\_depth mà không tìm thấy, trả về None.

**Các chỉ số thu thập được**

**Path length**: Độ dài đường đi tìm được (số bước). **Path cost**: Chi phí của lời giải (ở đây chính là độ sâu). **Nodes expanded**: Số lượng node được mở rộng trong toàn bộ quá trình. **Max Frontier**: Số lượng node nhiều nhất trên frontier tại một thời điểm.

Kết quả chạy:

A black rectangle with white text

AI-generated content may be incorrect.

Nhận xét:

IDS mở rộng **rất nhiều nút** so với các thuật toán khác (như GBFS, A\*).Con số này tăng mạnh theo độ phức tạp và kích thước mê cung: small\_maze: 1394 nodes. medium\_maze: 15699 nodes. large\_maze: 6980 nodes. open\_maze: tới 24072 nodes. Điều này phản ánh nhược điểm chính của IDS: phải lặp lại nhiều lần quá trình DFS với độ sâu tăng dần, gây lãng phí tài nguyên.

Giá trị Max Frontier của IDS **khá thấp** so với BFS/DFS trong biểu đồ trước. small\_maze: 26, medium\_maze: 91. open\_maze: 105. Điều này chứng minh IDS có ưu điểm về **tiết kiệm bộ nhớ**, vì frontier chỉ lưu đường đi hiện tại chứ không phải toàn bộ tầng (như BFS).

**Task 4.5: More Advanced Problems: Intersection as States**

**Yêu cầu:**

Trong phiên bản trước, mỗi **ô vuông (square)** trong maze được coi là một trạng thái (state). Điều này dẫn đến số lượng trạng thái lớn, đặc biệt khi maze có kích thước lớn hoặc nhiều ô trống.

Bài toán yêu cầu :

**Chỉ coi các giao lộ (intersections)** là trạng thái.Khi đó, số trạng thái được giảm xuống đáng kể, vì chỉ còn các điểm rẽ, điểm bắt đầu (S), và điểm kết thúc (G). Tuy nhiên, **độ dài đường đi giữa hai giao lộ** có thể khác nhau (ví dụ một cạnh có thể dài 3 ô, cạnh khác dài 7 ô). Cần đảm bảo rằng **BFS và IDS vẫn tối ưu** nếu ta đo chi phí đường đi bằng số ô (path cost).

**2.1. Xác định giao lộ**

Một ô được coi là giao lộ nếu nó có ít nhất 2 hướng đi hợp lệ.

Hàm is\_intersection kiểm tra điều này.

Hàm find\_intersections tìm toàn bộ các giao lộ trong maze, đồng thời luôn thêm **S** và **G** để chắc chắn có trong graph.

**2.2. Xây dựng graph giao lộ**

Dùng BFS (bfs\_path) để tìm đường đi ngắn nhất từ một giao lộ đến tất cả giao lộ khác.

Mỗi cạnh trong graph lưu:

**target** (giao lộ đích)

**edge\_cost** = số ô vuông (squares) giữa 2 giao lộ

**path** = chuỗi các ô thực tế giữa 2 giao lộ

Hàm build\_intersection\_graph tạo graph dạng adjacency list.

**2.3. BFS trên graph**

Thực hiện BFS nhưng ở mức **giao lộ** thay vì từng ô.

Lưu ý: cost tính bằng tổng số ô trên các cạnh đi qua.

Đảm bảo tìm được đường đi tối ưu (shortest path).

**2.4. IDS trên graph**

Cài đặt dfs\_limited\_graph và ids\_on\_graph.

DFS giới hạn độ sâu duyệt trên graph giao lộ, với giới hạn theo số cạnh (edge depth).

Chi phí (cost) vẫn được cộng dồn chính xác bằng số ô vuông.

IDS chạy lặp với độ sâu tăng dần, đảm bảo tìm ra lời giải tối ưu về cost.

Kết quả chạy:

A screenshot of a computer

AI-generated content may be incorrect.

Nhận xét:

Số lượng intersections trong các maze càng lớn/phức tạp thì số intersections càng nhiều nên ảnh hưởng trực tiếp đến số node phải expand của BFS, DFS, IDS. Path Length = 2 là do cách biểu diễn trong bảng này (chỉ tính số intersections đi qua, không tính số bước thật sự). Số intersections cho thấy độ phức tạp của maze: càng nhiều intersection nên thuật toán càng phải mở rộng nhiều nodes (nhất là IDS)

**Task 4.6: Weighted A\* Search**

**Yêu cầu:**

Trong phần cơ bản, A\* sử dụng công thức:

f(n)=g(n)+h(n)

g(n): chi phí đường đi từ start đến node hiện tại.

h(n) heuristic (ước lượng chi phí từ node hiện tại đến goal).

