Lời cảm ơn

Nhóm xin gửi lời cảm ơn chân thành và sâu sắc nhất đến **Thầy Nguyễn An Tế** – giảng viên phụ trách môn học Lập trình phân tích dữ liệu ngành Khoa học Dữ liệu. Trong suốt quá trình học tập thầy đã tận tình hướng dẫn, định hướng tư duy và truyền đạt cho nhóm những kiến thức về lập trình, xử lý dữ liệu và các phương pháp phân tích trong lĩnh vực Khoa học Dữ liệu.

Sự chỉ bảo tận tâm và nghiêm túc của thầy không chỉ giúp nhóm hiểu rõ hơn về lý thuyết và kỹ thuật lập trình, mà còn khơi gợi tinh thần nghiên cứu, khả năng tư duy phản biện. Thông qua môn học, nhóm đã có cơ hội vận dụng kiến thức vào thực tiễn, củng cố kỹ năng làm việc nhóm, cũng như nâng cao năng lực tư duy logic và kỹ năng giải quyết vấn đề trên nền tảng dữ liệu.

Cuối cùng, nhóm mong rằng thầy Nguyễn An Tế luôn dồi dào sức khỏe, thành công trong sự nghiệp giảng dạy và tiếp tục truyền cảm hứng học tập, nghiên cứu đến các thế hệ sinh viên sau này.

Nhóm sinh viên thực hiện

Danh mục bảng biểu

Danh mục hình vẽ

Mục lục

Chương 1

Tổng quan đề tài

1.1 Sơ lược đề tài

Trong lĩnh vực Trí tuệ nhân tạo, bài toán tìm đường đi tối ưu đóng vai trò quan trọng trong nhiều ứng dụng thực tế như điều hướng robot, trò chơi, bản đồ thông minh, hay hệ thống logistics. Một trong những thuật toán phổ biến và hiệu quả nhất để giải quyết loại bài toán này là thuật toán A* (A star). Thuật toán A* kết hợp giữa tìm kiếm theo chi phí thực tế (g-cost) và ước lượng chi phí còn lại đến đích (h-cost) để lựa chọn đường đi có tổng chi phí thấp nhất. Ưu điểm của A* là đảm bảo tìm được đường đi ngắn nhất, đồng thời tối ưu về mặt thời gian tìm kiếm so với các thuật toán duyệt toàn bộ. Đề tài "Xây dựng chương trình tìm đường tối ưu cho robot hút bụi bằng thuật toán A*" minh họa ứng dụng thuật toán AI trong đời sống thực tế, cụ thể là việc điều khiển robot tự hành có khả năng di chuyển và làm sạch các ô "dirty" trong một môi trường mô phỏng nhất định.

1.2 Mục tiêu

Mục tiêu của đề tài là phát triển chương trình mô phỏng quá trình làm việc của robot hút bụi sử dụng thuật toán A* nhằm tối ưu hóa lộ trình di chuyển và giảm thiểu tổng chi phí làm sạch. Cụ thể, đề tài tập trung vào ba mục tiêu chính:

- Xây dựng mô hình mô phỏng cho phép người dùng chỉ định kích thước ma trận, số lượng và vị trí ngẫu nhiên của các ô bẩn.
- Áp dụng thuật toán A* để xác định đường đi ngắn nhất giúp robot làm sạch toàn bộ các ô *dirty* với chi phí thấp nhất.

- Đánh giá hiệu quả của thuật toán thông qua việc thể hiện trực quan đường đi của robot, tổng chi phí thực hiện và so sánh các cách tính cho hàm heuristic khác nhau cùng với các thuật toán tìm đường khác như Dijkstra, BFS và DFS.

1.3 Phạm vi

Phạm vi nghiên cứu của đề tài được giới hạn trong mô phỏng đơn giản, robot hút bụi hoạt động trên một ma trận hai chiều gồm các ô vuông. Mỗi ô mang một trong ba trạng thái: ô trống có thể di chuyển (free - F), ô bẩn cần được làm sạch (dirty - D), hoặc ô đã được làm sạch (clean - C). Robot có thể bắt đầu tại bất kỳ vị trí nào trên bản đồ và có khả năng thực hiện hai loại hành động cơ bản là di chuyển (Move) và hút bụi (Suck). Hành động di chuyển cho phép robot tiến đến một trong tám ô lân cận trong khung 3x3, trong khi hành động hút bụi được thực hiện khi robot đứng tại ô bẩn. Mỗi lần di chuyển, robot phải tiêu tốn một đơn vị chi phí. Môi trường được giả định là tĩnh, không có chướng ngại vật hay sự thay đổi trong quá trình di chuyển của robot.

Đề tài chỉ tập trung vào việc mô phỏng và đánh giá hiệu quả của thuật toán A* trong phạm vi ma trận kích thước vừa phải, không đi sâu vào các yếu tố cơ học, cảm biến thực tế hoặc tối ưu hóa năng lượng của robot thật.

Chương 2

Cơ sở lý thuyết

2.1 Mô hình hoạt động của Robot máy hút bụi

Máy hút bụi có thể được mô hình hóa trong lĩnh vực Trí tuệ nhân tạo như một tác tử (agent) hoạt động trong môi trường rời rạc có cấu trúc dạng lưới. Sự trừu tượng hóa này cho phép đơn giản hóa quá trình dọn dẹp thực tế thành một bài toán tìm kiếm hoặc lập kế hoạch (planning problem), trong đó robot di chuyển qua các ô trên không gian hai chiều nhằm đạt trạng thái "sạch hoàn toàn" với chi phí tối thiểu. Cách mô hình hóa này phù hợp với hướng tiếp cận agent–environment được mô tả trong *Artificial Intelligence: A Modern Approach* của Russell and Norvig (2022), ngoài ra thường được sử dụng trong các giáo trình và mô phỏng AI để minh họa quá trình ra quyết định dựa trên thuật toán tìm kiếm.

2.1.1 Môi trường và Trạng thái của các Ô

Môi trường được biểu diễn dưới dạng lưới hai chiều có kích thước $n \times m$, trong đó mỗi ô có một trong ba trạng thái. Trạng thái **Tự do (F)** biểu thị các ô có thể di chuyển mà không cần dọn dẹp. Trạng thái **Bẩn (D)** đại diện cho các ô chứa bụi cần được làm sạch. Trạng thái **Sạch (C)** là kết quả của hành động hút bụi thành công hoặc là các ô ban đầu không có bụi. Trong các mô hình tiêu chuẩn (Russell & Norvig, 2022), các ô chỉ được phân biệt là "bẩn" hoặc "sạch", song trong một số mô hình mở rộng, có thể bổ sung thêm chướng ngại vật (obstacle) là các ô không thể di chuyển qua. Trạng thái tổng thể của môi trường được mô tả bởi vị trí hiện tại của robot cùng cấu hình của toàn bộ các ô trên lưới. Mục tiêu của robot là đạt tới trạng thái mà mọi ô đều được chuyển về trạng thái Sạch.

2.1.2 Mô hình hoạt động cơ bản: Di chuyển và Hút bụi

Robot vận hành dựa trên mô hình chuyển tiếp trạng thái để lập kế hoạch hành động. Theo Sutton and Barto (2018), quá trình hoạt động của tác tử có thể chia thành các giai đoạn:

- Nhận thức – robot cảm nhận vị trí của mình và trạng thái cục bộ của môi trường, với giả định rằng toàn bộ không gian là quan sát được đầy đủ.
- Lập kế hoạch – sử dụng các thuật toán tìm kiếm như Breadth-First Search (BFS), Dijkstra, hoặc A* để xác định chuỗi hành động tối ưu dẫn đến mục tiêu.
- Thực thi – robot di chuyển đến các ô bẩn gần nhất và thực hiện hành động hút bụi để làm sạch khu vực, đồng thời cập nhật lại trạng thái môi trường sau mỗi bước.
- Kết thúc – quá trình hoạt động kết thúc khi tất cả các ô bẩn đều được dọn sạch, tức là robot đã đạt được trạng thái mục tiêu của môi trường.

Mô hình này có thể được mở rộng để thích ứng trong thời gian thực, giúp robot lập kế hoạch lại khi môi trường thay đổi trong quá trình vận hành.

2.1.3 Hành động của Robot

Robot có hai loại hành động cơ bản. Thứ nhất là **Di chuyển (Move)**, cho phép robot tiến đến bất kỳ ô liền kề nào trong tám hướng (lên, xuống, trái, phải và chéo), miễn là ô mục tiêu không bị chướng ngại. Mô hình di chuyển tám hướng (8-connectivity) thường được áp dụng trong các bài toán tìm đường. Di chuyển chéo có thể được gán chi phí cao hơn — ví dụ, $\sqrt{2}$ thay vì 1 — để phản ánh độ dài thực tế trong không gian Euclid, song trong nhiều nghiên cứu mô phỏng đơn giản, chi phí này được giả định đồng nhất nhằm giảm độ phức tạp tính toán. Thứ hai là **Hút bụi (Suck)**. Khi robot đứng tại một ô bẩn (D), hành động hút bụi sẽ chuyển trạng thái ô đó thành Sạch (C). Nếu ô đang ở trạng thái Tự do hoặc đã Sạch, hành động này không làm thay đổi môi trường nhưng vẫn có thể tiêu tốn năng lượng hoặc thời gian thực hiện. Các hành động trong mô hình này được giả định là xác định (deterministic), tức kết quả của hành động là chắc chắn.

