TRƯỜNG ĐH SƯ PHẠM KỸ THUẬT TP.HCM

KHOA CÔNG NGHỆ THÔNG TIN

MÔN HỌC: TRÍ TUỆ NHÂN TẠO

ĐỒ ÁN CÁ NHÂN

TRÒ CHƠI 8 Ô TÍCH HỢP

CÁC THUẬT TOÁN TÌM KIẾM

GVHD: ThS. Phan Thị Huyền Trang

SVTH:

1. HÀ NHƯ QUỲNH 23110298

Mã lớp học: ARIN330585_04

Thành phố Hồ Chí Minh, tháng 05 năm 2025

MỤC LỤC

I. Giới thiệu đề tài

Tìm kiếm là một trong những kỹ thuật cốt lõi của trí tuệ nhân tạo, đóng vai trò quan trọng trong việc giải quyết các bài toán ra quyết định, lập kế hoạch, và định tuyến trong không gian trạng thái lớn. Trong lĩnh vực này, các thuật toán tìm kiếm được chia thành nhiều loại, từ những phương pháp không sử dụng thông tin tri thức (heuristic) như tìm kiếm theo chiều sâu, chiều rộng, đến những thuật toán tiên tiến hơn như A* có khả năng khai thác tri thức để tăng hiệu quả tìm kiếm.

Bài toán 8 ô là một trong những ví dụ kinh điển thường được sử dụng để minh họa và thử nghiệm các thuật toán tìm kiếm. Với không gian trạng thái vừa phải nhưng vẫn đủ phức tạp, bài toán này cho phép đánh giá trực quan hiệu suất và hành vi của các thuật toán trong quá trình tìm lời giải.

Đề tài này tập trung vào việc triển khai và so sánh các thuật toán tìm kiếm khác nhau trên bài toán 8 ô, bao gồm:

- Nhóm thuật toán tìm kiếm trong môi trường không có thông tin (BFS, DFS, UCS, IDS)
- Nhóm thuật toán tìm kiếm trong môi trường có thông tin (A, IDA*)*
- Nhóm thuật toán tìm kiếm ràng buộc (Backtracking)
- Nhóm thuật toán tìm kiếm trong môi trường không xác định (And Or Search)
- Nhóm thuật toán tìm kiếm cục bộ (Simple Hill Climbing, Steepest Ascent Hill Climbing, Stochastic Hill Climbing, Simulated Annealing, Beam Search, Genetic Algorithm)
- Nhóm thuật toán tìm kiếm học cũng cố (Q-learning)

Đề tài sẽ cung cấp cái nhìn tổng quan và định lượng về hiệu quả của từng thuật toán. Ngoài ra, quá trình tìm kiếm cũng được trực quan hóa nhằm làm nổi bật sự khác biệt trong chiến lược mở rộng trạng thái của từng phương pháp.

II. Mục tiêu đề tài

Bài tập này nhằm mục đích nghiên cứu và triển khai các thuật toán tìm kiếm trong lĩnh vực trí tuệ nhân tạo, bao gồm cả các phương pháp tìm kiếm không sử dụng thông tin heuristic và các thuật toán khai thác tri thức chuyên biệt của bài toán. Bên cạnh đó, bài tập còn mở rộng sang các chiến lược tìm kiếm cục bộ, kỹ thuật giải bài toán trong môi trường có ràng buộc, các phương pháp hoạt động hiệu quả trong môi trường phức tạp, học củng cố lẫn cả môi trường có và không có thông tin.

Trò chơi 8 ô được lựa chọn làm đối tượng thử nghiệm nhằm trực quan hóa và phân tích đặc điểm hoạt động của từng thuật toán. Thông qua việc triển khai và thực nghiệm trên bài toán này, người học có thể quan sát, so sánh và đánh giá hiệu quả của các thuật toán.

Kết quả thu được sẽ giúp làm rõ ưu – nhược điểm của từng thuật toán, đồng thời cung cấp cái nhìn thực tiễn về cách các chiến lược tìm kiếm vận hành trong những bài toán có không gian trạng thái lớn và phức tạp.

III. Nội dung

Trò chơi 8 ô thực chất là một ma trận 3x3 với các số từ 1-8 (số '0' được nhập từ bàn phím được hiểu là ô trắng), tập hành động có thể được thực thi và hợp lệ: Lên, xuống, hải trong phạm vi 1 ô - và tất cả các hành động đều có cùng mức chi phí (bằng 1 như chương trình đã định).