Trong **Weighted A\***, công thức được sửa thành: f(n)=g(n)+w⋅h(n)

**Giải quyết yêu cầu này:**

weighted\_a\_star(maze, start, goal, weight) được xây dựng như sau:

Hàm heuristic: manhattan(a, b) tính khoảng cách Manhattan.

Cấu trúc dữ liệu:

Priority queue (heapq) để quản lý frontier theo giá trị f. Từ điển visited để lưu giá trị ggg tốt nhất từng node đã thăm.

Mỗi bước lặp:

Lấy node có f nhỏ nhất từ heap.

Nếu node = goal → trả về đường đi và chi phí ggg.

Nếu node đã được thăm với ggg tốt hơn → bỏ qua.

Sinh các node kề, tính g, f, rồi thêm vào heap.

Kết quả chạy với small maze:

A black background with white text

AI-generated content may be incorrect.

Nhận xét:

Trongkết quả này **A**\* cho kết quả giống nhau ở mọi trọng số (đường đi tối ưu, cost = 15). Tuy nhiên, nếu maze phức tạp hơn, **w > 1** có thể làm giảm số node mở rộng nhưng đường đi có nguy cơ không tối ưu.

**Task 4.7: Unknown Maze**

**Bài toán đặt ra**

Trong các phần trước, ta giả sử agent biết toàn bộ layout của mê cung (toàn bộ bản đồ và vị trí tường). Tuy nhiên, thực tế có thể khác: **agent không biết trước môi trường** mà chỉ quan sát được những ô xung quanh khi di chuyển. Điều này khiến agent không thể định nghĩa chính xác transition model từ đầu.

**Mô hình hóa môi trường (PEAS)**

**Performance measure (P):**

Thời gian để đến đích. Độ dài đường đi. Hiệu quả khám phá (không đi vòng quá nhiều).

**Environment (E):**

Maze không biết trước (unknown). Agent chỉ quan sát được các ô kề cạnh khi đang ở một ô. Có tường (#), đường trống ( ). Start (S) và Goal (G).

**Actuators (A):**

Di chuyển lên, xuống, trái, phải.

**Sensors (S):**

Quan sát được các ô liền kề. Nếu có thêm GPS thì sensor cung cấp cả khoảng cách Manhattan tới Goal.

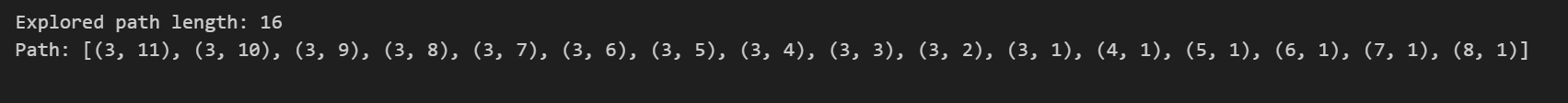
**3. Rational agent trong maze unknown**

Agent duy trì một **bản đồ tạm thời (map\_known)**, ban đầu toàn bộ là . Khi di chuyển, agent quan sát xung quanh và cập nhật bản đồ này. Sau mỗi lần cập nhật, agent **chạy Weighted A**\* trên map\_known để tìm đường tạm thời tới Goal. Agent đi một bước theo đường tìm được → lại quan sát → lại cập nhật bản đồ. Quá trình lặp lại cho đến khi đến Goal hoặc xác định Goal không thể đạt được. Code mô phỏng: weighted\_a\_star\_unknown: tìm đường ngắn nhất trên bản đồ mà agent **hiện biết**. explore\_unknown\_maze: agent vừa di chuyển, vừa quan sát, vừa cập nhật bản đồ.

**4. Khi có GPS**

Nếu agent có thêm GPS (trả về khoảng cách Manhattan tới Goal): Agent vẫn không biết trước toàn bộ bản đồ, nhưng có thêm thông tin định hướng tốt hơn. GPS giúp agent biết được mình đang tiến gần hay xa Goal, nhờ đó heuristic h(n) chính xác hơn. Khi đó, Weighted A\* sẽ càng hiệu quả vì heuristic đáng tin cậy hơn → giảm số bước thừa.

Kết quả chạy:



Nhận xét:

Đường đi cho thấy một chiến lược khá trực tiếp: đầu tiên di chuyển dọc theo cột 3 từ hàng 11 xuống hàng 1, sau đó chuyển sang di chuyển ngang từ cột 3 đến cột 8 dọc theo hàng 1. Điều này gợi ý rằng agent có thể đang sử dụng thông tin heuristic (khoảng cách đến goal) để hướng dẫn tìm kiếm.