2.1.4 Định nghĩa Chi phí

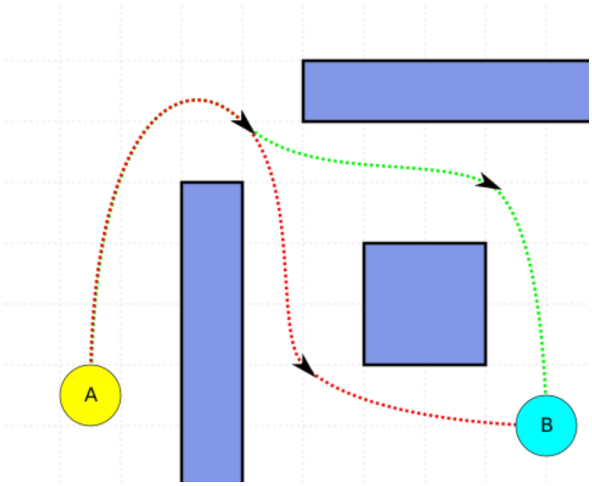
Khái niệm chi phí (cost) được sử dụng để đánh giá hiệu quả của quá trình lập kế hoạch, với mục tiêu tối thiểu hóa tổng chi phí di chuyển từ trạng thái ban đầu đến trạng thái mục tiêu. Theo Pearl (1984), tổng chi phí đường đi là cơ sở để xác định tính tối ưu của thuật toán tìm kiếm. Chi phí di chuyển (move cost) thường được giả định cố định và nhỏ, ví dụ 1 cho mỗi bước di chuyển. Ngược lại, chi phí hút bụi (suck cost) được đặt cao hơn, chẳng hạn 5 cho mỗi lần hút, để phản ánh mức tiêu hao năng lượng lớn hơn so với việc di chuyển.

Một số mô hình còn mở rộng chi phí hút bụi theo hướng tăng dần – nghĩa là chi phí của hành động có thể tăng theo số lần hút, hoặc phụ thuộc vào lượng bụi còn lại trong môi trường.

Đường đi tối ưu được xác định bằng các thuật toán tìm kiếm có nhận thức chi phí, như Uniform-Cost Search hoặc A* kết hợp với hàm heuristic phù hợp, chẳng hạn khoảng cách Manhattan hoặc Chebyshev để ước lượng chi phí từ vị trí hiện tại đến ô bẩn gần nhất.

2.2 Bài toán tìm đường đi

2.2.1 Khái niệm



Hình 2.1: Minh họa bài toán tìm đường đi

Trong lý thuyết đồ thị, bài toán tìm đường đi được xem như một dạng của bài toán tìm kiếm trên không gian trạng thái (*state-space search*) nhằm xác định một chuỗi hành động tối ưu để chuyển trạng thái ban đầu (điểm xuất phát) đến trạng thái mục

tiêu (điểm đích) với chi phí thấp nhất có thể Russell and Norvig (2022). Mục tiêu của bài toán có thể thay đổi tùy theo ngữ cảnh: tìm đường đi có tổng chi phí di chuyển nhỏ nhất, đi qua tất cả các đỉnh trong đồ thị ít nhất một lần – như trong bài toán người du lịch Dantzig et al. (1954), hoặc tìm đường trong không gian có vật cản nhằm tránh các khu vực không thể di chuyển LaValle (2006).

2.2.2 Các thành phần cơ bản

Bài toán tìm đường bao gồm một số thành phần cơ bản cấu thành nên mô hình tìm kiếm. Thứ nhất là **Trạng thái (state)**, đại diện cho một tình huống cụ thể trong không gian tìm kiếm, thường tương ứng với một đỉnh trong đồ thị. Thứ hai là **Hành động (action)**, là tập hợp các phép biến đổi được áp dụng để di chuyển từ trạng thái hiện tại đến trạng thái mới, tương ứng với các cạnh của đồ thị.

Một thành phần quan trọng khác là **Hàm chi phí** $g(n)$, biểu diễn chi phí tích lũy phải trả khi di chuyển từ trạng thái ban đầu đến trạng thái n. Nếu các trạng thái thay đổi lần lượt theo chuỗi $n_0 \to n_1 \to n_2 \to \cdots \to n_k$, trong đó n_0 là trạng thái ban đầu và n_k là trạng thái hiện tại, thì chi phí được xác định bởi công thức:

$$g(n) = \sum_{i=0}^{k-1} w(n_i, n_{i+1})$$

Trong đó, $w(n_i, n_{i+1})$ là chi phí của hành động di chuyển từ trạng thái n_i đến n_{i+1}. Giá trị $g(n)$ có thể biểu thị cho thời gian, năng lượng, quãng đường, hoặc bất kỳ đại lượng nào mà bài toán cần tối ưu hóa (Pearl, 1984).

Cuối cùng là **Hàm đánh giá (heuristic function)** $h(n)$, biểu diễn ước lượng chi phí còn lại từ trạng thái hiện tại đến trạng thái mục tiêu. Hàm heuristic đóng vai trò định hướng quá trình tìm kiếm bằng cách cung cấp ước lượng có cơ sở về chi phí cần thiết để đạt mục tiêu Felner (2011). Một hàm heuristic được xem là *chấp nhận được* (*admissible*) nếu thoả mãn điều kiện $h(n) \leq h^*(n)$, trong đó $h^*(n)$ là chi phí thực tế từ trạng thái hiện tại đến đích. Điều này đảm bảo rằng thuật toán tìm kiếm, chẳng hạn như A*, sẽ không bao giờ đánh giá thấp chi phí thực sự và do đó duy trì được tính tối ưu của lời giải.

Tổng hợp hai thành phần trên, bài toán tìm đường đi có thể được biểu diễn thông qua hàm đánh giá tổng hợp:

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

Trong đó, $g(n)$ là chi phí thực tế từ trạng thái ban đầu đến n, còn $h(n)$ là chi phí ước

lượng từ n đến đích. Hàm $f(n)$ đại diện cho chi phí ước lượng toàn phần của đường đi thông qua trạng thái n, là cơ sở cốt lõi cho các thuật toán tìm kiếm heuristic như A*.

2.3 Thuật toán A*

2.3.1 Giới thiệu chung

Thuật toán A* được Hart et al. (1968) đề xuất là một trong những thuật toán tìm kiếm có thông tin (*informed search*) phổ biến và hiệu quả nhất trong các bài toán tìm đường đi và duyệt đồ thị. A* hoạt động dựa trên việc kết hợp chi phí thực tế từ điểm xuất phát đến trạng thái hiện tại ($g(n)$) với ước lượng chi phí còn lại đến mục tiêu ($h(n)$), tạo thành hàm đánh giá $f(n) = g(n) + h(n)$ dùng để lựa chọn đường đi tối ưu.

A* có thể được xem như sự kết hợp giữa hai thuật toán tìm kiếm kinh điển. Thứ nhất là thuật toán Dijkstra Dijkstra (1959), vốn đảm bảo tìm được đường đi ngắn nhất bằng cách mở rộng toàn bộ các đỉnh có chi phí nhỏ nhất từ gốc. Thứ hai là thuật toán Greedy Best-First Search (GBFS), hoạt động dựa trên hàm heuristic để chọn trạng thái được xem là gần với mục tiêu nhất. Trong khi Dijkstra đảm bảo tính tối ưu nhưng kém hiệu quả về thời gian, GBFS lại nhanh hơn nhưng không đảm bảo tìm được đường ngắn nhất do không xét đến trọng số các cạnh. A* dung hòa giữa hai cách tiếp cận này bằng việc đồng thời sử dụng cả chi phí thực ($g(n)$) và ước lượng ($h(n)$), giúp quá trình tìm kiếm vừa có định hướng vừa đảm bảo tính tối ưu.

Nhờ đặc tính này, A* được ứng dụng rộng rãi trong nhiều lĩnh vực thực tế, chẳng hạn như hệ thống định vị và dẫn đường, robot và các hệ thống tự động hóa, cũng như trong phát triển trò chơi điện tử và mô phỏng hành vi nhân vật.

