TRƯỜNG ĐẠI HỌC SÀI GÒN
KHOA CÔNG NGHỆ THÔNG TIN

Báo cáo môn: Trí tuệ nhân tạo nâng

Bài tập tuần Lab02

Giảng viên hướng dẫn: Đỗ Như Tài

Sinh viên thực hiện

Nguyễn Thành Long
MSSV: 3121410297

Huỳnh Hoàng Huy
MSSV: 3121410226

Nguyễn Trần Uyển Nhi
MSSV: 3122410281

Phan Thanh Thịnh
MSSV: 3123410360

Tháng 05 - 2025

Mục lục

Thành viên	**Nhiệm vụ được giao**	**Đánh giá của nhóm trưởng**
Nguyễn Thành Long	Phát triển *Lunar Lander*, viết báo cáo, thiết kế slide thuyết minh.	Hoàn thành 100%
Phan Thanh Thịnh	Phát triển *Maze*, viết báo cáo, thiết kế slide thuyết minh.	Hoàn thành 100%
Nguyễn Trần Uyển Nhi	Phát triển *Lunar Lander*, biên tập báo cáo tổng bằng LaTeX.	Hoàn thành 100%
Huỳnh Hoàng Huy	Phát triển *Robot Vacuum*, viết báo cáo.	Hoàn thành 100%

Bảng 1: Bảng phân công công việc của nhóm

I Agents

1 Lunar Lander

1.1 Mục tiêu

Phát triển một Reflex Base Agent tốt hơn phiên bản có trong bài tập.

1.2 Cách thức thực hiện

Chúng ta sẽ tạo ra một danh sách các luật để agent thực hiện theo thứ tự từ trên xuống dưới. Lúc ban đầu, chỉ khi rơi nhanh thì mới kích hoạt động cơ đẩy chính để hãm tốc, rất ngây thơ, nhược điểm của luật này là quá lỏng lẻo và thiếu các yếu tố điều hướng để mô hình có thể tìm về bãi đáp. Do đó tôi đã triển khai thêm các luật hoàn thiện với thứ tự thực hiện ưu tiên như sau, bao gồm 3 giai đoạn chính: Hãm tốc, Canh góc, và cuối cùng là Đưa tàu về điểm đáp. Giai đoạn Hãm tốc là khi tàu rơi quá nhanh, ta sẽ bật động cơ chính để hãm tốc, gia tốc chiều thẳng đứng $<$ -0.2. Giai đoạn Canh góc, khi tàu đáp hoặc bật động cơ chính, tàu có thể không rơi thẳng mà rơi ngang, vecto lực di chuyển theo phương xiên, điều này khiến tàu dễ xiên vẹo và đâm đất nên luật này giúp tàu lúc nào cũng trong trạng thái gần thẳng đứng, khi mũi tàu nghiêng sang trái thì ta dùng động cơ trái đẩy cho tàu thẳng lại và khi mũi tàu nghiêng sang phải thì ngược lại. Ở giai đoạn cuối, sau khi tốc độ ổn, góc gần thẳng đứng, lúc này nhiệm vụ cuối cùng là giúp cho tàu về đích, vì sao lại về đích, về đích giúp điểm cao hơn và lúc nào vạch đích cũng là nơi đáp an toàn nhất, vì thế nên tôi đã cho thêm luật nếu tàu ở vị trí X cách vạch đích thì khởi động động cơ trái hoặc phải để đẩy tàu về gần đích.

Dây là phần luật chính của Agent:

```
    rules = [
        # Dieu kien, hanh dong
        # Ham toc do roi
        (lambda obs: obs[Obs.VY.value] < -0.2, Act.MAIN.value),

        # Canh goc
        (lambda obs: obs[Obs.ANGLE.value] > 0.15, Act.RIGHT.value),
        # Mui tau nghieng phai nen dung dong co phai tra day ve.
        (lambda obs: obs[Obs.ANGLE.value] < -0.15, Act.LEFT.value),
        # Mui tau nghieng trai nen dung dong co trai tra day ve.

        # Dua tau ve bai dap chi dinh
        (lambda obs: obs[Obs.X.value] > 0.1, Act.LEFT.value),
        (lambda obs: obs[Obs.X.value] < -0.1, Act.RIGHT.value)
    ]
```

1.3 Kết quả

Agent đã đạt 60/100 lần đáp thành công với số điểm trung bình là 20 điểm. Tốt hơn mô hình ban đầu tới 75 lần về số lần đáp thành công.

