Maze

**Task 1: Defining the search problem and determining the problem size**

**Mục tiêu:**

Trong task này, yêu cầu chính là xác định và mô tả các thành phần cơ bản của một bài toán tìm kiếm trong môi trường mê cung, bao gồm: trạng thái ban đầu, tập hợp hành động, mô hình chuyển trạng thái, trạng thái mục tiêu và hàm chi phí đường đi. Việc định nghĩa đúng những yếu tố này là tiền đề để có thể áp dụng các thuật toán tìm kiếm như BFS, DFS, A\*, GBFS… nhằm giải quyết bài toán.

**Cách giải quyết:**

**Trạng thái ban đầu (Initial state):** Đây là vị trí xuất phát của agent trong mê cung, thường được biểu diễn dưới dạng tọa độ 2 chiều:

Start = (xstart​,ystart​)

Đây là điểm khởi đầu mà từ đó agent phải lên kế hoạch di chuyển để tìm đến đích.

**Các hành động (Actions**): Tại mỗi trạng thái, agent có thể di chuyển theo bốn hướng cơ bản

Actions = {UP,DOWN,LEFT,RIGHT}

Tuy nhiên, một số hành động có thể không hợp lệ nếu dẫn đến việc đi ra ngoài phạm vi mê cung hoặc va vào tường.

**Mô hình chuyển trạng thái (Transition model):** Mô hình này xác định trạng thái mới sau khi agent thực hiện một hành động từ trạng thái hiện tại. Nếu agent ở trạng thái sss và thực hiện hành động a, nó sẽ đến trạng thái s′.

Cụ thể: **s′=s+Δ**

Trong đó Δ phụ thuộc vào hành động:

* UP → (−1,0)
* DOWN → (1,0)
* LEFT → (0,−1)
* RIGHT → (0,1)

**Trạng thái mục tiêu (Goal state):** Đây là vị trí đích mà agent cần tìm đến trong mê cung, cũng được biểu diễn dưới dạng tọa độ 2 chiều

Goal = (xgoal​,ygoal​)

Agent hoàn thành nhiệm vụ khi trạng thái hiện tại trùng với goal state.

**Chi phí đường đi (Path cost):** Trong trường hợp đơn giản, mỗi bước đi được tính với chi phí bằng 1, do đó tổng chi phí bằng đúng số bước di chuyển để đến đích.

Thông qua việc xác định rõ các thành phần trên, bài toán mê cung đã được mô hình hóa thành một **bài toán tìm kiếm trong không gian trạng thái**. Đây là cơ sở quan trọng để triển khai các thuật toán tìm kiếm và đánh giá hiệu quả giải quyết đường đi của agent trong những task tiếp theo.

**Describe how you would determin these values for a given maze.**

**n = tổng số trạng thái có thể.**

Trong maze, mỗi ô trống là một trạng thái.

**D = số bước đi ngắn nhất từ start đến goal.**

Cách xác định d:

Dùng thuật toán BFS (luôn tìm đường ngắn nhất trong đồ thị không trọng số).

Đếm số bước đi từ start đến goal trong đường đi tìm được

**M = độ sâu lớn nhất của cây tìm kiếm nếu khám phá toàn bộ state space**

trong maze, xét trường hợp xấu nhất agent đi qua mọi ô trống:

m ≤ n

**b = số hành động khả dĩ tối đa tại bất kỳ trạng thái nào.**

Trong maze chuẩn 2D:

b≤4 (UP, DOWN, LEFT, RIGHT)

Lưu ý: nếu ở cạnh hoặc gần tường b<4.

**Task 2: Uninformed search: Breadth-first and depth-first [40 điểm]**

**Yêu cầu**

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

1. **Breadth-First Search (BFS)** – Tìm kiếm theo chiều rộng.
2. **Depth-First Search (DFS)** – Tìm kiếm theo chiều sâu.

Phải tuân theo **giả mã (pseudocode)** trong sách giáo trình hoặc slide giảng dạy. Sau khi tìm ra nghiệm, hãy sử dụng **cây tìm kiếm (search tree)** để trích xuất đường đi cuối cùng.

**Code BFS:**

A computer screen shot of a program

AI-generated content may be incorrect.

Trong phần code:

* **Node representation:**  
  Mỗi trạng thái (state) trong mê cung được biểu diễn bằng một Node, gồm:
  + state: vị trí hiện tại (tọa độ trong maze).
  + parent: con trỏ trỏ về node cha (để truy vết đường đi).
* **Cấu trúc dữ liệu sử dụng:**
  + frontier: một **hàng đợi (queue, dùng deque)** để lưu các node đang chờ mở rộng.
  + reached: một **tập hợp (set)** lưu các trạng thái đã được thăm, đảm bảo BFS không đi lặp lại.
* **Cơ chế hoạt động:**
  + Khởi tạo frontier với node bắt đầu.
  + Lặp cho đến khi frontier rỗng:
    - Lấy node đầu tiên (popleft) ra để mở rộng.
    - Nếu node này là **goal**, thì dừng lại và **truy vết đường đi** ngược lại bằng cách đi theo chuỗi parent.
    - Nếu chưa phải goal, sinh ra các trạng thái con bằng valid\_actions và move.
    - Mỗi trạng thái con chưa có trong reached sẽ được thêm vào frontier và đánh dấu trong reached.
  + Kết quả: BFS tìm thấy **đường đi ngắn nhất (optimal path)** trong mê cung (theo số bước).

**Code DFS:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Trong phần code:**

* **Node representation:** giống BFS (state + parent).
* **Cấu trúc dữ liệu:**
  + **frontier:** một stack (ngăn xếp, dùng list + pop()) để lưu các node cần mở rộng.
  + Không có reached toàn cục, thay vào đó chỉ kiểm tra vòng lặp trên đường đi hiện tại.
* **Cơ chế hoạt động:**
  + Khởi tạo frontier với node bắt đầu.
  + Lặp cho đến khi frontier rỗng:
    - Lấy node cuối cùng ra để mở rộng (pop – tính chất LIFO của stack).
    - Nếu node này là goal → truy vết đường đi về start bằng parent.
    - Nếu chưa phải goal → sinh ra các trạng thái con.
    - Trước khi thêm node con vào frontier, phải kiểm tra chu trình (cycle checking):
      * Duyệt ngược theo parent của node hiện tại.
      * Nếu next\_state đã tồn tại trong đường đi hiện tại → bỏ qua để tránh vòng lặp vô hạn.
  + Kết quả: DFS tìm được một lời giải (không đảm bảo ngắn nhất), nhưng dùng ít bộ nhớ hơn BFS.