2.3.2 Cấu trúc dữ liệu và thuật toán

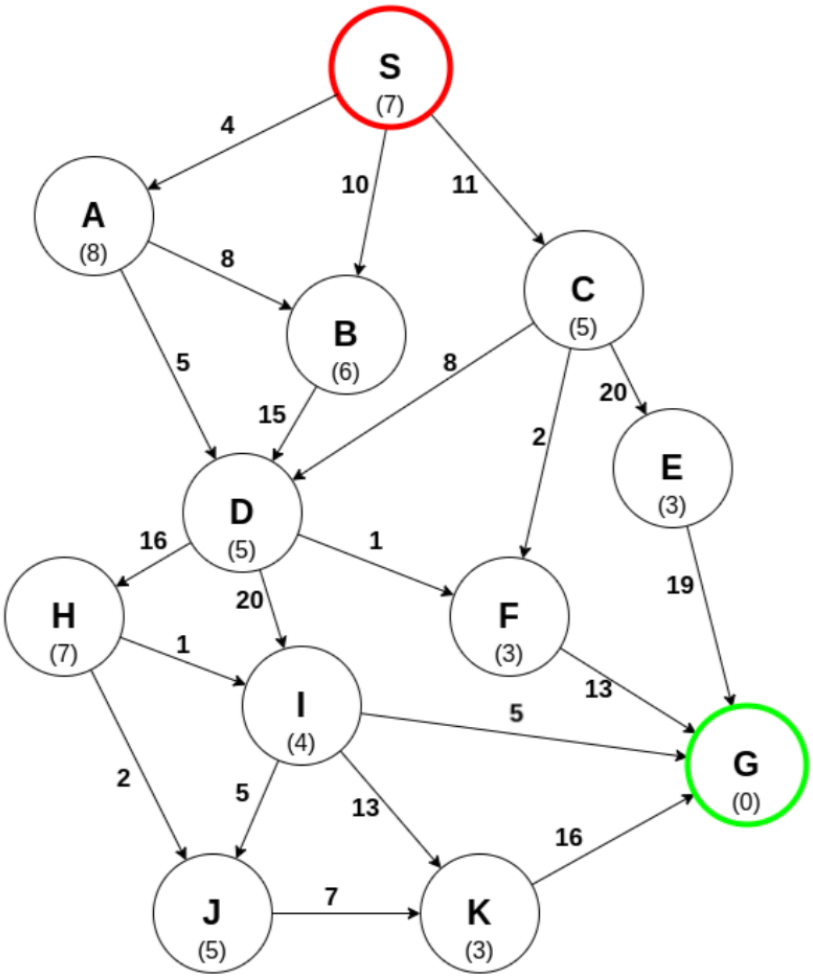
Thuật toán A* sử dụng một số cấu trúc dữ liệu chính để quản lý quá trình tìm kiếm. Đầu tiên là **tập mở (open set)**, thường được cài đặt dưới dạng hàng đợi ưu tiên (*priority queue*) chứa các trạng thái đang chờ được mở rộng, được sắp xếp theo giá trị $f(n) = g(n) + h(n)$ tăng dần. Tiếp theo là **tập đóng (closed set)**, lưu các trạng thái đã được duyệt và không cần xem xét lại. Ngoài ra, một bảng ánh xạ (*dictionary* hoặc *hash map*) được duy trì để lưu thông tin "điểm cha" của từng trạng thái, giúp tái tạo lại đường đi ngắn nhất khi tìm thấy mục tiêu.

Quy trình thuật toán

Thuật toán A* bắt đầu bằng việc khởi tạo tập mở với điểm xuất phát S. Trong mỗi vòng lặp, thuật toán chọn trạng thái có giá trị $f(n)$ nhỏ nhất trong tập mở để mở rộng. Trạng thái này sau đó được chuyển sang tập đóng, và các trạng thái kề (hàng xóm) của nó sẽ được tính toán lại chi phí $f(n)$ để quyết định có thêm vào tập mở hay không.

Giả sử trong một ví dụ đơn giản, điểm xuất phát S có ba điểm kề là A, B, và C. Khi đó, giá trị $f(n)$ được tính như sau:

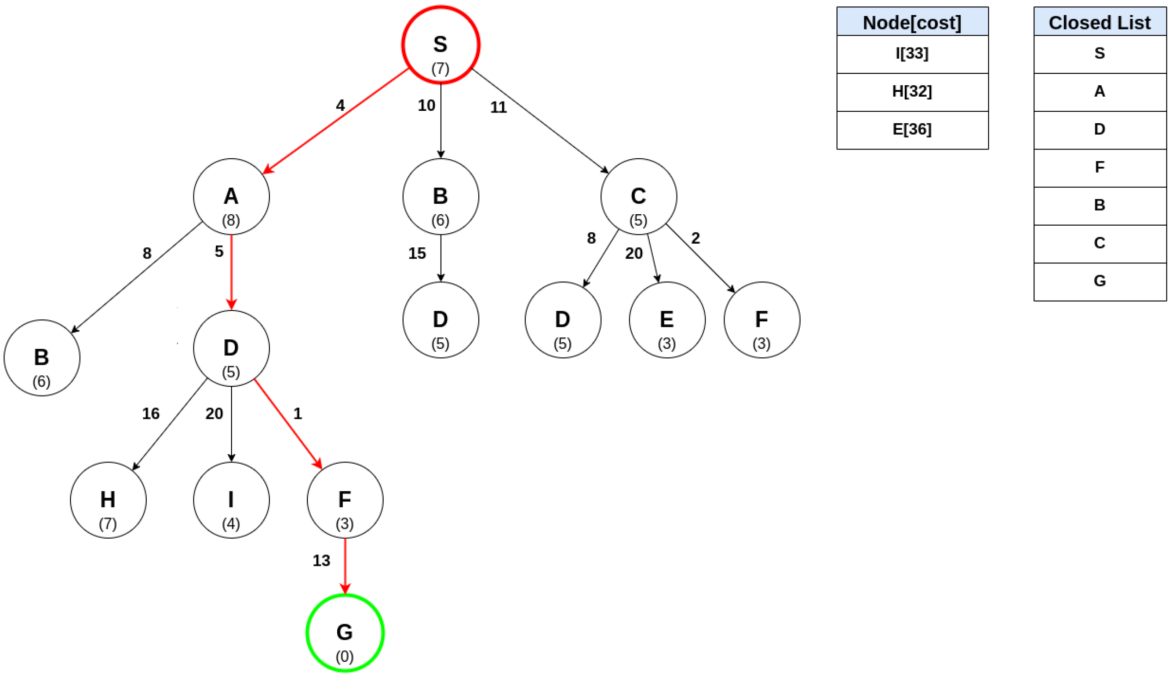
$$f(A) = g(A) + h(A) = 4 + 8 = 12, \quad f(B) = 10 + 6 = 16, \quad f(C) = 11 + 5 = 16$$



Hình 2.2: Minh họa ví dụ tìm đường ngắn nhất từ S đến G trong đồ thị. Nguồn: SpaceDev (2024)

Vì $f(A)$ nhỏ nhất, điểm A được chọn để mở rộng tiếp theo. Sau khi duyệt các điểm kề của A (ví dụ B và D), thuật toán so sánh lại giá trị $f(B)$ mới với giá trị cũ. Nếu giá

trị mới lớn hơn, A* giữ nguyên giá trị nhỏ hơn hiện có. Quá trình này được lặp lại cho đến khi đạt được trạng thái mục tiêu G. Ngay cả khi G đã có trong tập mở, thuật toán vẫn tiếp tục cho đến khi $f(G)$ là nhỏ nhất trong tất cả các trạng thái đang xét, đảm bảo tính tối ưu toàn cục.



Hình 2.3: Lời giải của thuật toán Astar trong đồ thị. Nguồn: SpaceDev (2024)

Trong quá trình tìm kiếm, A* lưu lại các trạng thái cha trong bảng ánh xạ để truy ngược đường đi tối ưu khi đạt mục tiêu. Kết quả là đường đi ngắn nhất được xác định, chẳng hạn như chuỗi $S \to A \to D \to F \to G$, với tổng chi phí tối ưu và số bước di chuyển tối thiểu.

2.3.3 Các hàm heuristic của A*

Trọng tâm của thuật toán A* nằm ở cách lựa chọn hàm heuristic $h(n)$, vốn ước lượng chi phí tối thiểu còn lại từ trạng thái hiện tại đến mục tiêu. Một hàm heuristic hiệu quả sẽ giúp giảm đáng kể số lượng trạng thái cần mở rộng, từ đó cải thiện tốc độ tìm kiếm (Felner, 2011).

Các hàm heuristic phổ biến bao gồm:

- **Khoảng cách Euclid (Euclidean distance)** – tính bằng công thức

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

mô phỏng chuyển động đường thẳng trong không gian 2D.

- **Khoảng cách Manhattan (Manhattan distance)** – tính bằng

$$d = |x_1 - x_2| + |y_1 - y_2|$$

áp dụng trong mô hình di chuyển 4 hướng.

- **Khoảng cách Chebyshev (Chebyshev distance)** – tính bằng

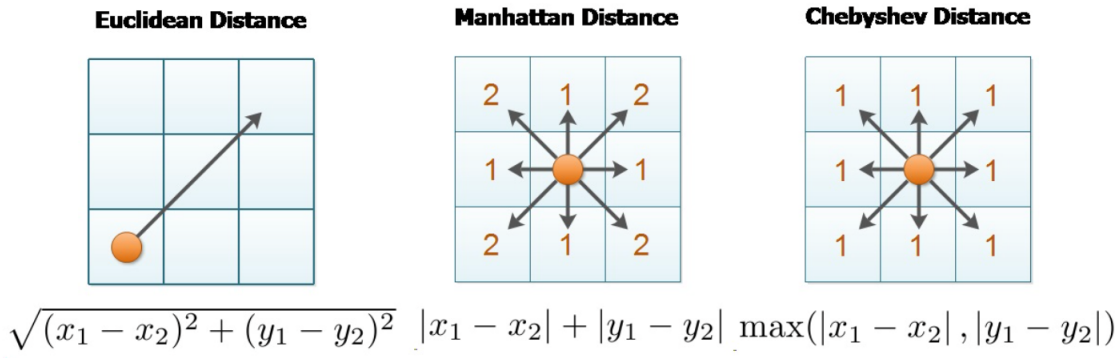
$$d = \max(|x_1 - x_2|, |y_1 - y_2|)$$

cho phép di chuyển theo cả hướng dọc và chéo, tuy nhiên chỉ được 1 bước.Phù hợp với môi trường 8 hướng.