Đối với bài toán 8 ô, lời giải sẽ được định nghĩa là một chuỗi các trạng thái nối tiếp nhau, bắt đầu từ trạng thái bắt đầu và kết thúc tại trạng thái đích. Mỗi trạng thái thể hiện một cấu hình cụ thể của bảng 3x3 và được lưu trữ dưới dạng danh sách lồng danh sách, giúp dễ dàng quan sát và kiểm tra trong quá trình tìm kiếm.

1. Nhóm thuật toán tìm kiếm trong môi trường không có thông tin (BFS, DFS, UCS, IDS)

Thuật toán sẽ bắt đầu từ trạng thái bắt đầu và lần lượt mở rộng các trạng thái mới thông qua các hành động hợp lệ – cụ thể là việc di chuyển ô trống sang trái, phải, lên hoặc xuống (nếu còn nằm trong giới hạn ma trận). Trong phạm vi này, các thuật toán không sử dụng bất kỳ thông tin heuristic nào liên quan đến khoảng cách đến trạng thái mục tiêu. Thay vào đó, việc lựa chọn trạng thái kế tiếp phụ thuộc vào chiến lược duyệt cây hoặc chi phí hành động đã thực hiện.

Thành phần chính:

- Trạng thái: Ma trận 3x3 biểu diễn 8-puzzle.
- Trạng thái đầu: start_state
- Trạng thái đích: goal_state
- Hành động: get_states() – di chuyển ô trống theo 4 hướng
- Hàm chi phí: UCS dùng chi phí thực tế (g), các thuật toán còn lại dùng chi phí đơn vị (1 bước = 1).
- Chiến lược mở rộng
  - BFS: theo hàng đợi FIFO
  - DFS: theo ngăn xếp LIFO
  - IDS: duyệt sâu tăng dần
  - UCS: hàng đợi ưu tiên theo chi phí

Solution: Trả về danh sách các trạng thái từ start_state đến goal_state.

Các thuật toán được sử dụng:

- **Breadth-First Search (BFS)**: Duyệt theo mức độ, mở rộng tất cả các nút ở một độ sâu trước khi đi sâu hơn. Ưu điểm lớn nhất là đảm bảo tìm được lời giải ngắn nhất về số bước đi. Tuy nhiên, nhược điểm là tốn nhiều bộ nhớ do cần lưu trữ toàn bộ nút ở mỗi mức.
  - Độ phức tạp thời gian và không gian: O(b^d), trong đó b là bậc phân nhánh (branching factor) và d là độ sâu của lời giải.
- **Depth-First Search (DFS):** Mở rộng theo chiều sâu trước, đi sâu vào một nhánh cho đến khi gặp nút không còn con hoặc tìm được lời giải. Phương pháp này thường nhanh hơn trong một số trường hợp, nhưng dễ rơi vào vòng lặp hoặc đi sâu vô ích nếu không có cơ chế kiểm soát độ sâu hoặc kiểm tra trạng thái đã duyệt.
  - Độ phức tạp thời gian: O(b^d).

- Độ phức tạp không gian: O(d).

- **Uniform-Cost Search (UCS)**: Là một biến thể của BFS, nhưng thay vì duyệt theo mức độ, UCS mở rộng các trạng thái dựa trên chi phí thực tế đã bỏ ra để đến được trạng thái đó. Nó đảm bảo tìm được lời giải tối ưu nếu chi phí hành động dương và đồng nhất hoặc không âm. Ưu điểm của nó là cho phép tính đến chi phí nếu có các hành động không đồng đều.

    - Độ phức tạp thời gian: O(b^d), tương tự BFS.

- **Iterative Deepening Search (IDS)**: Kết hợp ưu điểm của BFS và DFS bằng cách thực hiện DFS nhiều lần với độ sâu giới hạn tăng dần. Mỗi vòng lặp là một DFS với độ sâu tối đa mới. Phương pháp này đảm bảo tìm được lời giải ngắn nhất như BFS nhưng tốn ít bộ nhớ hơn.

    - Độ phức tạp thời gian: O(b^d) (do lặp lại các nút).

    - Độ phức tạp không gian: O(d).