**How does BFS and DFS (without a reached data structure) deal with loops (cycles)?**

Discussion

**1. BFS với tập reached (visited)**

* BFS sử dụng một tập reached để lưu tất cả trạng thái đã được mở rộng.
* Mỗi khi BFS xem xét thêm một node mới vào frontier:
* Kiểm tra xem trạng thái đó đã có trong reached chưa.
* Nếu có → bỏ qua, tránh quay lại trạng thái cũ.
* Nếu chưa → thêm vào frontier và reached.

Điều này đảm bảo BFS không bao giờ rơi vào vòng lặp vô hạn, ngay cả khi mê cung có chu trình.

**2. DFS không dùng tập reached**

* Không có bộ nhớ toàn cục → không nhớ tất cả trạng thái đã đi qua.
* Cycle checking chỉ thực hiện trên đường đi từ root đến node hiện tại.
* Nếu một node con đã xuất hiện trên đường đi hiện tại → không thêm vào frontier → ngăn chặn vòng lặp trực tiếp.

**Hạn chế**: DFS có thể vẫn khám phá lại cùng trạng thái qua các đường khác, nên không đảm bảo hoàn chỉnh trong không gian trạng thái có chu trình hoặc vô hạn.

**Are your implementations complete and optimal? Explain why. What is the time and space complexity of each of \*\*your\*\* implementations? Especially discuss the difference in space complexity between BFS and DFS.**

**1. Completeness và Optimality**

| **Algorithm** | **Complete?** | **Optimal?** | **Giải thích** |
| --- | --- | --- | --- |
| BFS | Có | Có (trong đồ thị không trọng số) | BFS mở rộng theo tầng → sẽ chắc chắn tìm thấy một đường đi nếu tồn tại. Vì mở rộng theo tầng, đường đi đầu tiên đến goal là ngắn nhất (minimal path cost). |
| DFS (không reached) | Không | Không | DFS không nhớ tất cả trạng thái đã thăm → nếu maze có chu trình hoặc không gian vô hạn, DFS có thể rơi vào vòng lặp vô hạn. DFS tìm theo chiều sâu → không đảm bảo tìm được đường đi ngắn nhất. |

**2. Time Complexity (Độ phức tạp thời gian)**

Giả sử:

b = branching factor (số hành động khả dĩ tối đa)

d = độ sâu của giải pháp tối ưu

m = độ sâu tối đa của cây tìm kiếm (DFS có thể đi xa nhất)

**BFS**

Worst-case time: O(b^d)

Vì BFS mở rộng tất cả nút từ tầng 0 đến d.

**DFS (không reached)**

Worst-case time: O(b^m)

DFS có thể đi sâu đến độ sâu tối đa m, khám phá tất cả đường đi trước khi tìm ra solution.

**Nhận xét**: DFS có thể tốn **thời gian hơn nếu m ≫ d, nhưng trong maze nhỏ DFS thường nhanh hơn vì bộ nhớ thấp.**

**3. Space Complexity (Độ phức tạp bộ nhớ)**

**BFS**

Frontier và reached phải lưu tất cả trạng thái ở các tầng từ 0 → d.

Space complexity: O(b^d)

BFS tốn bộ nhớ rất lớn, nhất là trong mê cung rộng hoặc state space lớn.

**DFS (không reached)**

Chỉ lưu stack các node trên đường đi hiện tại (từ gốc → node hiện tại).

Space complexity: O(b.m)

DFS bộ nhớ thấp hơn nhiều so với BFS, vì không cần lưu tất cả node đã thăm.

**Tóm lại:**

BFS → bộ nhớ cao nhưng đảm bảo tìm giải pháp ngắn nhất.

DFS → bộ nhớ thấp, nhưng không đảm bảo tìm ngắn nhất, và có thể không hoàn chỉnh.

**4. Summary Table**

| **Algorithm** | **Complete** | **Optimal** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| **BFS** | Có | Có | O(b^d) | O(b^d) |
| **DFS (no reached)** | Không | Không | O(b^m) | O(b·m) |

**Task 3: Informed Search – Greedy Best-First Search và A\* Search**

1. **Yêu cầu của bài toán**

Trong Task 3, thay vì sử dụng các thuật toán **uninformed search** (BFS, DFS) vốn chỉ dựa trên việc mở rộng nút theo tầng hoặc theo chiều sâu mà **không biết gì về khoảng cách tới đích**, ta cần cài đặt hai thuật toán **tìm kiếm có thông tin (informed search)**:

* **Greedy Best-First Search (GBFS)**
* **A\* Search (A-star)**

Điểm khác biệt quan trọng là cả hai thuật toán này đều sử dụng **hàm heuristic** để ước lượng chi phí từ trạng thái hiện tại đến đích. Trong bài toán mê cung, heuristic được sử dụng là **khoảng cách Manhattan**:

**h(n) = ∣xgoal ​− xn​∣ + ∣ ygoal ​− yn​∣**

**Trong đó:**

* (xn,yn) là tọa độ trạng thái hiện tại.
* (xgoal,ygoal) là tọa độ trạng thái đích.

Khoảng cách Manhattan (hay "taxicab geometry") thích hợp vì agent chỉ có thể di chuyển theo bốn hướng (lên, xuống, trái, phải).

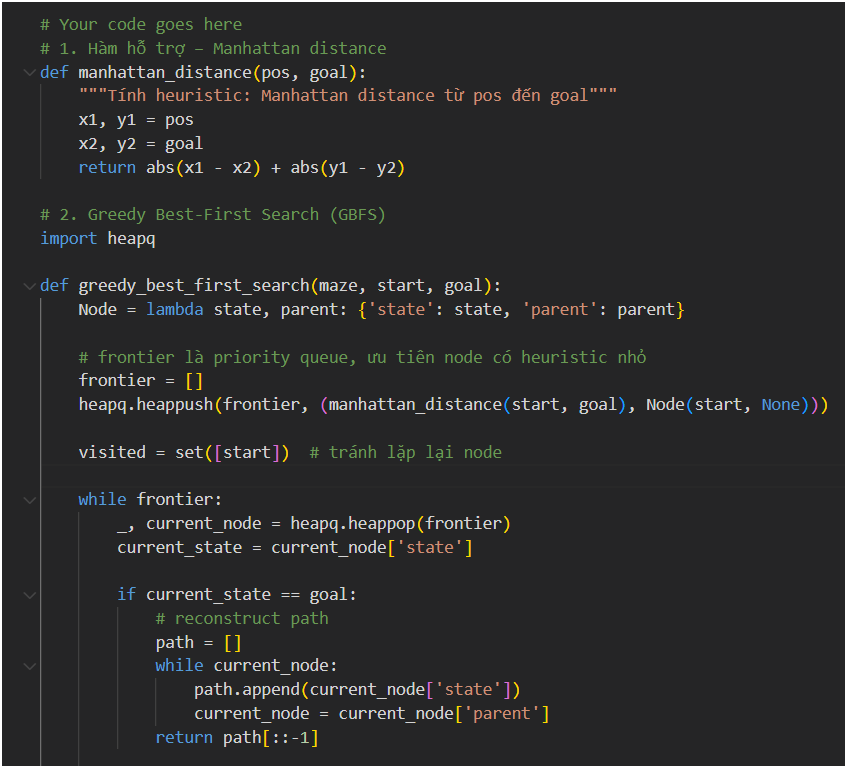
1. **Nguyên lý hoạt động của hai thuật toán**