- **Khoảng cách Octile (Octile distance)**: đặt $dx = |x_1 - x_2|$, $dy = |y_1 - y_2|$, $M = \max(dx, dy)$, $m = \min(dx, dy)$. Khi giả định chi phí bước thẳng bằng 1 và bước chéo bằng $\sqrt{2}$, ta có

$$d = (M - m) + m\sqrt{2}$$

phù hợp môi trường 8 hướng với chi phí chéo lớn hơn chi phí thẳng.



Hình 2.4: Minh họa các hàm Heuristic. Nguồn: IQ (2023)

Trong bài toán Máy hút bụi, robot được phép di chuyển theo 8 hướng với chi phí di chuyển đồng nhất. Do đó, khoảng cách Chebyshev được xem là hàm heuristic phù hợp nhất, vì nó phản ánh chính xác khoảng cách thực tế mà robot phải di chuyển trong không gian lưới.

Chương 3

Thiết kế và cài đặt chương trình

3.1 Cài đặt chương trình

3.1.1 Lớp `Position`

Lớp `Position` được thiết kế nhằm biểu diễn một điểm trong không gian hai chiều (2D) thông qua hai tọa độ x và y. Mỗi đối tượng thuộc lớp này tượng trưng cho một ô (cell) trong bản đồ hoạt động của robot, giúp xác định vị trí cụ thể trong lưới tọa độ.

```
1 class Position:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
```

Lớp `Position` đóng vai trò là thành phần cơ bản trong việc quản lý không gian hoạt động của robot. Hai thuộc tính chính gồm: x – tọa độ theo trục hoành và y – tọa độ theo trục tung.

3.1.2 Lớp `Cell`

Lớp `Cell` đại diện cho một ô (cell) trong bản đồ hoạt động của robot. Mỗi đối tượng `Cell` chứa thông tin về tọa độ, trạng thái, chi phí di chuyển, và liên kết với ô cha, giúp mô hình hóa không gian và hỗ trợ quá trình tìm đường. Các thành phần chính bao gồm:

- `position`: lưu thông tin tọa độ (đối tượng `Position`),
- `parent`: ô cha trong đường đi,

- `status`: tình trạng ô (*free, start, dirty, clean*),
- `g`, `h`, `f`: lần lượt là chi phí thực, chi phí ước lượng và tổng chi phí.

```
class Cell:
    def __init__(self, position:(), parent:(), status, g, h, f):
        self.position = position      # (x, y)
        self.parent   = parent        # parent (object) in a path
        self.status   = status        # {'free','start','dirty','clean'}
        self.g        = g             # current cost
        self.h        = h             # estimated cost
        self.f        = f             # full path cost
```

3.1.3 Lớp `Grid`

Lớp `Grid` được thiết kế nhằm mô phỏng môi trường hoạt động của robot hút bụi. Không gian làm việc được biểu diễn dưới dạng ma trận $m \times n$ gồm các ô (cells). Mục tiêu của lớp là:

- Xây dựng cấu trúc dữ liệu lưới cho thuật toán A*;
- Quản lý trạng thái của từng ô (*free, dirty, start, clean*);
- Cung cấp các hàm tiện ích để kiểm tra biên, sinh láng giềng và cập nhật trạng thái.

Mô hình toán học Không gian lưới được biểu diễn bởi:

$$Grid = \{cell_{i,j} \mid 0 \leq i < m, 0 \leq j < n\}$$

Trạng thái của mỗi ô:

$$status(cell_{i,j}) \in \{free, start, dirty, clean\}$$

Tập các ô bẩn cần làm sạch:

$$D = \{(x_k, y_k) \mid k = 1, 2, \ldots, N\}, \quad S = (x_s, y_s)$$

Mục tiêu là tìm đường đi từ S qua toàn bộ D với chi phí nhỏ nhất.

Cơ chế hoạt động Lưới được khởi tạo với trạng thái *free*, sau đó gán các ô *start* và *dirty*. Hàm `in_bounds(x, y)` kiểm tra biên, `neighbors(x, y)` sinh láng giềng hợp lệ theo 8 hướng:

$$MOVES_8 = \{(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)\}$$

```
MOVES_8 = [(-1,-1), (-1,0), (-1,1),
           ( 0,-1),         ( 0,1),
           ( 1,-1), ( 1,0), ( 1,1)]

class Grid:
    def __init__(self, m, n, dirty_positions, start):
        self.m = m
        self.n = n
        self.cells = [[Cell(Position(x, y), None, 'free', 0, 0, 0)
                       for x in range(n)] for y in range(m)]
        self.start = start
        self.dirty_set = set(dirty_positions)
        if start in self.dirty_set:
            self.dirty_set.remove(start)
        self.cells[start[1]][start[0]].status = 'start'
        for (dx, dy) in dirty_positions:
            if (dx, dy) != start:
                self.cells[dy][dx].status = 'dirty'

    def in_bounds(self, x, y):
        return 0 <= x < self.n and 0 <= y < self.m

    def neighbors(self, x, y):
        for dx, dy in MOVES_8:
            nx, ny = x + dx, y + dy
            if self.in_bounds(nx, ny):
                yield (nx, ny)

    def status(self, x, y):
        return self.cells[y][x].status

```

```
32      def set_status(self, x, y, s):
33          self.cells[y][x].status = s
```

3.1.4 Các hàm metric

Các hàm metric được xây dựng nhằm đo lường khoảng cách giữa hai ô – đóng vai trò là hàm heuristic cho A*. Giả sử hai điểm $a = (x_1, y_1)$ và $b = (x_2, y_2)$, ta có:

$$dx = |x_1 - x_2|, \quad dy = |y_1 - y_2|$$

Bốn metric được sử dụng:

- **Manhattan**: $d_M(a, b) = dx + dy$
- **Euclid**: $d_E(a, b) = \sqrt{dx^2 + dy^2}$
- **Chebyshev**: $d_C(a, b) = \max(dx, dy)$
- **Octile**: $d_O(a, b) = (M - m) + m\sqrt{2}$ với $M = \max(dx, dy), m = \min(dx, dy)$

```
1   def chebyshev(a, b):
2       return max(abs(a[0]-b[0]), abs(a[1]-b[1]))
3
4   def manhattan(a, b):
5       dx, dy = abs(a[0]-b[0]), abs(a[1]-b[1])
6       return dx + dy
7
8   def euclid(a, b):
9       dx, dy = abs(a[0]-b[0]), abs(a[1]-b[1])
10      return (dx**2 + dy**2) ** 0.5
11
12  def octile(a, b):
13      dx, dy = abs(a[0]-b[0]), abs(a[1]-b[1])
14      m, M = min(dx, dy), max(dx, dy)
15      return (M - m) + m * math.sqrt(2.0)
```

3.1.5 Lớp `ProblemEncoder`

Lớp `ProblemEncoder` đóng vai trò trung tâm, đóng gói toàn bộ bài toán tìm kiếm. Nó định nghĩa không gian trạng thái, luật chuyển trạng thái và điều kiện dừng của robot hút bụi. Lớp nhận đầu vào là `Grid` và danh sách các ô bẩn, cung cấp các hàm `initial_state()`, `actions()`, `step()` và `goal_test()`.