So sánh giữa các thuật toán:

Thuật toán	Chiến lược duyệt	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
BFS	Theo mức (level-order)	Tìm được lời giải ngắn nhất (về số bước)	Tốn bộ nhớ nhiều	O(b^d)	O(b^d)
DFS	Theo chiều sâu	Bộ nhớ thấp	Không tối ưu, dễ lặp	O(b^d)	O(d)
UCS	Theo chi phí thấp nhất	Tìm được lời giải tối ưu (nếu có chi phí khác	Tốn bộ nhớ, chậm nếu không có khác	O(b^d)	O(b^d)

Thuật toán	Chiến lược duyệt	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
		nhau)	biệt chi phí		
IDS	DFS giới hạn độ sâu tăng dần	Tối ưu và tiết kiệm bộ nhớ	Phải lặp lại nhiều lần	O(b^d)	O(d)

Nhận xét:

Trong quá trình thử nghiệm trò chơi với các thuật toán tìm kiếm cơ bản như BFS, DFS, UCS và IDS, mỗi phương pháp đều thể hiện những đặc trưng riêng về hiệu quả, độ tối ưu và mức tiêu thụ tài nguyên. Thuật toán BFS có ưu điểm là luôn tìm được lời giải ngắn nhất nếu tồn tại, nhờ vào cách duyệt theo từng mức độ. Tuy nhiên, nhược điểm lớn nhất của BFS là tiêu tốn rất nhiều bộ nhớ, do phải lưu trữ toàn bộ các trạng thái cùng cấp trong không gian tìm kiếm. Trái lại, DFS lại sử dụng rất ít bộ nhớ, đi sâu vào từng nhánh nên có thể cho kết quả nhanh trong một số trường hợp. Tuy nhiên, DFS không đảm bảo tìm ra lời giải ngắn nhất và rất dễ rơi vào vòng lặp nếu không có cơ chế kiểm soát độ sâu hay trạng thái đã duyệt.

UCS là một biến thể của BFS có xét đến chi phí thực tế của từng hành động. UCS đảm bảo tìm ra lời giải tối ưu trong trường hợp chi phí các bước di chuyển khác nhau, nhưng đối với trò chơi này (nơi mọi bước đều có chi phí bằng nhau), UCS không mang lại ưu thế rõ rệt so với BFS mà còn chậm hơn do phải quản lý hàng đợi ưu tiên. Cuối cùng, IDS - được đánh giá là thuật toán kết hợp được điểm mạnh của cả BFS và DFS. IDS vẫn đảm bảo tìm lời giải ngắn nhất như BFS nhưng lại tiết kiệm bộ nhớ gần như DFS. Dù có sự lặp lại các nút ở những mức sâu thấp hơn, IDS tỏ ra rất phù hợp cho những bài toán như 8-Puzzle, nơi độ sâu lời giải không quá lớn.

Tóm lại, lựa chọn thuật toán phù hợp phụ thuộc vào ưu tiên giữa tốc độ, độ tối ưu của lời giải và khả năng xử lý bộ nhớ. Trong trường hợp cần lời giải ngắn và dung lượng bộ nhớ không quá hạn chế, BFS là lựa chọn phù hợp. Nếu cần tiết kiệm bộ nhớ và vẫn giữ được độ chính xác, IDS là phương án cân bằng tốt nhất. DFS và UCS có thể dùng để minh họa về giới hạn của các chiến lược tuyến tính hoặc có trọng số, nhưng không phải là lựa chọn lý tưởng trong hầu hết trường hợp của bài toán này.

2. Nhóm thuật toán tìm kiếm trong môi trường có thông tin

Trong môi trường có thông tin, quá trình tìm kiếm được hỗ trợ bởi các hàm heuristic – các hàm đánh giá ước lượng chi phí từ trạng thái hiện tại đến trạng thái mục tiêu. Nhờ đó, thuật toán có thể ưu tiên mở rộng những trạng thái "hứa hẹn" hơn, tức là có khả năng dẫn đến lời giải nhanh hơn hoặc với chi phí thấp hơn. Việc lựa chọn trạng thái kế tiếp không còn chỉ phụ thuộc vào thứ tự duyệt hay chi phí đã đi, mà còn dựa trên ước lượng thông minh về khoảng cách còn lại đến đích. Điều này giúp cải thiện hiệu suất và giảm đáng kể số trạng thái cần duyệt so với các chiến lược tìm kiếm mù.

Thành phần chính

- Trạng thái: 8-puzzle
- Hành động: get_states()
- Heuristic: manhattan_distance()
- Hàm đánh giá:
  - A*: f(n) = g(n) + h(n)
  - IDA*: sử dụng f tương tự
  - Greedy: chỉ dùng h(n)
- Trạng thái mục tiêu: goal_state

Solution: Trả về đường đi tối ưu hoặc gần tối ưu.