**Hai thuật toán tìm kiếm có thông tin:**

* **Greedy Best-First Search (GBFS)**: luôn ưu tiên mở rộng node nào "gần goal nhất" theo heuristic (ở đây là Manhattan distance).
* **A\***: kết hợp cả chi phí đã đi (số bước từ start) và khoảng cách còn lại đến goal, nhờ đó vừa hoàn chỉnh vừa tối ưu.

Điểm chung của cả hai: đều sử dụng **priority queue** (heapq) để lựa chọn node tiếp theo dựa trên giá trị ưu tiên.

1. **Code**

****

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A black screen with white text

AI-generated content may be incorrect.**

**Giải thích code:**

**Hàm manhattan\_distance**: hỗ trợ tính giá trị heuristic, dùng để ước lượng khoảng cách còn lại từ vị trí hiện tại đến goal.

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

* Khởi tạo frontier là một priority queue, trong đó các node được xếp theo giá trị heuristic.
* Ở mỗi bước, node có heuristic nhỏ nhất được chọn mở rộng.
* Dùng visited để tránh lặp lại node.
* Khi tới goal, truy ngược theo parent để dựng lại đường đi.
* Ưu điểm: nhanh, nhưng không đảm bảo tối ưu.

**A\* Search**:

* Frontier cũng là priority queue, nhưng giá trị xếp hạng mỗi node là g+h (số bước đã đi + heuristic).
* Sử dụng reached để lưu chi phí tốt nhất tìm được đến mỗi state.
* Nếu tìm thấy một đường đi tốt hơn đến cùng một state, cập nhật lại và đẩy vào frontier.
* Khi tới goal, dựng lại đường đi bằng cách lần theo parent.
* Đặc điểm: đảm bảo tìm được đường đi ngắn nhất nếu heuristic hợp lệ.

**Tóm lại:**

* **GBFS** chọn đường "có vẻ gần goal nhất", nên nhanh nhưng không chắc tối ưu.
* **A\*** chọn đường "vừa gần goal vừa tính cả số bước đã đi", nên vừa hoàn chỉnh vừa tối ưu.

**Are your implementations complete and optimal? What is the time and space complexity?**

**1. Completeness và Optimality**

| **Thuật toán** | **Tính hoàn chỉnh (Completeness)** | **Tính tối ưu (Optimality)** | **Giải thích chi tiết** |
| --- | --- | --- | --- |
| **Greedy Best-First Search (GBFS)** | Không | Không | Thuật toán GBFS chỉ dựa vào giá trị heuristic h(n) để chọn node tiếp theo, nghĩa là nó luôn ưu tiên mở rộng trạng thái có **ước lượng gần goal nhất** theo khoảng cách heuristic. Tuy nhiên, vì không xét đến chi phí thực tế của đường đi (g(n)), GBFS có thể **bỏ qua một số đường đi khả thi** và có thể **mắc kẹt trong các nhánh sai** nếu heuristic không chính xác. Do đó, GBFS **không đảm bảo tìm thấy nghiệm dù có tồn tại** (không hoàn chỉnh) và cũng **không đảm bảo nghiệm tìm được là tối ưu**. |
| **A\*** | Có (nếu heuristic admissible) | Có (nếu heuristic admissible) | Thuật toán A\* kết hợp giữa chi phí thực tế từ điểm xuất phát g(n) và giá trị heuristic h(n) bằng công thức f(n) = g(n) + h(n). Nếu heuristic được chọn là **admissible** (không đánh giá cao hơn chi phí thực sự đến goal), chẳng hạn như Manhattan distance trong môi trường dạng lưới, thì A\* sẽ luôn **khám phá đầy đủ các đường đi cần thiết** để đảm bảo tìm ra nghiệm nếu tồn tại (tính hoàn chỉnh). Đồng thời, nhờ việc kết hợp g(n) và h(n), A\* sẽ chọn được đường đi với **chi phí thấp nhất**, từ đó đảm bảo **tính tối ưu** của nghiệm tìm được. |

**Tóm lại:**

* **GBFS**: Không hoàn chỉnh, không tối ưu, vì chỉ dựa vào heuristic.
* **A\***: Hoàn chỉnh và tối ưu, nếu heuristic là admissible (ví dụ Manhattan distance).

**2. Time Complexity (Độ phức tạp thời gian)**

**Giả sử:**

b = branching factor (số hành động khả dĩ tối đa)

d = độ sâu của giải pháp tối ưu

m = độ sâu tối đa của cây tìm kiếm

| **Thuật toán** | **Độ phức tạp thời gian (Worst-case)** | **Giải thích chi tiết** |
| --- | --- | --- |
| Greedy Best-First Search (GBFS) | O(bm)O(b^m) | Trong trường hợp xấu nhất, GBFS có thể đi theo các nhánh không dẫn đến goal nhưng heuristic lại đánh giá là "gần" goal nhất. Do chỉ dựa vào giá trị heuristic, thuật toán có thể khám phá sâu theo một hướng sai lầm và bỏ qua nhiều hướng khả dĩ khác, dẫn đến việc mở rộng tới độ sâu tối đa mm của không gian trạng thái. |
| A\* | O(bd)O(b^d) đến O(bm)O(b^m), phụ thuộc vào heuristic | Nếu heuristic tốt (admissible và gần với chi phí thực tế), A\* có thể giảm đáng kể số lượng node phải mở rộng, thường chỉ cần mở đến độ sâu nghiệm dd. Tuy nhiên, trong trường hợp heuristic kém (gần như không cung cấp thông tin), A\* có thể thoái hóa thành BFS, dẫn đến độ phức tạp gần O(bm)O(b^m). |

**Nhận xét tổng quát:**

* GBFS có thể rất nhanh nếu heuristic chính xác, nhưng trong không gian tìm kiếm lớn, nó dễ bị rơi vào “đường cụt” và chi phí tính toán tăng mạnh.
* A\* mặc dù có chi phí bộ nhớ và thời gian cao hơn do phải quản lý cả g(n) và h(n), nhưng về mặt lý thuyết luôn hiệu quả hơn BFS/DFS trong các bài toán với heuristic tốt, đồng thời duy trì được tính hoàn chỉnh và tối ưu.