Hàm khởi tạo Dùng để lưu môi trường `Grid` và danh sách các ô bẩn để ánh xạ giữa chỉ số bit và vị trí tương ứng trong không gian lưới.

```
def __init__(self, grid, dirty_positions):
    self.grid = grid
    self.m = grid.m
    self.n = grid.n
    self.dirty_positions = list(dirty_positions)
    self.index_by_pos = {pos:i for i, pos in enumerate(self.dirty_positions)}
    self.K = len(self.dirty_positions)
```

Hàm khởi tạo trạng thái ban đầu Trả về state tuple $(x_0, y_0, mask)$ với $mask$ gồm các bit 1 tương ứng với ô bẩn ban đầu.

```
def initial_state(self):
    x0, y0 = self.grid.start
    mask = 0
    for i, pos in enumerate(self.dirty_positions):
        mask = self.set_bit(mask, i)
    return (x0, y0, mask)
```

Hàm sinh hành động Sinh ra tập hợp các hành động hợp lệ tại trạng thái hiện tại. Nếu robot đang ở ô bẩn, thêm hành động `('suck', (x, y))`, ngoài ra có thể di chuyển đến các ô láng giềng.

```
def actions(self, state):
    x, y, mask = state
    actions = []
    idx = self.pos_to_index((x, y))
    if idx is not None and self.is_dirty(mask, idx):
```

```
        actions.append(('suck', (x, y)))
    for nx, ny in self.grid.neighbors(x, y):
        actions.append(('move', (nx, ny)))
    return actions
```

Hàm chuyển trạng thái Thực hiện bước chuyển theo hành động kèm chi phí. Nếu hành động là `'suck'`, chi phí được tính dựa trên số ô bẩn còn lại sau khi hút; nếu hành động là `'move'`, chi phí dựa trên số ô bẩn còn lại trước khi di chuyển.

```
def step(self, state, action):
    x, y, mask = state
    name, param = action
    if name == 'suck':
        idx = self.pos_to_index((x, y))
        new_mask = self.clear_bit(mask, idx) if idx is not None else mask
        penalty = self.dirty_count(new_mask)
        step_cost = 1 + penalty
        return (x, y, new_mask), step_cost
    else: # move
        nx, ny = param
        penalty = self.dirty_count(mask)
        step_cost = 1 + penalty
        return (nx, ny, mask), step_cost
```

Hàm kiểm tra mục tiêu Trả về `True` nếu $mask = 0$, tức không còn ô bẩn nào trong bản đồ.

```
def goal_test(self, state):
    return state[2] == 0
```

3.1.6 Các hàm heuristic

Hàm khoảng cách Manhattan Hàm này được sử dụng cho môi trường di chuyển 4 hướng, ước lượng tổng khoảng cách cần di chuyển còn lại.

```
def manhattan(a, b):
    return abs(a.x - b.x) + abs(a.y - b.y)
```

Hàm ước lượng dựa trên cây bao trùm nhỏ nhất (MST) Hàm heuristic nâng cao này ước lượng chi phí tối thiểu cần để kết nối toàn bộ các điểm bẩn. Chi phí của cây bao trùm nhỏ nhất là cận dưới cho hành trình tối ưu, vì mọi hành trình qua tất cả các điểm đều phải bao chứa một cây khung.

```
def mst_lower_bound(points, distance_func):
    if len(points) <= 1:
        return 0
    remaining = set(range(len(points)))
    current = next(iter(remaining))
    remaining.remove(current)
    in_tree = {current}
    total = 0
    while remaining:
        best = None
        best_d = float('inf')
        for i in in_tree:
            for j in remaining:
                d = distance_func(points[i], points[j])
                if d < best_d:
                    best_d = d
                    best = j
        total += best_d
        in_tree.add(best)
        remaining.remove(best)
    return total
```

3.1.7 Khởi chạy thuật toán A*

Hàm tái tạo đường đi Dựng lại toàn bộ đường đi tối ưu dựa trên bảng `came_from`.

```
def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path
```

Hàm tính chi phí tổng thể Sau khi A* tìm được đường đi, hàm `compute_plan_cost` sẽ tính tổng chi phí, số bước và số ô bẩn còn lại.

```
def compute_plan_cost(encoder, path_states):
    if not path_states:
        return 0, 0, 0
    cost = 0
    steps = 0
    for i in range(1, len(path_states)):
        prev = path_states[i-1]
        cur  = path_states[i]
        if prev[0] == cur[0] and prev[1] == cur[1] and prev[2] != cur[2]:
            penalty = bin(cur[2]).count("1")
            step_cost = 1 + penalty
        else:
            penalty = bin(prev[2]).count("1")
            step_cost = 1 + penalty
        cost += step_cost
        steps += 1
    leftovers = bin(path_states[-1][2]).count("1")
    return cost, steps, leftovers
```

Hàm tìm kiếm A* Đây là phần trung tâm của chương trình. Hàm `astar_search()` tìm đường đi ngắn nhất từ trạng thái khởi đầu tới mục tiêu, sử dụng hàm heuristic do người dùng cung cấp. Nếu không có, mặc định dùng khoảng cách Chebyshev.

```
def astar_search(encoder:ProblemEncoder, max_expansions=200000,
                 heuristic_fn=None):
    if heuristic_fn is None:
        heuristic_fn = make_heuristic(chebyshev)

    start = encoder.initial_state()
    frontier = [] # Priority queue
    g_score = {start: 0}
    f0 = heuristic_fn(start, encoder)
    heapq.heappush(frontier, (f0, start))
    came_from = {start: None}
```

```
    expansions = 0
    t0 = time.time()

    while frontier:
        _, current = heapq.heappop(frontier)

        if encoder.goal_test(current):
            t_ms = (time.time() - t0)*1000.0
            path_states = reconstruct_path(came_from, current)
            total_cost, steps, leftovers = compute_plan_cost(encoder,
                                                             path_states)
            return {"name":"A*",
                    "path_states":path_states,
                    "cost":total_cost,
                    "expanded":expansions,
                    "steps":steps,
                    "time_ms":t_ms,
                    "leftovers":leftovers}

        expansions += 1
        if expansions > max_expansions:
            break

        for action in encoder.actions(current):
            nxt, step_cost = encoder.step(current, action)
            tentative_g = g_score[current] + step_cost

            if tentative_g < g_score.get(nxt, float('inf')):
                came_from[nxt] = current
                g_score[nxt] = tentative_g
                f = tentative_g + heuristic_fn(nxt, encoder)
                heapq.heappush(frontier, (f, nxt))

    t_ms = (time.time() - t0)*1000.0
    return {"name":"A*", "path_states":[], "cost":float('inf'),
            "expanded":expansions, "steps":0, "time_ms":t_ms,
            "leftovers":None}
```

3.1.8 Các hàm tiện ích

Hàm `generate_random_problem()` Hàm này tạo ngẫu nhiên một bài toán robot hút bụi trong lưới $m \times n$. Kết quả trả về gồm kích thước lưới, tập ô bẩn và tọa độ bắt đầu.

```
def generate_random_problem(m=10, n=10, dirty_count=5, seed=0):
    rnd = random.Random(seed)
    start = (rnd.randrange(n), rnd.randrange(m))
    dirty_positions = set()
    while len(dirty_positions) < dirty_count:
        p = (rnd.randrange(n), rnd.randrange(m))
        dirty_positions.add(p)
    return m, n, list(dirty_positions), start
```

Hàm `run_all_algorithms()` Thực thi, so sánh và đánh giá hiệu quả giữa các thuật toán tìm đường, bao gồm A* (với nhiều heuristic), Dijkstra, BFS và DFS.

```
def run_all_algorithms(m=10, n=10, dirty_count=5, seed=0,
    astar_heuristics: Iterable[str] = ("Chebyshev", "Manhattan",
                                       "Euclid", "Octile"),
    visualize=True):
    # ... (Implementation details omitted)
    pass
```

Hàm `plot_grid()` Hàm `plot_grid` dùng để trực quan hóa bản đồ môi trường và đường đi mà thuật toán tìm được. Hàm này vẽ lưới, tô màu các ô đặc biệt (*dirty*, *start*) và biểu diễn đường đi bằng các mũi tên.

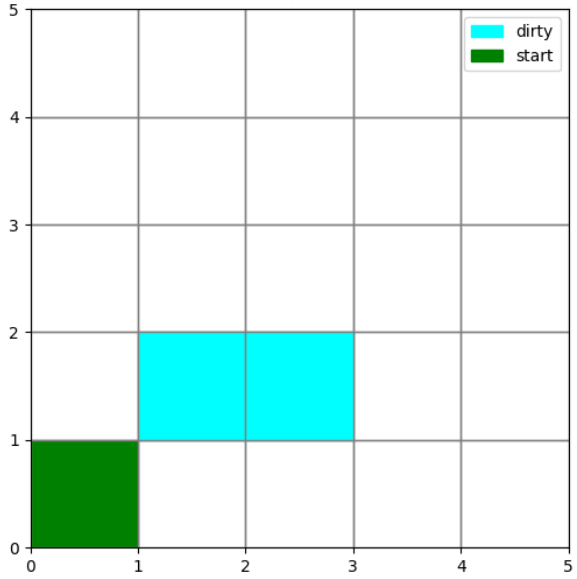
```
def plot_grid(grid, path, title="Path"):
    m, n = grid.m, grid.n
    plt.figure(figsize=(6, 6))
    # Vẽ lưới
    for y in range(m):
        for x in range(n):
            rect = plt.Rectangle((x, y), 1, 1,
            fill=False, edgecolor='gray')
```

```
            plt.gca().add_patch(rect)
    # Tô màu ô đặc biệt
    for y in range(m):
        for x in range(n):
            st = grid.status(x, y)
            if st == 'dirty':
                rect = plt.Rectangle((x, y), 1, 1,
                facecolor='cyan', fill=True)
                plt.gca().add_patch(rect)
            if st == 'start':
                rect = plt.Rectangle((x, y), 1, 1,
                facecolor='green', fill=True)
                plt.gca().add_patch(rect)
    # Vẽ đường đi
    for i in range(len(path)-1):
        x1, y1 = path[i]
        x2, y2 = path[i+1]
        px1, py1 = x1 + 0.5, y1 + 0.5
        px2, py2 = x2 + 0.5, y2 + 0.5
        dx, dy = px2 - px1, py2 - py1
        plt.arrow(px1, py1, dx*0.85, dy*0.85, head_width=0.15,
                  length_includes_head=True, color='blue')
    plt.xlim(0, n)
    plt.ylim(0, m)
    plt.title(title)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.grid(False)
    legend_patches = [
        mpatches.Patch(color='cyan', label='dirty'),
        mpatches.Patch(color='green', label='start')
    ]
    plt.legend(handles=legend_patches, loc='upper right')
    plt.show()
```