Các thuật toán được sử dụng:

- **A***: A* là thuật toán tìm kiếm sử dụng thông tin heuristic để dẫn dắt quá trình mở rộng trạng thái. Nó tính toán tổng chi phí f(n) = g(n) + h(n), trong đó g(n) là chi phí từ trạng thái ban đầu đến hiện tại, và h(n) là ước lượng chi phí còn lại đến mục tiêu (heuristic). A* đảm bảo tìm được lời giải tối ưu nếu h(n) là hàm heuristic chấp nhận được (không đánh giá thấp chi phí thực). Ưu điểm của thuật toán là tìm được lời giải tối ưu, ít duyệt trạng thái dư thừa nếu heuristic tốt.
  - Độ phức tạp thời gian: O(b^d) trong trường hợp xấu.

- Độ phức tạp không gian: O(b^d).

- **IDA*:** IDA* kết hợp giữa A* và DFS lặp sâu. Thay vì dùng hàng đợi ưu tiên như A*, IDA* sử dụng tìm kiếm theo chiều sâu nhưng có giới hạn f(n) (chi phí tổng hợp). Giới hạn này sẽ được tăng dần sau mỗi vòng lặp. IDA* giữ được tính tối ưu của A* nhưng tiết kiệm bộ nhớ như DFS. Ưu điểm của thuật toán là tối ưu, tiết kiệm bộ nhớ, phù hợp với không gian tìm kiếm lớn.

  - Độ phức tạp thời gian: O(b^d) (do lặp lại).

  - Độ phức tạp không gian: O(d).

- **Greedy Search:** Đây là thuật toán tìm kiếm sử dụng thông tin heuristic để ưu tiên mở rộng các trạng thái có giá trị h(n) thấp nhất, tức là gần mục tiêu nhất theo ước lượng. Không giống như A*, thuật toán này bỏ qua chi phí đã đi (g(n)) và chỉ quan tâm đến ước lượng còn lại. Do đó, Greedy có thể rất nhanh nếu heuristic tốt, nhưng không đảm bảo tìm ra lời giải tối ưu.

  - Độ phức tạp thời gian: O(b^d), tương tự A*, nhưng thường nhanh hơn do ít xét g(n).

  - Độ phức tạp không gian: O(b^d), do vẫn cần lưu các nút đang mở.

So sánh giữa các thuật toán:

Thuật toán	Chiến lược duyệt	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
A*	Theo f(n) = g(n) + h(n)	Tối ưu, thông minh nếu heuristic tốt	Tốn bộ nhớ nhiều	O(b^d)	O(b^d)
IDA*	DFS với giới hạn f(n) tăng	Tối ưu, tiết kiệm bộ nhớ	Phải lặp lại, hiệu quả phụ thuộc vào h(n)	O(b^d)	O(d)

Thuật toán	Chiến lược duyệt	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
	dần				
Greedy	Ưu tiên theo h(n) nhỏ nhất	Rất nhanh nếu heuristic đúng hướng	Không đảm bảo tối ưu, dễ bị "lạc hướng"	O(b^d)	O(b^d)

Nhận xét:

A* là một trong những thuật toán mạnh mẽ nhất trong các chiến lược tìm kiếm có sử dụng thông tin heuristic. Nhờ việc kết hợp chi phí thực (g(n)) và chi phí ước lượng (h(n)), A* có khả năng định hướng quá trình tìm kiếm theo hướng tối ưu nhất, giảm số lượng trạng thái cần duyệt so với các thuật toán không heuristic như BFS hoặc UCS. Tuy nhiên, điểm yếu lớn nhất của A* là mức tiêu thụ bộ nhớ rất cao, do cần lưu toàn bộ trạng thái mở rộng và sắp xếp chúng trong hàng đợi ưu tiên.

Ngược lại, IDA* khắc phục được vấn đề bộ nhớ bằng cách thay thế hàng đợi ưu tiên bằng tìm kiếm theo chiều sâu có giới hạn f(n). Với đặc điểm này, IDA* vẫn đảm bảo tìm được lời giải tối ưu như A*, nhưng tiết kiệm đáng kể tài nguyên bộ nhớ, đặc biệt phù hợp với các bài toán có không gian trạng thái lớn hoặc thiết bị hạn chế tài nguyên. Tuy vậy, do không lưu trữ trạng thái trung gian, IDA* phải lặp lại quá trình mở rộng nhiều lần nên có thể mất nhiều thời gian hơn, nhất là khi heuristic không đủ chính xác.