1.4 Hạn chế

Hạn chế chính của agent nằm ở khả năng xử lý địa hình không bằng phẳng, phối hợp động cơ khi nghiêng, và quản lý tình huống khẩn cấp gần mặt đất. Dây là những điểm

cần cải thiện trong các phiên bản tiếp theo, chẳng hạn bằng cách kết hợp thêm logic kiểm tra độ cao, gia tốc tổng hợp, hoặc chiến lược điều khiển dự đoán thay vì chỉ phản xạ theo luật.

2 Robot Vacuum

2.1 Mục tiêu

Xây dựng một agent mô phỏng robot hút bụi tự động trong môi trường lưới 2D. Agent cần có khả năng cảm nhận ô hiện tại (sạch hay bẩn), di chuyển trong không gian giới hạn, và thực hiện hành vi làm sạch theo quy tắc hoặc chiến lược nhất định.

2.2 Cách thức thực hiện

Bài được chia làm nhiều phần (tasks):

1. **Tạo môi trường mô phỏng (Task 1)**

   - Xây dựng lưới kích thước $m \times n$, mỗi ô có xác suất bẩn `dirt_prob`.
   - Robot di chuyển trên lưới, cập nhật vị trí và trạng thái từng ô.

2. **Xây dựng Simple Reflex Agent (Task 2)**

   - Áp dụng luật đơn giản:
     - Nếu ô hiện tại bẩn → CLEAN
     - Nếu sạch → di chuyển ngẫu nhiên
   - Không lưu trạng thái trước đó (stateless).

3. **Model-Based Reflex Agent (Task 3)**

   - Thêm bộ nhớ (`visited`, `grid_state`) để lưu các ô đã làm sạch.
   - Di chuyển đến các ô chưa được làm sạch → bao quát tốt hơn.

4. **Simulation Study (Task 4)**

   - Chạy thử cả ba loại agent: Random, Simple Reflex, Model-Based.
   - So sánh hiệu suất trung bình (performance score) trên nhiều kích thước phòng (5x5, 10x10, 100x100).

5. **Phân tích độ ổn định (Task 5)**

   - Kiểm tra hoạt động khi cảm biến hoặc môi trường không hoàn hảo (bẩn sai 10%, vật cản, ...).
   - Model-based agent vẫn giữ hiệu quả tốt hơn do có khả năng ghi nhớ và thích nghi.

2.3 Kết quả

- Simple Reflex Agent: hoạt động tốt trong môi trường nhỏ, nhưng thường bỏ sót ô và lặp lại đường đi.
- Model-Based Agent: bao quát môi trường tốt hơn, đạt hiệu suất cao nhất trong thử nghiệm.
- Khi mô phỏng lỗi cảm biến 10%, hiệu suất giảm nhẹ (5–10%), nhưng agent vẫn hoàn thành nhiệm vụ.

2.4 Hạn chế

- Agent chưa tối ưu đường đi (chưa có lập kế hoạch tìm đường toàn cục).
- Không xử lý được chướng ngại vật phức tạp.
- Có thể cải thiện bằng cách kết hợp thuật toán tìm đường (A*) hoặc reinforcement learning để tối ưu quyết định.

Tổng kết:
Robot Vacuum Agent là ví dụ điển hình của hệ thống AI phản xạ (reflex-based agent). Qua bài này, ta hiểu rõ quy trình thiết kế, đánh giá và cải tiến agent từ đơn giản đến thông minh hơn.

II Search

1 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.

1.1 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: $S = (r_s, c_s)$. Hàm `find_positions()` trả về S.

1.2 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).

1.3 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: $\text{Result}(s, a) = s'$. Với ràng buộc: $arr[r', c'] \neq \#$ và $0 \leq r' \leq rows,\ 0 \leq c' \leq cols$. Được hiện thực bởi `valid_moves()`.

1.4 Goal State (Trạng thái đích)

Trạng thái đích là tọa độ G trong mê cung. Bài toán kết thúc khi agent đạt tới $G = (r_g, c_g)$. Hàm `find_positions()` trả về G.

1.5 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 G. Đây 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.

	file	rows	cols	S	G	N_free	b_avg	manhattan	shortest_length_BFS
0	small_maze.txt	10	22	(3, 11)	(8, 1)	220	3.709	15	15
1	medium_maze.txt	18	36	(1, 34)	(16, 1)	648	3.833	48	48
2	large_maze.txt	37	37	(35, 35)	(35, 1)	1369	3.892	34	34
3	open_maze.txt	23	37	(1, 35)	(21, 1)	851	3.859	54	54
4	loops_maze.txt	12	12	(10, 1)	(1, 1)	144	3.667	9	9
5	empty_maze.txt	12	12	(9, 2)	(2, 9)	144	3.667	14	14
6	empty_maze_2.txt	12	12	(2, 9)	(9, 2)	144	3.667	14	14

2 Task 1.2

Cho một mê cung, ta có thể 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).