3.2 Phân tích hoạt động của thuật toán A*

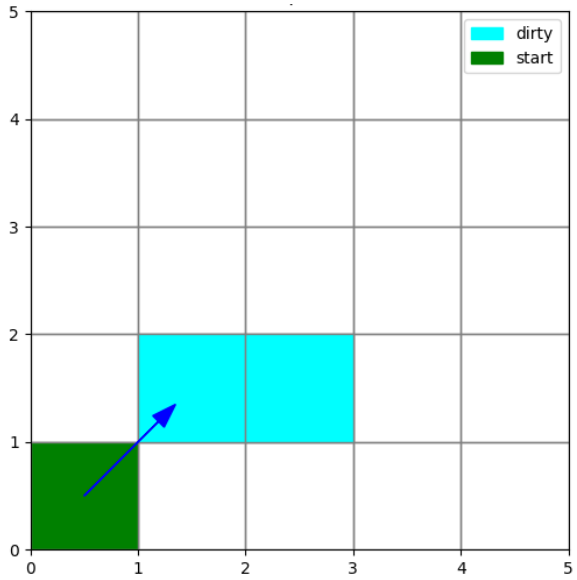
Để minh họa rõ hơn cơ chế hoạt động của thuật toán A*, nhóm đã thực hiện một mô phỏng trực quan trên lưới kích thước 5×5 với điểm xuất phát tại $(0, 0)$ và hai ô bẩn cần dọn là $(1, 1)$ và $(2, 1)$. Mỗi bước thực thi của thuật toán được ghi nhận cùng với trạng thái `OPEN LIST`, tập các đỉnh kề (neighbors), và biểu đồ trực quan hiển thị vị trí của robot cùng các ô bẩn còn lại.

Bước 1: Khởi tạo tại $(0, 0)$ Thuật toán bắt đầu tại vị trí $(0, 0)$ với $g = 0$, $h = 2$, $f = 2$. Các trạng thái láng giềng được sinh ra gồm $(0, 1)$, $(1, 0)$, và $(1, 1)$. Trong đó, trạng thái $(1, 1)$ có giá trị $f = 4$ nhỏ nhất, do đó sẽ được chọn để mở rộng ở bước kế tiếp. Các duyệt các láng giềng sẽ được mô tả ở bước 3.



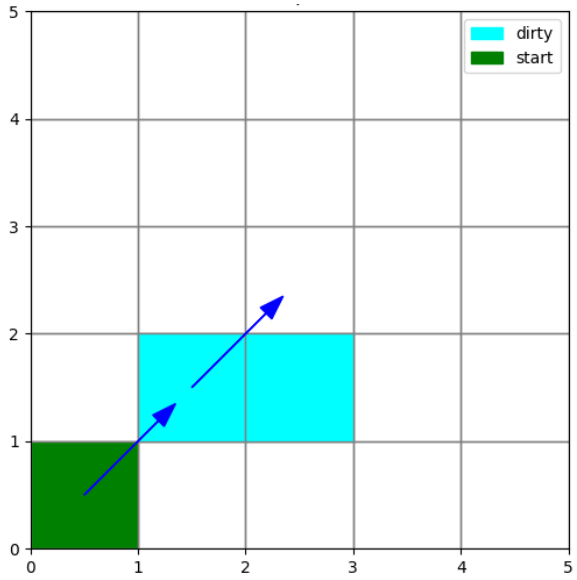
Hình 3.1: Khởi tạo lưới 5x5

Bước 2: Di chuyển đến $(1, 1)$ **và thực hiện hút bụi** Tại vị trí $(1, 1)$, robot đang đứng trên một ô bẩn. Thuật toán thực hiện hành động `suck`, chuyển trạng thái ô này từ "dirty" sang "clean". Giá trị chi phí được cập nhật $g = 3$, $h = 1$, $f = 4$. Các trạng thái láng giềng được sinh ra gồm $(0, 2)$, $(1, 2)$, $(2, 0)$, $(2, 1)$, $(2, 2)$, trong đó $(2, 1)$ được ưu tiên vì có giá trị f nhỏ nhất.



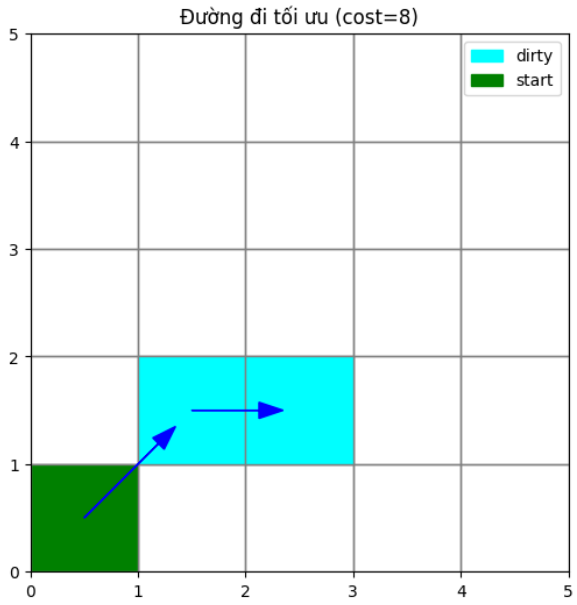
Hình 3.2: Robot di chuyển đến ô (1,1)

Bước 3: Duyệt các láng giềng kế cận $(1, 1)$ Thuật toán tiếp tục mở rộng các trạng thái có f bằng 8 trong `OPEN LIST`. Các bước này không tạo ra hành động "hút bụi" mới mà chỉ duyệt qua các hướng có thể di chuyển.



Hình 3.3: Dò các hướng đi khác

Bước 4: Quay lại ô $(1, 1)$ Sau khi duyệt hết `OPEN LIST`, robot quay về $(1, 1)$ và tiếp tục mở rộng theo hướng đến ô $(2, 1)$. Tại đây, giá trị chi phí được cập nhật $g = 5$, $h = 1$, $f = 6$.



Hình 3.4: Di chuyển đến (2,1)

Bước 5: Tìm thấy đích tại $(2, 1)$ Robot di chuyển đến ô bẩn cuối cùng $(2, 1)$ và thực hiện hành động `suck`. Tại thời điểm này, $g = 8$, $h = 0$, $f = 8$. Điều kiện dừng được thỏa mãn không còn ô bẩn nào, thuật toán kết thúc.

Kết quả đường đi tối ưu tìm được:

$$(0, 0) \rightarrow (1, 1) \rightarrow (2, 1)$$

với tổng chi phí $cost = 8$ và tổng số bước di chuyển $steps = 4$.

Chương 4

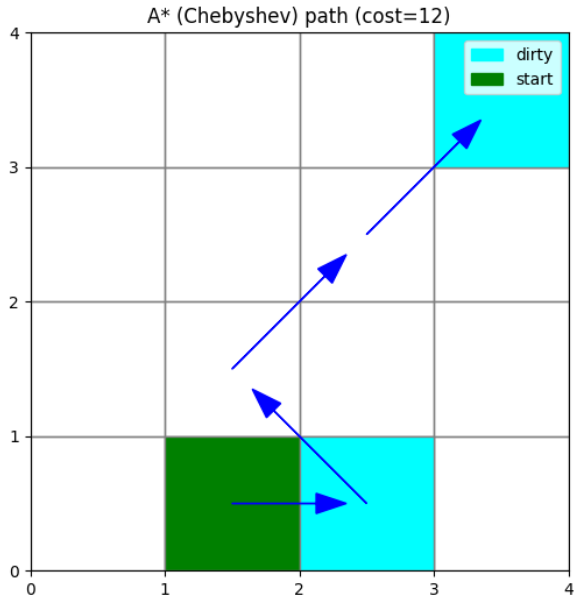
Đánh giá và thảo luận

4.1 Thiết kế so sánh

Để đánh giá hiệu quả của các thuật toán tìm đường trong bối cảnh bài toán Máy hút bụi, nhóm thực hiện ba thí nghiệm mô phỏng được thực hiện với kích thước lưới và số lượng ô bẩn khác nhau. Cụ thể, thí nghiệm thứ nhất sử dụng lưới nhỏ 4×4 với hai ô bẩn; thí nghiệm thứ hai mở rộng lên lưới 30×30 nhưng vẫn giữ hai ô bẩn; và thí nghiệm cuối cùng tăng số ô bẩn lên năm trên cùng kích thước 30×30. Các thuật toán được so sánh bao gồm A* với bốn loại hàm heuristic (Chebyshev, Manhattan, Euclid, Octile), cùng với các thuật toán cơ sở khác như Dijkstra, BFS và DFS. Mỗi thuật toán được đánh giá qua bốn tiêu chí: tổng chi phí di chuyển (Cost), số bước thực hiện (Steps), số đỉnh được mở rộng (Expanded) và thời gian thực thi (Time, tính bằng mili–giây).