**3. Space Complexity (Độ phức tạp bộ nhớ)**

| Thuật toán | Độ phức tạp bộ nhớ | Giải thích |
| --- | --- | --- |
| Greedy Best-First Search (GBFS) | O(bm)O(b^m) | GBFS duy trì frontier dưới dạng priority queue dựa trên heuristic. Trong không gian tìm kiếm lớn, frontier có thể lưu trữ rất nhiều trạng thái, dẫn đến tốn bộ nhớ tỷ lệ với độ sâu tối đa mm. |
| A\* | O(bd)O(b^d) | Ngoài frontier (priority queue), A\* còn lưu bảng reached (trạng thái và chi phí tốt nhất tìm được). Điều này giúp A\* tránh lặp lại trạng thái nhưng cũng làm tăng nhu cầu bộ nhớ. Tuy vậy, nhờ heuristic hướng về goal, số trạng thái mở rộng thường nhỏ hơn BFS thuần, đặc biệt khi d≪m |

**So sánh với BFS/DFS:**

* GBFS/A\* dùng priority queue → tốn bộ nhớ nhưng kiểm soát tốt nhờ heuristic.
* DFS không dùng reached → bộ nhớ thấp (O(b·m)), nhưng không tối ưu và không hoàn chỉnh.

1. **Summary Table – Task 3**

| Algorithm | Complete | Optimal | Time Complexity | Space Complexity |
| --- | --- | --- | --- | --- |
| Greedy Best-First Search (GBFS) | Không | Không | O(b^m) | O(b^m) |
| A\* | Có (nếu heuristic admissible) | Có (nếu heuristic admissible) | O(b^d)∼O(b^m) tùy heuristic | O(b^d) |

* **GBFS** thường mở rộng nhanh nhờ heuristic nhưng dễ rơi vào bế tắc (không hoàn chỉnh, không tối ưu).
* **A\*** kết hợp chi phí đường đi và heuristic, đảm bảo cả tính **hoàn chỉnh** và **tối ưu** (khi heuristic admissible). Trong thực tế, A\* có hiệu quả cao hơn BFS và đáng tin cậy hơn GBFS.

**Task 4: Comparison and Discussion**

Mục tiêu chính là so sánh các thuật toán đã cài đặt **(BFS, DFS, GBFS và A\*)** thông qua thực nghiệm trên các bản đồ mê cung. Việc chạy thử nghiệm giúp đánh giá hiệu quả và hạn chế của từng thuật toán trong các tình huống khác nhau.

Hướng giải quyết:

1. **Chạy thực nghiệm trên cùng một maze**:

* Cho mỗi thuật toán chạy từ start đến goal.
* Ghi nhận kết quả: độ dài đường đi, thời gian chạy, số node đã mở rộng.
* Nếu thuật toán thất bại (không tìm được đường đi hoặc chạy vô hạn), kết quả sẽ đánh dấu là **N/A**\* kèm theo giải thích.

1. **Xử lý vấn đề phát sinh**:

* Nếu kết quả khác thường (unexpected result), cần **debug** bằng cách trực quan hóa: hiển thị mê cung, đường đi hiện tại, frontier sau mỗi bước. Điều này giúp xác định lỗi cài đặt hoặc vấn đề trong kiểm tra vòng lặp (cycle checking).
* Nếu sau khi debug mà thuật toán vẫn không nhất quán (ví dụ: DFS dễ rơi vào vòng lặp vô hạn khi không kiểm soát reached), kết quả sẽ được đánh dấu **N/A\*** và giải thích đây là **hạn chế cố hữu của thuật toán** chứ không chỉ là lỗi code.

**Code Task 4:**

A computer screen shot of a program code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A computer screen shot of text

AI-generated content may be incorrect.

**Kết quả:**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A black rectangular screen with white text

AI-generated content may be incorrect.**

**Ý nghĩa so sánh**

**BFS**: Hoàn chỉnh và tối ưu trên đồ thị không trọng số, nhưng tốn nhiều bộ nhớ khi không gian trạng thái lớn.

**DFS (không dùng reached)**: Có thể nhanh trong một số trường hợp nhỏ, nhưng dễ rơi vào vòng lặp vô hạn, nên kém ổn định.

**GBFS**: Tìm kiếm nhanh nhờ heuristic, nhưng dễ chọn nhánh sai dẫn đến không hoàn chỉnh hoặc không tối ưu.

**A\***: Kết hợp giữa chi phí thực tế và heuristic, cho kết quả ổn định và tối ưu, thường mở ít node hơn BFS.

**Kết luận**:

Thực nghiệm cho thấy:

* **DFS** dễ thất bại hoặc chạy vô hạn → cần đánh dấu \* hoặc N/A\*.
* **GBFS** nhanh nhưng thiếu ổn định → có thể bỏ sót lời giải ngắn nhất.
* **BFS và A\*** đáng tin cậy nhất, trong đó **A\*** thường hiệu quả hơn khi heuristic được lựa chọn phù hợp.

**Present the results as using charts (see [Python Code Examples/charts and tables]**

**Định nghĩa mê cung:**

**A computer screen shot of a program code

AI-generated content may be incorrect.**

**Trong đó:**

* 0 : ô trống, agent có thể đi qua.
* -1 : ô tường, agent không thể đi qua.

**Các hàm hỗ trợ**

Để agent có thể di chuyển và tìm đường, ta cài đặt các hàm hỗ trợ sau:

A screen shot of a computer program

AI-generated content may be incorrect.

* **valid\_actions(pos, maze)**:

Hàm này kiểm tra từ vị trí hiện tại trong mê cung thì agent có thể đi theo hướng nào. Nó sẽ loại bỏ các hướng dẫn tới tường hoặc ra ngoài mê cung, và trả về danh sách các bước đi hợp lệ (lên, xuống, trái, phải).