Cách xác định các giá trị này cho một mê cung cụ thể:

2.1 Số trạng thái n

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

$$n = |\{(r, c) \mid arr[r, c] \neq \#\}|$$

Hàm `count_free(arr)` trả về số trạng thái tương ứng với số ô trống trong mê cung.

2.2 Độ sâu nghiệm tối ưu d

Đây là số bước đi ngắn nhất từ trạng thái xuất phát S đến trạng thái đích G. Xác định bằng cách chạy BFS:

$$d = shortest\_path\_length(S, G)$$

Hàm `bfs_shortest(arr, S, G)` trả về số bước đi ngắn nhất.

2.3 Độ sâu cực đại m

Độ sâu cực đại của cây trạng thái bị chặn bởi tổng số trạng thái khả dĩ trong mê cung. Một upper bound:

$$m \leq n - 1$$

Vì không có đường đi nào dài hơn số trạng thái có thể đi qua. Ta có thể gán $m_{upper} = n-1$.

2.4 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:

$$b = \max_{s \in StateSpace} |Actions(s)|$$

Hàm `compute_b_avg_and_bmax(arr)` trả về b_{max}.

Bảng kết quả giá trị từ các mê cung:

	Maze	Size	n (states)	d (opt depth)	m (max depth)	b_avg	b_max
0	small_maze.txt	(10, 22)	220	15	219	3.71	4
1	medium_maze.txt	(18, 36)	648	48	647	3.83	4
2	large_maze.txt	(37, 37)	1369	34	1368	3.89	4
3	open_maze.txt	(23, 37)	851	54	850	3.86	4
4	loops_maze.txt	(12, 12)	144	9	143	3.67	4
5	empty_maze.txt	(12, 12)	144	14	143	3.67	4
6	empty_maze_2.txt	(12, 12)	144	14	143	3.67	4

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.

3 Task 2.1

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.

3.1 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 $\rightarrow$ 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.

3.2 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 tăng 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.

3.3 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:

```
=== Maze: small_maze.txt ===
BFS:
  Path length      = 15
  Path cost        = 15
  Nodes expanded   = 207
  Max depth        = 14
  Max frontier size= 20
  Runtime (s)      = 0.001
DFS:
  Path length      = 119
  Path cost        = 119
  Nodes expanded   = 120
  Max depth        = 119
  Max frontier size= 138
  Runtime (s)      = 0.001

=== Maze: medium_maze.txt ===
BFS:
  Path length      = 48
  Path cost        = 48
  Nodes expanded   = 640
  Max depth        = 47
  Max frontier size= 20
  Runtime (s)      = 0.003
```

Nhận xét: Path cost của DFS rất lớn so với BFS. Vì 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, trong khi 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 độ sâu ≈ 14), nên tổng số node được mở rộng có thể lớn hơn. DFS đi sâu nhanh hơn, nên trong trường hợp này nó đã tìm được một đường (không tối ưu) mà không cần mở quá nhiều node.

4 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.

5 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.

5.1 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.

5.2 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.

6 Task 3.1

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 3.1:

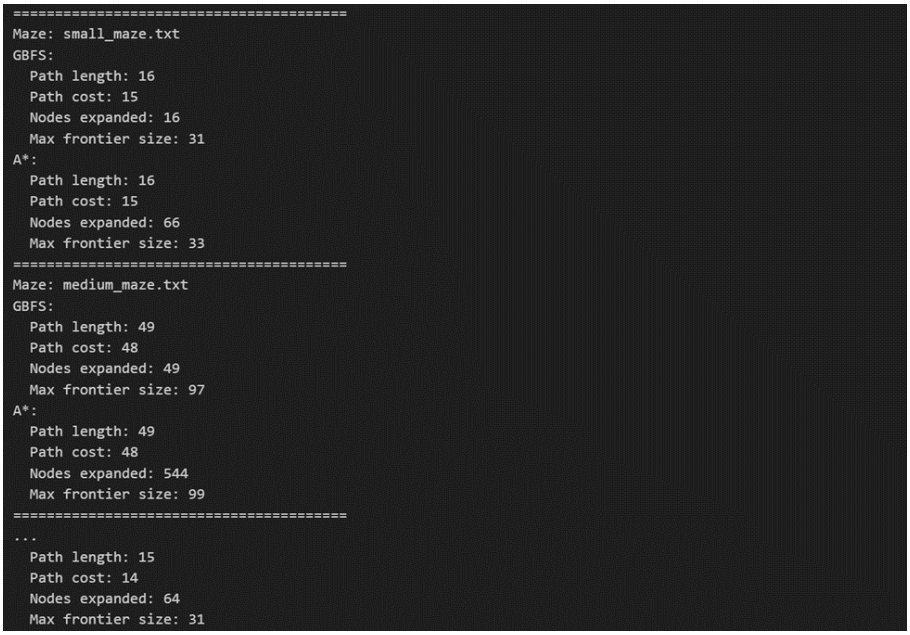
Hàm heuristic (Manhattan distance, `manhattan_distance()`)
Hàm này tính khoảng cách Manhattan giữa một trạng thái hiện tại và trạng thái đích. Nó ước lượng chi phí còn lại để đi đến đích, giúp thuật toán có định hướng tốt hơn thay vì mở rộng một cách mù quáng. Hàm Manhattan phù hợp cho mê cung vì ta 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 he