Bảng 4.1: So sánh hiệu quả các thuật toán với $M = 4$, $N = 4$, $DIRTY = 2$

Thuật toán	Heuristic	Cost	Steps	Expanded	Time (ms)
A*	Chebyshev	12	6	26	0.55
A*	Manhattan	12	6	21	0.27
A*	Euclid	12	6	23	0.26
A*	Octile	12	6	22	0.28
Dijkstra	—	12	6	33	0.40
BFS	—	12	6	41	0.37
DFS	—	18	8	17	0.10

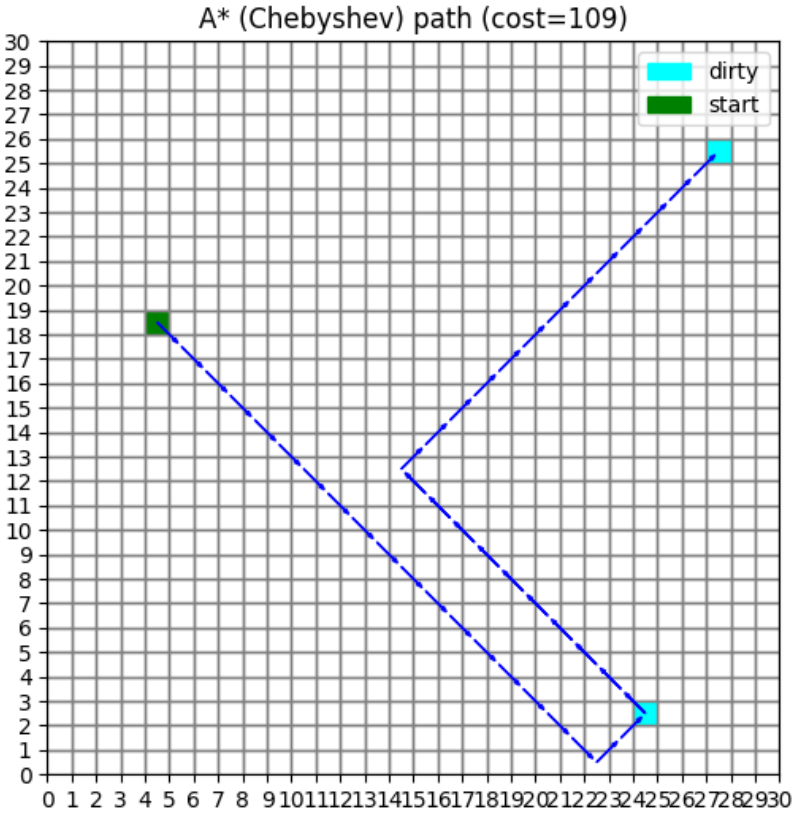


Hình 4.1: Kết quả Astar Chebyshev thí nghiệm 1

Trong môi trường nhỏ, tất cả các biến thể của A* đều đạt được chi phí tối ưu 12, trong khi DFS cho kết quả kém chính xác hơn với tổng chi phí 18. A* sử dụng khoảng cách Manhattan thể hiện hiệu quả cao nhất, đạt thời gian xử lý chỉ 0.27 ms và mở rộng ít đỉnh nhất (21). Dijkstra và BFS tuy vẫn tìm được đường đi tối ưu nhưng yêu cầu nhiều phép mở rộng hơn, lần lượt là 33 và 41 đỉnh.

Bảng 4.2: So sánh hiệu quả các thuật toán với $M = 30$, $N = 30$, $DIRTY = 2$

Thuật toán	Heuristic	Cost	Steps	Expanded	Time (ms)
A*	Chebyshev	109	45	1584	16.93
A*	Manhattan	109	45	1351	17.43
A*	Euclid	109	45	1502	15.72
A*	Octile	109	45	1468	16.25
Dijkstra	—	109	45	2180	16.86
BFS	—	109	45	2283	12.41
DFS	—	538	208	1542	9.84

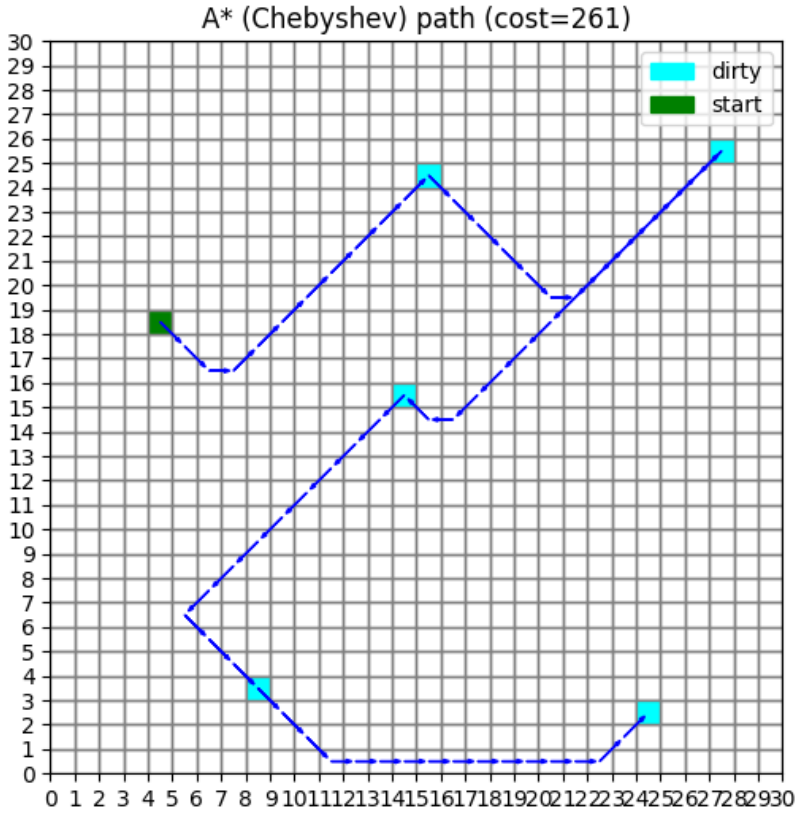


Hình 4.2: Kết quả Astar Chebyshev thí nghiệm 2

Khi kích thước lưới tăng lên, số đỉnh mở rộng của tất cả các thuật toán tăng đáng kể. Bốn biến thể của A* tiếp tục duy trì tính tối ưu với chi phí 109, trong đó A*–Manhattan và A*–Euclid cho số đỉnh mở rộng ít nhất (1,351 và 1,502). Dijkstra và BFS vẫn tìm được kết quả chính xác nhưng mở rộng nhiều hơn (trên 2,000 đỉnh). DFS không còn đảm bảo tính tối ưu, cho ra chi phí rất cao (538) dù thời gian thực thi ngắn nhất (9.84 ms), cho thấy đặc tính ưu tiên chiều sâu nhưng thiếu định hướng toàn cục.

Bảng 4.3: So sánh hiệu quả các thuật toán với $M = 30$, $N = 30$, $DIRTY = 5$

Thuật toán	Heuristic	Cost	Steps	Expanded	Time (ms)
A*	Chebyshev	261	69	13,267	395.13
A*	Manhattan	261	69	10,769	210.43
A*	Euclid	261	69	12,354	362.98
A*	Octile	261	69	12,057	310.07
Dijkstra	—	261	69	19,271	254.4
BFS	—	261	69	25,280	346.55
DFS	—	5,536	1,197	2,353	48.93



Hình 4.3: Kết quả Astar Chebyshev thí nghiệm 3

Trong môi trường lớn với năm ô bẩn, sự khác biệt giữa các thuật toán trở nên rõ rệt. A* với heuristic Manhattan tiếp tục thể hiện tốt nhất khi chỉ mở rộng 10,769 đỉnh, thấp hơn khoảng 45% so với BFS và 30% so với Dijkstra, đồng thời đạt thời gian xử lý 210.43 ms — nhanh nhất trong các biến thể A*. Ngược lại, DFS mất tính ổn định,

mở rộng ít hơn nhưng chi phí đường đi tăng đột biến (5,536), cho thấy không còn khả năng tối ưu khi không có hướng dẫn heuristic.

Nhìn chung, qua ba thí nghiệm mô phỏng, thuật toán A* luôn duy trì được tính chính xác và hiệu năng cao nhất, đặc biệt khi sử dụng hàm heuristic Manhattan hoặc Euclid. Các thuật toán tìm kiếm không có heuristic như BFS và Dijkstra vẫn đảm bảo kết quả đúng nhưng kém hiệu quả về mặt mở rộng và thời gian. DFS, tuy có ưu thế về tốc độ trong môi trường nhỏ, song không thích hợp cho các bài toán cần tính tối ưu toàn cục.