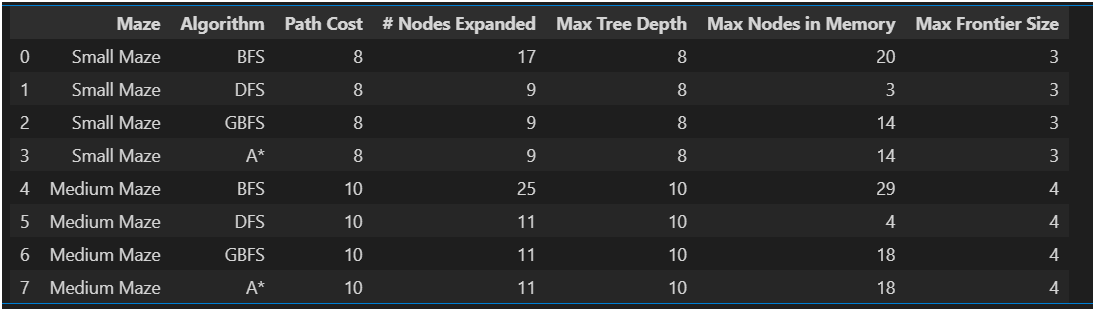
* **move(pos, action)**:

Hàm này thực hiện việc di chuyển. Dựa trên hành động (lên, xuống, trái, phải), nó trả về vị trí mới của agent trong mê cung.

* **manhattan\_distance(pos, goal)**:

Hàm này tính khoảng cách “ước lượng” giữa vị trí hiện tại của agent và vị trí đích, dựa trên tổng số bước đi theo trục ngang và dọc. Khoảng cách này thường dùng để định hướng tìm đường nhanh hơn.

Sau đó sử dụng các hàm đã code ở trên như BFS / DFS / GBFS / A\* with stats để hoàn thành bảng ta có được:



Nhìn vào bảng này có thể thấy:

* **BFS** mở rộng nhiều node nhất và tốn nhiều bộ nhớ nhất (nhất là ở Medium Maze: 29 node trong bộ nhớ).
* **DFS** mở ít node và dùng ít bộ nhớ hơn, nhưng độ sâu tối đa vẫn bằng BFS.
* **GBFS** và **A\*** hoạt động hiệu quả hơn BFS, mở ít node hơn nhưng vẫn tìm ra lời giải có cùng path cost.
* **A\*** có tính ổn định và tối ưu hơn GBFS, mặc dù số liệu trong thử nghiệm nhỏ chưa thể hiện rõ sự khác biệt.

**Discuss the most important lessons you have learned from implementing the different search strategies.**

**1. BFS (Breadth-First Search)**

Hoàn chỉnh (Complete): Luôn tìm được giải pháp nếu tồn tại.

Tối ưu (Optimal): Luôn tìm đường đi ngắn nhất.

Nhược điểm: Chiếm nhiều bộ nhớ vì lưu toàn bộ frontier và reached set → space complexity cao.

Bài học: BFS thích hợp cho maze nhỏ hoặc nơi tối ưu đường đi là quan trọng, nhưng với maze lớn hoặc không gian trạng thái rộng, bộ nhớ là vấn đề lớn.

**2. DFS (Depth-First Search)**

Hoàn chỉnh có điều kiện: Nếu có cycle checking, tránh vòng lặp vô hạn.

Không tối ưu: Đường đi tìm được không nhất thiết ngắn nhất.

Ưu điểm: Bộ nhớ thấp hơn BFS, dễ triển khai theo đệ quy hoặc stack.

Nhược điểm: Có thể mắc kẹt trong không gian mở rộng lớn, hoặc đi theo nhánh sai lâu mới quay lại.

Bài học: DFS phù hợp khi bộ nhớ hạn chế và không quan tâm đến độ dài đường đi ngắn nhất, nhưng phải xử lý cycle checking để tránh vòng lặp vô hạn.

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

Ưu điểm: Dựa vào heuristic → đi thẳng về goal nhanh, ít node được mở rộng.

Nhược điểm: Không tối ưu, có thể mắc kẹt trong bẫy hoặc đường đi dài nếu heuristic sai lệch.

Bài học: Heuristic tốt giúp tăng tốc tìm kiếm đáng kể, nhưng không đảm bảo giải pháp ngắn nhất. GBFS là lựa chọn khi tốc độ quan trọng hơn độ chính xác.

**4. A\* Search**

Hoàn chỉnh và tối ưu nếu heuristic không quá đánh giá cao (admissible).

Hiệu quả: Kết hợp BFS (đảm bảo tối ưu) và GBFS (heuristic hướng về goal) → mở ít node hơn BFS.

Bài học: A\* là sự cân bằng tuyệt vời giữa tối ưu và hiệu quả, nhưng cần heuristic phù hợp để đạt hiệu quả tốt nhất.

**5. Tổng hợp các bài học quan trọng**

Bộ nhớ và độ sâu là yếu tố quan trọng khi chọn thuật toán: BFS dùng nhiều bộ nhớ, DFS ít bộ nhớ nhưng không tối ưu.

Heuristic là chìa khóa: GBFS và A\* chỉ hiệu quả nếu heuristic gần đúng; A\* tối ưu nếu heuristic admissible.

Cycle checking cần thiết: DFS hoặc các thuật toán không lưu reached phải kiểm tra để tránh vòng lặp vô hạn.

Trade-off giữa thời gian, bộ nhớ và tối ưu: Không có thuật toán nào hoàn hảo cho mọi bài toán → cần hiểu yêu cầu để chọn thuật toán phù hợp.

 Visualizing helps debugging: Hiển thị maze và frontier giúp phát hiện lỗi nhanh và hiểu hành vi thuật toán.

**Advanced task: IDS and Multiple goals**

**1. IDS (Iterative Deepening Search)**

* **Ý tưởng**: IDS kết hợp ưu điểm của BFS (đảm bảo tìm được đường đi ngắn nhất) và DFS (tiết kiệm bộ nhớ).
* Nó chạy DFS nhiều lần với giới hạn độ sâu tăng dần (depth limit = 0, 1, 2, …).
* Nếu chưa tìm thấy mục tiêu ở độ sâu hiện tại thì tăng giới hạn và chạy lại.
* Khi đến đúng độ sâu có chứa mục tiêu thì thuật toán sẽ dừng và trả về lời giải.