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.

7 Task 3.2

Đánh giá việc triển khai và phân tích độ phức tạp:

1. Greedy Best-First Search (GBFS)

Đầy đủ (Completeness):
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. Tuy nhiên, trong bài toán mê cung hữu hạn và có kiểm tra `visited/reached`, thuật toán sẽ đầy đủ (tức là sẽ tìm ra đường đi nếu tồn tại).

Tối ưu (Optimality):
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.

Độ phức tạp:

- **Thời gian:** $O(b^d)$ trong trường hợp xấu nhất, vì vẫn có thể phải mở rộng tất cả các node tới độ sâu nghiệm d (phụ thuộc vào chất lượng của heuristic).
- **Không gian:** $O(b^d)$, do phải lưu trữ các node trong `frontier` và `reached`.

2. A* Search

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

Tối ưu (Optimality):
A* sẽ tối ưu (tức là tìm được đường đi ngắn nhất) khi heuristic vừa *admissible* vừa *consistent*. Khoảng cách Manhattan thỏa mãn cả hai điều kiện này, nên A* trong bài toán mê cung là tối ưu.

Độ phức tạp:

- **Thời gian:** Phụ thuộc vào chất lượng heuristic. Trong trường hợp xấu nhất (heuristic kém, gần như bằng 0), A* tương đương với BFS nên có độ phức tạp $O(b^d)$.
- **Không gian:** Cũng là $O(b^d)$, vì A* cần lưu toàn bộ frontier (open list) và tập các node đã mở rộng (closed list).

8 Task 4.1

Thực nghiệm bốn thuật toán tìm kiếm:

Trong task này, ta chạy bốn thuật toán sau:

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

Mỗi lần chạy, ta ghi lại các thông số sau:

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

Kết quả:

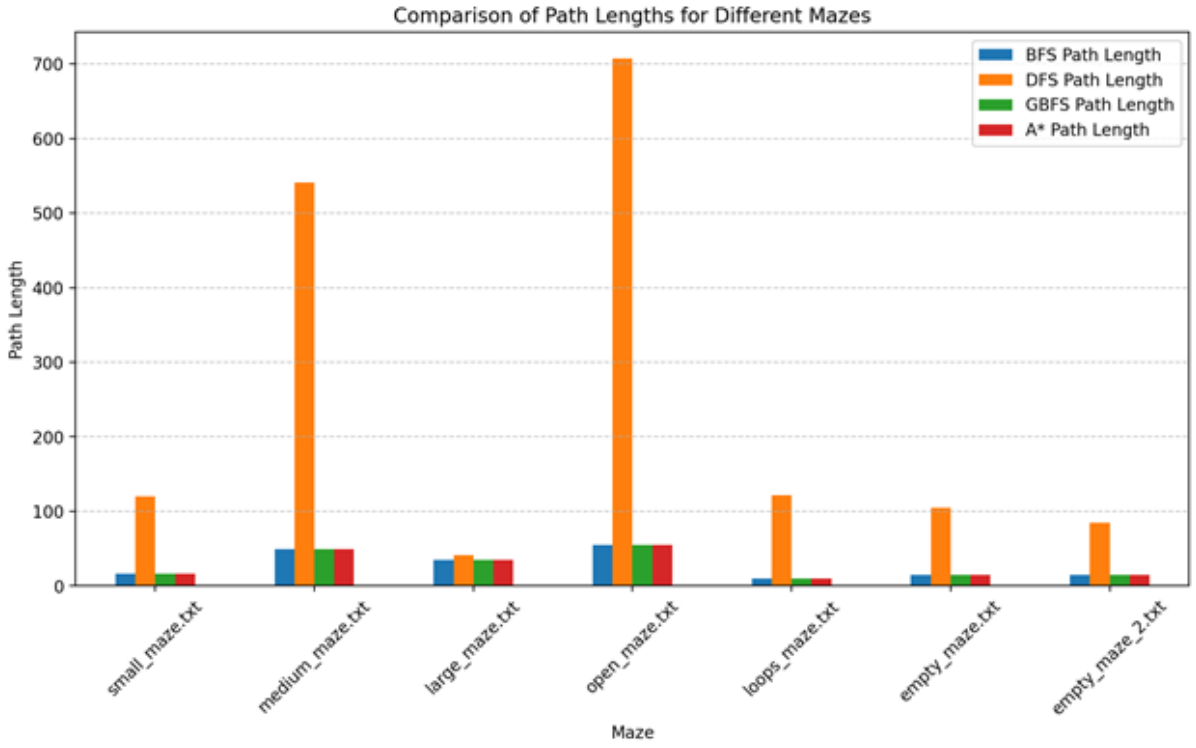
Maze: empty_maze_2.txt

	Maze	BFS Path Length	BFS Path Cost	BFS Nodes Expanded	BFS Max Frontier	DFS Path Length	DFS Path Cost	DFS Nodes Expanded	DFS Max Frontier	GBFS Path Length	GBFS Path Cost	GBFS Nodes Expanded	GBFS Max Frontier	A* Path Length	A* Path Cost	A* Nodes Expanded	A* Max Frontier
0	small_maze.txt	16	14	207	20	120	119	120	138	16	15	16	31	16	15	66	33
1	medium_maze.txt	49	47	640	20	541	540	541	563	49	48	49	97	49	48	544	99
2	large_maze.txt	35	33	662	38	41	40	41	40	35	34	35	70	35	34	35	70
3	open_maze.txt	55	53	843	25	707	706	707	725	55	54	55	109	55	54	735	111
4	loops_maze.txt	10	8	52	12	122	121	122	112	10	9	10	20	10	9	10	20
5	empty_maze.txt	15	13	126	15	105	104	118	109	15	14	15	29	15	14	64	31
6	empty_maze_2.txt	15	13	126	15	85	84	85	91	15	14	15	29	15	14	64	31

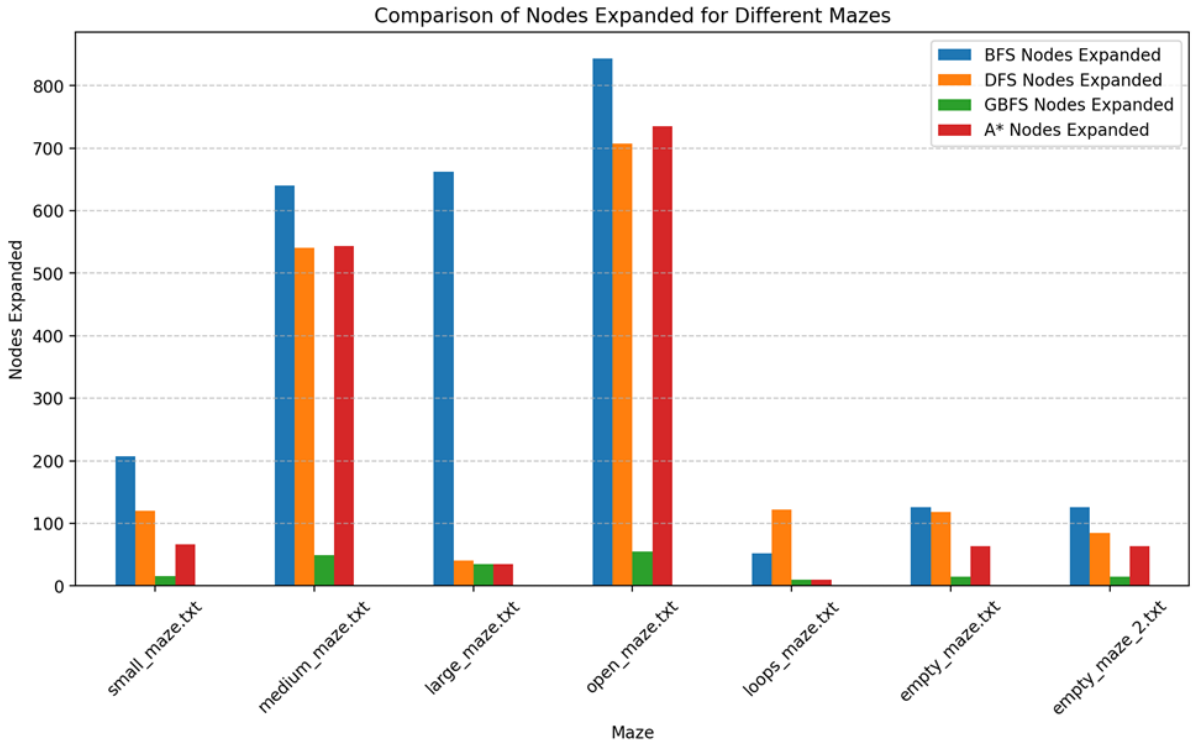
Nhận xét:
– BFS: Luôn tìm được đường ngắn nhất. Nhược điểm: bộ nhớ lớn, frontier phình to ở maze lớn.
– DFS: Bộ nhớ nhỏ (dùng stack). Có thể tìm ra đích nhưng không đảm bảo tối ưu. Nếu không có cycle checking thì có thể vòng lặp vô hạn trong maze có chu trình.
– Greedy Best-First Search (GBFS): Chạy 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.