4.2 Hiệu quả của thuật toán A*

Độ phức tạp và khả năng tối ưu Thuật toán A* đảm bảo tính tối ưu khi hàm heuristic thoả mãn điều kiện chấp nhận được, tức giá trị ước lượng $h(n)$ không bao giờ vượt quá chi phí thực tế. Tuy nhiên, A* đòi hỏi bộ nhớ và thời gian tính toán lớn vì phải lưu trữ cả các đỉnh đang chờ (open set) và đã duyệt (closed set). Khi lựa chọn được hàm heuristic phù hợp, số lượng đỉnh mở rộng giảm đáng kể, từ đó rút ngắn thời gian tìm kiếm và nâng cao hiệu năng tổng thể.

Ứng dụng trong thực tế A* được ứng dụng rộng rãi trong nhiều lĩnh vực thực tế, bao gồm hệ thống định vị và tìm đường như Google Maps, điều hướng nhân vật trong trò chơi điện tử, và lập kế hoạch di chuyển cho các hệ thống robot hoặc xe tự hành. Nhờ khả năng tìm đường tối ưu với chi phí hợp lý, A* trở thành nền tảng cho nhiều thuật toán mở rộng như D*, D* Lite và Theta*.

Ứng dụng trong bài toán máy hút bụi Trong bài toán "Robot hút bụi", thuật toán A* giúp xác định chuỗi hành động ngắn nhất hoặc ít tốn kém nhất để dọn sạch toàn bộ không gian. Hàm heuristic định hướng robot ưu tiên di chuyển đến các ô bẩn gần nhất, giúp giảm số bước di chuyển thừa và tối ưu hoá trình tự làm sạch. Sự kết hợp giữa $f(n) = g(n) + h(n)$ cho phép robot vừa đạt hiệu quả năng lượng, vừa tiết kiệm thời gian dọn dẹp.

4.3 Hạn chế

Trong mô hình hiện tại, thuật toán A* được triển khai nhằm tìm đường đi ngắn nhất giữa hai điểm, sử dụng các hàm heuristic (Euclid, Manhattan, Chebyshev, Octile) cho

phép mô phỏng di chuyển theo 4 hoặc 8 hướng. Tuy nhiên, chương trình vẫn còn một số giới hạn khi mở rộng sang các bài toán thực tế phức tạp hơn.

Thứ nhất, đây là **bài toán đa mục tiêu (multi-goal problem)**. A* chỉ tối ưu cho một điểm đích duy nhất; khi có nhiều điểm bẩn, chương trình buộc phải chạy lại thuật toán cho từng mục tiêu, dẫn đến việc tổng quãng đường không đảm bảo tối ưu toàn cục và tăng thời gian xử lý đáng kể.

Thứ hai, **chi phí di chuyển chưa phản ánh đúng thực tế**. Mọi bước di chuyển đều được gán chi phí bằng 1, dù là di chuyển thẳng hay chéo, gây sai lệch giữa chi phí mô phỏng và năng lượng tiêu hao thực tế. Điều này làm giảm độ chính xác của tổng chi phí $f(n)$ trong các mô hình vật lý.

4.4 Hướng phát triển

Để mô phỏng chính xác hơn hành vi robot hút bụi trong các môi trường phức tạp, chương trình có thể được mở rộng theo các hướng sau:

(1) Mở rộng sang bài toán đa mục tiêu. Kết hợp A* với các thuật toán giải quyết bài toán người du lịch (Travelling Salesman Problem – TSP) hoặc các phương pháp heuristic lặp như Greedy A*, giúp xác định thứ tự làm sạch tối ưu giữa nhiều điểm bẩn.

(2) Bổ sung cơ chế chi phí di chuyển linh hoạt. Hiện tại, chi phí giữa các bước di chuyển là đồng nhất. Hướng phát triển tiếp theo là điều chỉnh hàm chi phí $g(n)$ để phân biệt bước thẳng (1) và bước chéo ($\sqrt{2}$), kết hợp cùng heuristic Octile để đạt độ chính xác cao hơn trong môi trường 8 hướng.

(3) Tăng tính thích ứng với môi trường động. Trong mô hình hiện tại, bản đồ được giả định tĩnh. Việc tích hợp các thuật toán như D* hoặc D* Lite sẽ giúp robot tự động cập nhật lộ trình khi phát hiện chướng ngại vật mới, mà không cần khởi động lại toàn bộ quá trình tìm kiếm.

(4) Cải thiện hiệu suất trên bản đồ lớn. Khi kích thước lưới tăng, chi phí bộ nhớ và thời gian xử lý tăng nhanh. Các kỹ thuật tối ưu như phân vùng bản đồ, heuristic nhiều tầng hoặc học tăng cường có thể được áp dụng để giảm tải và cải thiện hiệu năng.

Những hướng phát triển này giúp thuật toán A* tiến gần hơn đến các ứng dụng thực tế, nơi robot cần khả năng ra quyết định tối ưu, linh hoạt và tiết kiệm tài nguyên.

Chương 5

Kết luận

5.1 Kết quả của thuật toán A*

Thuật toán A* là cơ sở cốt lõi trong việc giải quyết bài toán "Máy hút bụi". Nhờ sự kết hợp giữa chi phí thực $g(n)$ và ước lượng $h(n)$, A* có thể xác định đường đi ngắn nhất để dọn sạch toàn bộ các ô bẩn một cách nhanh chóng và hiệu quả. Kết quả mô phỏng cho thấy A* không chỉ đảm bảo tối ưu về chi phí mà còn tiết kiệm đáng kể thời gian xử lý, đồng thời minh chứng khả năng ứng dụng rộng rãi của thuật toán trong các lĩnh vực như định vị, dẫn đường, và hệ thống robot tự động.

5.2 Những đóng góp của đồ án

Đồ án mang lại cả giá trị lý thuyết lẫn thực tiễn. Về mặt lý thuyết, đề tài đã hệ thống hoá nguyên lý hoạt động và các yếu tố ảnh hưởng đến hiệu quả của thuật toán A*. Về mặt ứng dụng, mô hình mô phỏng robot hút bụi đã minh chứng khả năng áp dụng thực tế của thuật toán này trong các hệ thống tự động. Ngoài ra, đồ án cũng đề xuất các hướng nghiên cứu tương lai như lựa chọn hàm heuristic tối ưu hơn, và mở rộng mô hình sang môi trường động nhằm tăng tính linh hoạt và ứng dụng trong thực tế.

Tài liệu tham khảo

Dantzig, G. B., Fulkerson, R., & Johnson, S. M. (1954). Solution of a Large-Scale Traveling-Salesman Problem. *Operations Research*, *2*(4), 393–410. https://doi.org/10.1287/opre.2.4.393

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, *1*, 269–271. https://doi.org/10.1007/BF01386390

Felner, A. (2011). Position paper: Applying A* in the real world. *AI Communications*, *24*(1), 3–17.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

IQ, O. (2023). *Euclidean vs Manhattan vs Chebyshev distance*. Retrieved October 24, 2025, from https://iq.opengenus.org/euclidean-vs-manhattan-vs-chebyshev-distance/

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press. http://lavalle.pl/planning/

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Russell, S., & Norvig, P. (2022). *Artificial Intelligence: A Modern Approach*. Pearson.

SpaceDev. (2024). *Thuật toán A* (A-star Algorithm)*. Retrieved October 24, 2025, from https://spacedev.vn/resources/docs/ai/search-algorithms/a-search

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.

Phụ lục

1. Thừa nhận sử dụng AI

Nhóm xác nhận có sử dụng ChatGPT và các công cụ AI khác trong quá trình biên soạn báo cáo, nhằm cải thiện tính mạch lạc, cấu trúc diễn đạt và hạn chế lỗi ngữ pháp. Việc sử dụng công cụ này chỉ nhằm mục đích hỗ trợ trình bày và đảm bảo tính rõ ràng của nội dung, hoàn toàn không can thiệp hay thay đổi các kết quả, số liệu, hay nội dung chuyên môn trong báo cáo.

Toàn bộ quá trình xây dựng mô hình, thiết kế thuật toán, lập trình, thu thập và xử lý dữ liệu, phân tích kết quả cũng như viết phần nội dung kỹ thuật đều được thực hiện trực tiếp bởi các thành viên trong nhóm.

2. Mã nguồn dự án và file kết quả các thí nghiệm

Toàn bộ mã nguồn của đồ án, bao gồm các tệp chương trình, dữ liệu thử nghiệm và kết quả mô phỏng, được nhóm lưu trữ công khai tại kho GitHub sau:

https://github.com/hoaianthai345/LTPTDL25_TENA.git

3. Phân công công việc

Thành viên	**Nội dung báo cáo**	**Code/Thực thi**
Thái Hoài An	Tổng hợp, định dạng báo cáo	Thuật toán Astar
Hoàng Thụy Hồng Ân	Phần 2.3, 4.1, 4.2	Thiết kế thí nghiệm so sánh
Dương Phương Anh	Phần 2.2, Chương 5	Thiết kế Class Grid

Nguyễn Đức Tuấn Anh	Phần 2.1, 3.1	Thiết kế các thuật toán khác
Nguyễn Thị Minh Anh	Chương 1, phần 3.2	Thiết kế hành vi robot hút bụi

4. Kiểm tra đạo văn

Kết quả kiểm tra đạo văn của báo cáo được tải lên kho Github ở trên.