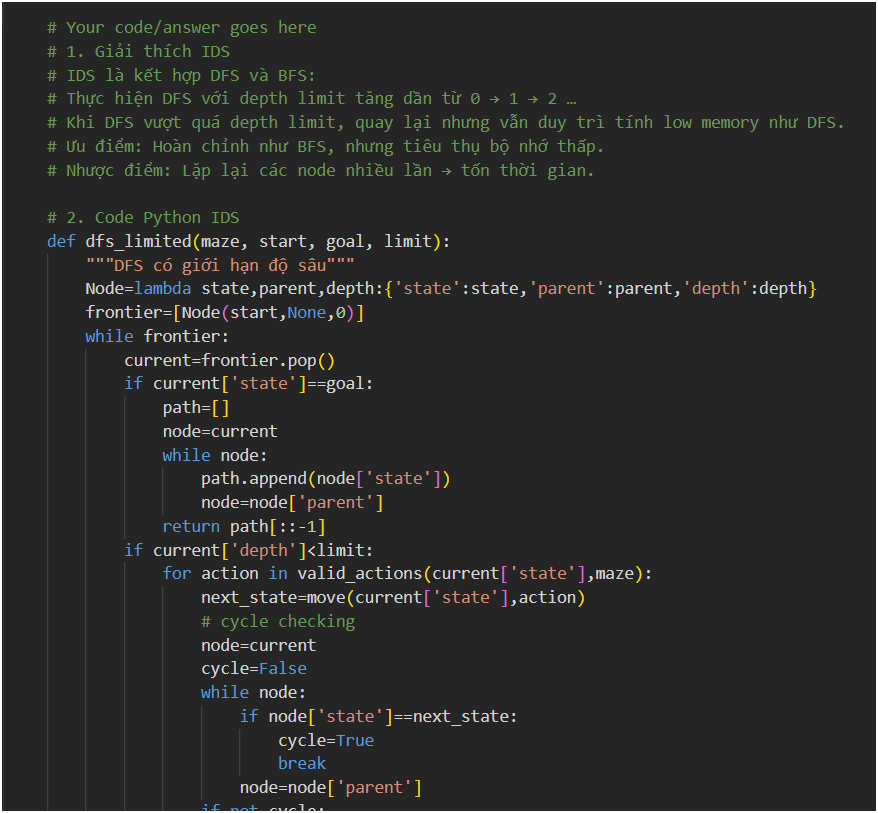
**2. Cách triển khai**

* Sử dụng hàm DFS có sẵn, chỉ cần thêm tham số giới hạn độ sâu (depth limit).
* Trong vòng lặp ngoài, ta tăng dần giới hạn độ sâu.
* Mỗi lần DFS chạy, nếu tìm được đích → trả về kết quả.
* Nếu hết giới hạn độ sâu mà không tìm thấy → tiếp tục lặp.

**3. Vấn đề có thể gặp phải**

* Trong mê cung có không gian mở rộng lớn (open space), IDS sẽ phải duyệt lại rất nhiều trạng thái giống nhau cho mỗi lần tăng độ sâu → lãng phí thời gian.
* IDS phù hợp nhất khi độ sâu của lời giải không quá lớn và không gian tìm kiếm có cấu trúc dạng cây.
* Trong mê cung, do có nhiều chu kỳ (vòng lặp), IDS cần thêm cơ chế visited để tránh đi vào lặp vô tận.

**Code**

****

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A black and white sign with white text

AI-generated content may be incorrect.**

**Multiple Goals**

**1. Ý tưởng**

* Thay vì chỉ có **một đích (goal)**, ta thêm **nhiều đích** vào mê cung.
* Thuật toán sẽ dừng ngay khi chạm được **một trong các đích**.
* Như vậy, thay vì so sánh với duy nhất (x\_goal, y\_goal), ta kiểm tra xem trạng thái hiện tại có nằm trong tập **goal\_set** hay không.

**2. Cách triển khai**

* Tạo một **maze mới** (ví dụ Medium Maze) và thêm 1–2 vị trí goal nữa.
* Khi chạy thuật toán (DFS, BFS, IDS):
  + Nếu trạng thái hiện tại ∈ goal\_set → dừng lại, trả về kết quả.

**3. Thí nghiệm**

Chạy cả 3 thuật toán (DFS, BFS, IDS) trên maze nhiều đích:

* **DFS**:
  + Có thể tìm ra lời giải nhưng **không đảm bảo tối ưu**.
  + Thứ tự duyệt quyết định đích nào được tìm thấy trước.
* **BFS**:
  + Luôn tìm ra **đường đi ngắn nhất** đến *một trong các goal*.
  + Đảm bảo **tối ưu** vì BFS duyệt theo tầng.
* **IDS**:
  + Giống BFS ở chỗ cuối cùng cũng tìm được lời giải tối ưu (nếu không gian hữu hạn).
  + Nhưng tốn thời gian hơn do phải lặp lại nhiều lần DFS với độ sâu tăng dần.

**4. Kết luận thảo luận**

* BFS: tìm ra **đường đi tối ưu** nhanh chóng.
* IDS: cũng tìm ra **đường đi tối ưu** nhưng kém hiệu quả hơn BFS.
* DFS: chỉ đảm bảo tìm được **một đích**, nhưng không chắc là ngắn nhất.

**5.Code**

**A screenshot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

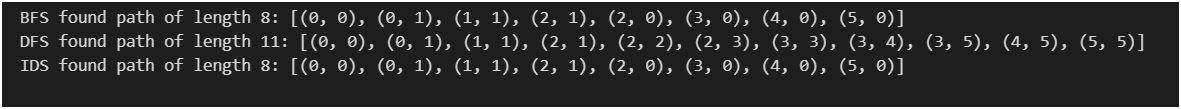
AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A computer screen with text and numbers

AI-generated content may be incorrect.**

****

**Nhận xét:**

Khi giải maze với nhiều goal, các thuật toán hoạt động như sau:

**BFS:** luôn tìm được đường đi ngắn nhất đến goal gần nhất (path length = 8). BFS mở rộng theo mức độ sâu từng bước, nên đảm bảo optimal solution.

**DFS:** tìm được đường đi đến một goal nhưng không đảm bảo ngắn nhất (path length = 11). DFS đi sâu theo nhánh đầu tiên, nên đôi khi đi vòng dài trước khi chạm goal → không optimal.

**IDS:** nhờ thực hiện DFS nhiều lần với độ sâu tăng dần, IDS tìm được đường đi ngắn nhất như BFS (path length = 8). IDS vừa đảm bảo complete vừa optimal, nhưng thời gian chạy thường dài hơn BFS vì DFS bị lặp lại nhiều lần.