9 Task 4.2

Các thuật toán với độ đo là path length



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



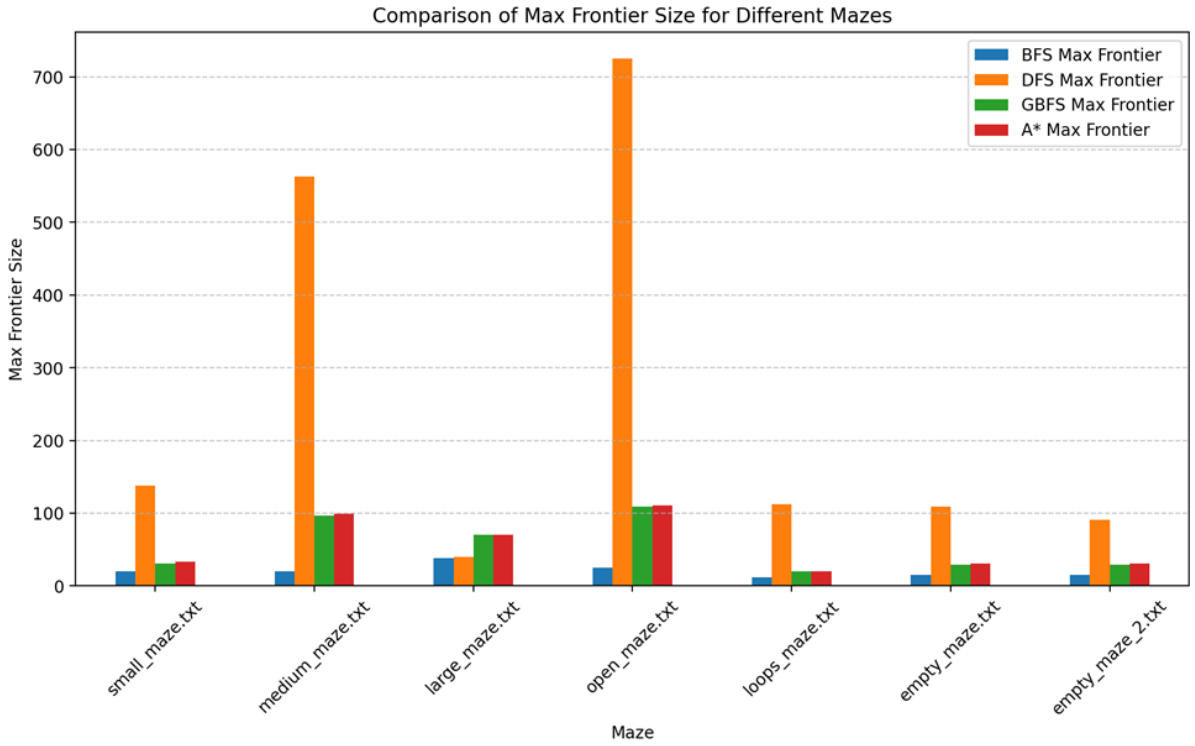
Nhận xét:

– 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.



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.

10 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

11 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 (Depth-Limited Search). Sau đó, tiến hành 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, đồng thời giải thích nguyên nhân.

DFS giới hạn độ sâu (`dfs_limited`):
Thuật toán 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:** Giúp kiểm soát độ sâu tìm kiếm, từ đó có thể lồng vào IDS để tránh lặp vô hạn.

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 vòng lặp, gọi `dfs_limited()` để tìm kiếm.
- Nếu tìm thấy `goal`, dừng lại và trả về kết quả.
- Nếu sau khi thử 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 (độ sâu của node đích).
- **Nodes Expanded:** Tổng số lượng node được mở rộng trong toàn bộ quá trình tìm kiếm.
- **Max Frontier:** Kích thước lớn nhất của `frontier` tại một thời điểm.