**Kết luận:**

BFS và IDS: hoàn chỉnh và tối ưu.

DFS: hoàn chỉnh nhưng không tối ưu.

Cycle checking là cần thiết với DFS/IDS để tránh lặp vô hạn.

Trong multiple goals, agent dừng khi đạt goal đầu tiên, nên đường đi tối ưu là đến goal gần nhất.

**More Advanced Problems to Think About (not for credit)**

**Intersection as States**

**1. Ý tưởng code**

* **Không coi mỗi ô là state** nữa, mà chỉ coi **intersections (ngã ba, ngã tư, dead-end)** là state.
* Sau đó xây dựng **graph**:
  + **Node** = intersection.
  + **Edge** = đường đi thẳng giữa 2 intersections.
  + **Weight** = số ô trống nằm giữa hai intersections.
* Hàm neighbors\_with\_cost() duyệt theo 4 hướng (lên, xuống, trái, phải) và tiếp tục đi cho đến khi gặp một intersection khác, rồi ghi nhận cả điểm đến và chi phí (số ô đi qua).
* Hàm bfs\_intersection() là BFS biến đổi:
  + Frontier mở rộng theo thứ tự FIFO.
  + Nhưng thay vì chỉ tính số bước duyệt node, nó tính **tổng cost dọc theo đường đi**.
  + Cơ chế giống **Uniform Cost Search (UCS)** hơn là BFS cổ điển, vì BFS bình thường không đảm bảo tối ưu khi edge có trọng số khác nhau.

.

**2. Vấn đề phát sinh**

* Khoảng cách giữa 2 giao điểm **không bằng 1 bước** nữa, mà bằng số ô vuông trên đoạn đường đó.
* Ví dụ: đi từ giao điểm A đến B có thể mất 3 bước (3 ô vuông), từ B đến C mất 5 bước.
* Như vậy, **path cost ≠ số bước duyệt state**, mà phải tính tổng số ô vuông đi qua.

**3. Điều chỉnh thuật toán**

* **BFS** và **IDS** mặc định coi chi phí mỗi bước là 1.
* Nhưng giờ phải tính **path cost = số ô vuông giữa 2 giao điểm**.

Cách xử lý:

1. Khi xây dựng graph mới (intersections graph), gán **trọng số** cho cạnh bằng số ô vuông.
   * Ví dụ: cạnh từ A → B có trọng số 3.
2. Khi chạy BFS/IDS:
   * Thay vì chỉ đếm số cạnh, ta phải cộng **tổng trọng số cạnh** để tính path cost.
   * Với BFS: nếu ta chỉ duyệt theo số edge thì không đảm bảo tối ưu.
   * Giải pháp: dùng **Uniform Cost Search (UCS)** hoặc **Dijkstra** → đảm bảo chọn đường có tổng chi phí nhỏ nhất.

**4. Kết luận thảo luận**

* Nếu chỉ coi mỗi intersection là state, thì:
  + **DFS**: vẫn tìm được lời giải nhưng không tối ưu.
  + **BFS/IDS (bản gốc)**: không còn tối ưu, vì chi phí cạnh khác nhau.
  + Để tối ưu: phải thay BFS/IDS bằng **Uniform Cost Search** (hoặc A\* với heuristic).

**Tóm lại:**

* Coi intersections là state sẽ tiết kiệm bộ nhớ.
* Nhưng vì path cost khác nhau, BFS/IDS không còn đảm bảo tối ưu.
* Giải pháp: thay thế bằng UCS hoặc A\*

**5.CODE**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A computer screen with colorful text

AI-generated content may be incorrect.**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**A black screen with white text

AI-generated content may be incorrect.**

**Ý nghĩa kết quả**

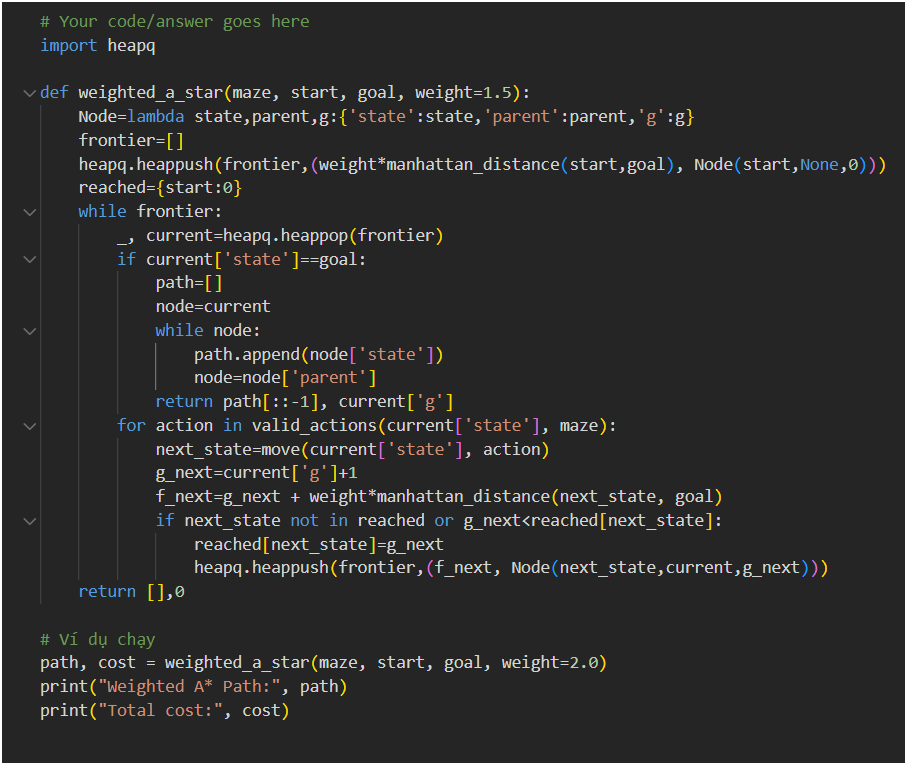
* BFS trên **intersections graph** đã tìm được **đường ngắn nhất theo số ô (path cost)**, không chỉ theo số edge.
* Nếu bạn dùng BFS cổ điển trên từng ô thì kết quả vẫn giống, nhưng sẽ tốn nhiều node hơn (vì phải lưu tất cả ô).
* Cách làm này tiết kiệm bộ nhớ hơn, nhưng vẫn giữ được tính **optimal** nhờ cộng dồn chi phí khi đi qua edges.