Kết quả:

```
Maze: small_maze.txt
IDS Path length: 16
IDS Path cost: 15
IDS Nodes expanded: 1394
IDS Max Frontier: 26
Maze: medium_maze.txt
IDS Path length: 57
IDS Path cost: 56
IDS Nodes expanded: 15699
IDS Max Frontier: 91
Maze: large_maze.txt
IDS Path length: 41
IDS Path cost: 40
IDS Nodes expanded: 6980
IDS Max Frontier: 75
Maze: open_maze.txt
IDS Path length: 71
IDS Path cost: 70
IDS Nodes expanded: 24072
IDS Max Frontier: 105
Maze: loops_maze.txt
IDS Path length: 18
IDS Path cost: 17
IDS Nodes expanded: 1184
IDS Max Frontier: 29
...
```

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).

12 Task 4.5

More Advanced Problems: Intersection as States

Trong phiên bản trước, mỗi ô vuông (`square`) trong mê cung được coi là một trạng thái (`state`). Cách biểu diễn này dẫn đến số lượng trạng thái rất lớn khi mê cung có kích thước lớn hoặc nhiều ô trống. Trong phiên bản này, ta chỉ coi **các giao lộ (intersections)** là trạng thái. Điều này giúp giảm đáng kể số lượng trạng thái, vì chỉ còn lại 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). Do đó, cần đảm bảo rằng **BFS** và **IDS** vẫn tối ưu nếu ta đo chi phí đường đi theo số ô (`path cost`).

12.1 Xác định giao lộ

Một ô được coi là giao lộ nếu nó có ít nhất hai hướng đi hợp lệ. Hàm `is_intersection()` dùng để kiểm tra điều này. Hàm `find_intersections()` duyệt toàn bộ mê cung để tìm các giao lộ, đồng thời luôn thêm `S` và `G` vào danh sách để đảm bảo chúng có trong đồ thị.

12.2 Xây dựng graph giao lộ

Từ tập giao lộ, ta xây dựng đồ thị theo dạng **adjacency list**. Để tìm cạnh giữa hai giao lộ, sử dụng **BFS** (`bfs_path()`) để tìm đường ngắn nhất giữa chúng trên bản đồ gốc. Mỗi cạnh trong graph lưu trữ thông tin:

- **target:** giao lộ đích.
- **edge_cost:** số lượng ô vuông (`squares`) giữa hai giao lộ.
- **path:** chuỗi các ô vuông thực tế tạo thành cạnh đó.

Hàm `build_intersection_graph()` thực hiện việc xây dựng đồ thị này.

12.3 BFS trên graph giao lộ

Sau khi có đồ thị các giao lộ, ta thực hiện **BFS** nhưng ở mức `intersections` thay vì từng ô. Chi phí (`cost`) được tính bằng tổng số ô đi qua trên các cạnh của đồ thị. Thuật toán `bfs_on_graph()` đảm bảo tìm được đường đi có tổng chi phí nhỏ nhất, vì BFS mở rộng theo độ sâu tăng dần (và chi phí cạnh dương).

12.4 IDS trên graph giao lộ

Tương tự, ta cài đặt **DFS giới hạn độ sâu** trên graph giao lộ (`dfs_limited_graph()`), với giới hạn theo số cạnh (`edge depth`). Chi phí (`cost`) vẫn được cộng dồn chính xác bằng số ô vuông thực tế. Thuật toán `ids_on_graph()` thực hiện lặp DFS nhiều lần với giới hạn độ sâu tăng dần, và dừng lại khi tìm thấy lời giải.

Nhờ cách đo chi phí theo số ô, **IDS trên graph giao lộ vẫn đảm bảo tìm ra đường đi tối ưu về cost**, tương tự như BFS, nhưng với không gian trạng thái nhỏ hơn đáng kể.
Kết quả:

```
small_maze.txt: 220 intersections found
medium_maze.txt: 648 intersections found
large_maze.txt: 1369 intersections found
open_maze.txt: 851 intersections found
loops_maze.txt: 144 intersections found
empty_maze.txt: 144 intersections found
empty_maze_2.txt: 144 intersections found
                Maze  BFS Path Length  BFS Path Cost  IDS Path Length  \
0    small_maze.txt                2             15                2
1   medium_maze.txt                2             48                2
2    large_maze.txt                2             34                2
3     open_maze.txt                2             54                2
4    loops_maze.txt                2              9                2
5    empty_maze.txt                2             14                2
6  empty_maze_2.txt                2             14                2

   IDS Path Cost
0             15
1             48
2             34
3             54
4              9
5             14
6             14
```

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)

13 Task 4.6

Weighted A* Search:

Trong phần cơ bản, thuật toán **A*** sử dụng công thức đánh giá:

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

Trong đó:

- $g(n)$: chi phí thực từ trạng thái `start` đến node hiện tại.
- $h(n)$: giá trị heuristic (ước lượng chi phí còn lại từ node hiện tại đến `goal`).

Trong **Weighted A***, công thức được điều chỉnh nhằm tăng tốc độ hội tụ:

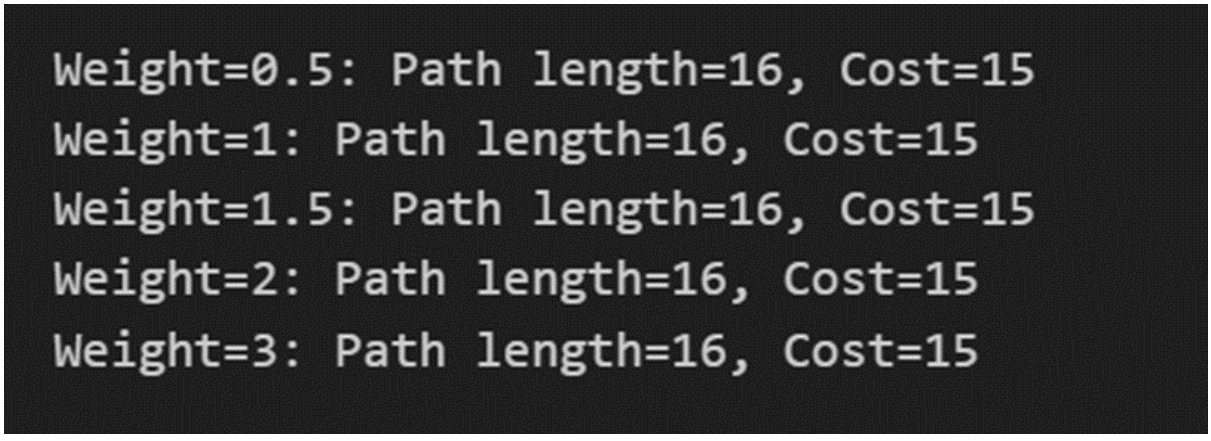
$$f(n) = g(n) + w \cdot h(n)$$

với $w > 1$ là hệ số trọng số (`weight`) quy định mức độ "ưu tiên heuristic". Khi $w = 1$, Weighted A* trở thành A* thông thường. Khi $w > 1$, thuật toán thiên về hướng `goal` hơn, hy sinh tính tối ưu để đổi lấy tốc độ tìm kiếm nhanh hơn.

Cài đặt hàm `weighted_a_star(maze, start, goal, weight)`:

- **Hàm heuristic:** `manhattan(a, b)` tính khoảng cách Manhattan giữa hai tọa độ (x_1, y_1) và (x_2, y_2):
$$h(a, b) = |x_1 - x_2| + |y_1 - y_2|$$
Đây là heuristic admissible và consistent trong mê cung di chuyển 4 hướng.
- **Cấu trúc dữ liệu:** Sử dụng **priority queue** (module `heapq`) để quản lý `frontier` theo giá trị f. Dùng từ điển `visited` để lưu giá trị g nhỏ nhất (tốt nhất) mà mỗi node đã đạt được.
- **Thuật toán:**
  1. Khởi tạo heap với trạng thái bắt đầu, $g = 0$, $f = w \cdot h(start, goal)$.
  2. Trong mỗi vòng lặp:
     - Lấy node có f nhỏ nhất từ heap.
     - Nếu node là `goal` $\Rightarrow$ trả về đường đi và chi phí g.
     - Nếu node đã được thăm với giá trị g tốt hơn trước đó $\Rightarrow$ bỏ qua.
     - Sinh các node kề hợp lệ, tính:
$$g' = g + 1, \quad f' = g' + w \cdot h(neighbor, goal)$$
và thêm vào heap.

Kết quả:



Nhận xét:
Trong kết 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.

14 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.

Rational agent trong maze unknown

Agent duy trì một bản đồ tạm thời (`map_known`), ban đầu toàn bộ là dấu .. 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.

`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.

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:

```
Explored path length: 16
Path: [(3, 11), (3, 10), (3, 9), (3, 8), (3, 7), (3, 6), (3, 5), (3, 4), (3, 3), (3, 2), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1)]
```

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.