**Weighted A\* search**

**Ý tưởng:**

* Weighted A\* giống A\*, nhưng thêm weight w ≥ 1 cho heuristic h(n): f(n)=g(n)+w⋅h(n)
* Khi w = 1, giống A\* bình thường → optimal.
* Khi w > 1, thuật toán ưu tiên heuristic nhiều hơn → có thể tìm đường đi nhanh hơn nhưng không luôn optimal.
* Weight càng lớn → thời gian chạy giảm, độ chính xác có thể giảm.

**CODE**

****

**Hướng đi của Weighted A\***

* Thuật toán giống A\*, chỉ khác ở chỗ hàm f(n) được tính bằng**: f(n)=g(n)+w⋅h(n)**
* Khi w=1w = 1w=1 → chính là A\*, luôn tìm đường đi **ngắn nhất**.
* Khi w>1w > 1w>1 → heuristic quan trọng hơn, nên thuật toán sẽ **ưu tiên đi nhanh về goal** (giống Greedy BFS), đôi khi **bỏ qua đường đi tối ưu**.

**Ý nghĩa các biến trong code**

* frontier: priority queue (min-heap), sắp xếp node theo giá trị f(n).
* reached: lưu chi phí nhỏ nhất để đến mỗi state.
* Node: dictionary lưu state, parent (dùng để reconstruct path), và g (cost thực tế từ start).
* g\_next: cộng thêm 1 cho mỗi bước đi.
* f\_next: công thức đánh giá với trọng số weigh

**Kết quả chạy**

* Nếu chọn weight = 1.0 → Weighted A\* chạy giống A\*, đảm bảo ra **đường đi ngắn nhất**.
* Nếu chọn weight = 1.5 hay weight = 2.0 → đường đi tìm được **ngắn hơn về số bước duyệt (tốc độ)** nhưng **chi phí có thể lớn hơn** so với A\*.
* Nếu chọn weight quá lớn → hành vi gần như **GBFS**: tìm đường nhanh, nhưng rất dễ chọn **đường vòng**
* Weighted A\* là cầu nối giữa **A\*** (chính xác, tối ưu) và **GBFS** (nhanh, không tối ưu).
* Giá trị **weight càng cao** → càng thiên về GBFS.

**Unknown Maze**

What happens if the agent does not know the layout of the maze in advance? This means that the agent faces an unknown environment, where it does not know the transition function. How does the environment look then (PEAS description)? How would you implement a rational agent to solve the maze? What if the agent still has a GPS device to tell the distance to the goal?

**1. Environment (Môi trường) khi maze chưa biết**

**Performance measure (P):**

* Minimize path cost (số bước di chuyển).
* Tìm goal càng nhanh càng tốt.
* Tránh dead-ends và đi vòng lặp.

**Environment (E):**

* Maze 2D chưa biết trước.
* Agent chỉ nhận thông tin tại vị trí hiện tại (ô hiện tại và ô xung quanh).
* Môi trường deterministic nếu di chuyển thành công theo lệnh.
* Partially observable: agent không biết trước map.

**Actuators (A):**

* Move up, down, left, right.

**Sensors (S):**

* Nhận trạng thái ô hiện tại và các ô xung quanh (free/wall).
* Nếu có GPS: nhận thêm khoảng cách Manhattan đến goal.

**2. PEAS description (tóm gọn)**

| **PEAS** | **Chi tiết** |
| --- | --- |
| **Performance** | Đến goal nhanh nhất, tránh dead-end, tối ưu số bước |
| **Environment** | Maze 2D chưa biết, partially observable, deterministic |
| **Actuators** | Move UP, DOWN, LEFT, RIGHT |
| **Sensors** | Thông tin ô hiện tại và lân cận; optional GPS khoảng cách đến goal |

**3. Rational Agent (tác nhân hợp lý)**

**Agent học dần map:**

* Khi đi, lưu lại các ô đã quan sát (free/wall).
* Sử dụng thông tin này để quyết định bước đi tiếp theo.

**Chọn bước đi:**

* Nếu không biết layout, agent có thể dùng online search algorithms:
* DFS/BFS online: mở rộng các ô chưa khám phá.
* LRTA\* (Learning Real-Time A\*): dự đoán khoảng cách đến goal dựa trên thông tin hiện tại, cập nhật dần heuristic khi khám phá.

**Nếu có GPS:**

* Sử dụng Manhattan distance như heuristic → agent hướng về goal.
* Khi khám phá, cập nhật map và heuristic → cải thiện path kế tiếp.

**4. Tóm tắt**

* Unknown maze → môi trường partially observable → agent phải khám phá.
* Agent cần online planning + memory map để giải quyết maze.
* GPS giúp agent định hướng tốt hơn nhưng vẫn cần tránh obstacles.
* Thuật toán thích hợp: Online DFS, BFS, LRTA\*, A online\*

**5.Code**

**A computer screen shot of a program code

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A black background with white text

AI-generated content may be incorrect.**

**Giải thích kết quả:**

**Đường đi tìm được**

* Agent xuất phát từ **(0,0)** và đi theo lối mở xuống và sang phải:

(0,0) → (0,1) → (1,1) → (2,1) → (2,2) → (2,3) → (2,4) → (3,4) → (4,4)

1. **Độ dài đường đi**
   * Có 9 bước (tính cả start và goal).
   * Nếu đếm số ô di chuyển thực sự thì là 8 bước di chuyển → đúng với path length hiển thị.
2. **Tại sao agent chọn đường này?**
   * Thuật toán bạn viết là **online DFS có GPS heuristic**:
     + DFS → dùng stack, mở rộng nhánh sâu trước.
     + GPS (Manhattan distance) → sắp xếp neighbors để **ưu tiên hướng gần goal**.
   * Vì vậy, agent không đi vòng lung tung, mà đi theo một nhánh thẳng hướng về (4,4).
3. **Đánh giá kết quả**
   * Đường đi tìm được **không chắc chắn là ngắn nhất** (optimal) vì DFS vốn không đảm bảo tối ưu.
   * Tuy nhiên, nhờ GPS, nó tìm ra đường hợp lý, gần như tối ưu trong trường hợp này.
   * Nếu chạy A\* hoặc BFS toàn cục, bạn có thể so sánh để thấy rõ hơn sự khác biệt.

Tóm lại:

Kết quả minh họa cho thấy **agent có thể giải maze chưa biết trước** bằng cách kết hợp **khám phá (DFS)** và **định hướng heuristic (GPS)**. Đường đi tìm được dẫn đến goal thành công, với tổng chiều dài 9 bước.