Tóm lại, nếu bộ nhớ không phải là vấn đề và heuristic được thiết kế tốt, A* là lựa chọn hàng đầu nhờ khả năng tìm lời giải nhanh và tối ưu. Trong khi đó, nếu làm việc với môi trường giới hạn tài nguyên, IDA* là một giải pháp hợp lý để duy trì sự tối ưu mà không đánh đổi quá nhiều về bộ nhớ.

3. Nhóm thuật toán tìm kiếm ràng buộc

Trong nhóm thuật toán tìm kiếm có ràng buộc, mục tiêu chính là tìm lời giải thỏa mãn một tập hợp các điều kiện nhất định bằng cách duyệt qua không gian trạng thái theo hướng có thể quay lui khi phát hiện trạng thái không hợp lệ. Phương pháp

này rất phổ biến trong các bài toán tổ hợp hoặc sắp xếp, nơi cần loại bỏ sớm các nhánh sai để giảm tải không gian tìm kiếm.

Đối với bài toán 8 ô, Backtracking có thể được áp dụng như một chiến lược duyệt sâu kết hợp với điều kiện kiểm tra trạng thái để tránh lặp vô hạn. Phương pháp này xây dựng lời giải bằng cách thử từng nước đi một cách tuần tự. Nếu tại một trạng thái không thể tiếp tục hợp lệ, thuật toán sẽ quay lui (backtrack) về trạng thái trước đó để thử hướng đi khác.

Thành phần chính:

- Trạng thái: 8-puzzle
- Hàm ràng buộc: Không trùng trạng thái, độ sâu giới hạn, forward checking (manhattan_distance + depth <= max_depth)
- Hành động: apply_action()
- Chiến lược:
  - Đệ quy quay lui (backtracking)
  - Forward checking
  - min-conflicts

Solution: Dãy trạng thái thoả mãn ràng buộc đến goal_state

Thuật toán được sử dụng:

- **Backtracking:** Phương pháp Backtracking là một chiến lược tìm kiếm theo chiều sâu, trong đó thuật toán sẽ quay lui mỗi khi gặp trạng thái không hợp lệ hoặc không còn khả năng mở rộng. Một đặc điểm quan trọng của Backtracking là việc áp dụng các điều kiện ràng buộc để loại bỏ sớm những nhánh không khả thi, chẳng hạn như không cho phép lặp lại các trạng thái đã duyệt. Điều này giúp giảm thiểu không gian tìm kiếm đáng kể.
  - Độ phức tạp thời gian: Trong trường hợp xấu nhất là O(b^d), với b là số hành động hợp lệ từ một trạng thái và d là độ sâu tối đa.
  - Độ phức tạp không gian: O(d).
- **Backtracking with Forward Checking:** Là một cải tiến của phương pháp Backtracking truyền thống, thường được áp dụng trong bài toán ràng buộc

(Constraint Satisfaction Problems - CSP). Thuật toán này không chỉ kiểm tra ràng buộc tại thời điểm gán biến, mà còn dự đoán trước xem việc gán đó có khiến cho các biến chưa gán còn lại bị ràng buộc đến mức không còn giá trị hợp lệ nào để chọn hay không. Nếu có, thuật toán sẽ quay lui ngay lập tức, nhờ đó tiết kiệm thời gian tìm kiếm.

Tóm tắt thuật toán:

Thuật toán	Chiến lược duyệt	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
Backtracking	DFS có kiểm tra ràng buộc	Bộ nhớ thấp, loại bỏ nhánh sai sớm	Không đảm bảo tìm lời giải ngắn nhất, dễ lặp nếu không kiểm tra trạng thái	O(b^d)	O(d)
Backtracking + Forward Checking	DFS + Kiểm tra ràng buộc và miền giá trị	Hiệu quả hơn Backtracking thuần, tránh mở rộng nhánh sai rõ ràng	Tốn thêm tài nguyên để theo dõi và cập nhật miền giá trị	O(b^d) (nhưng giảm nhiều trên thực tế)	O(d + n·m) (n biến, m miền)

Nhận xét:

Mặc dù Backtracking không phải là thuật toán tối ưu cho mọi bài toán tìm kiếm, nhưng trong phạm vi trò chơi 8 ô – nơi không gian trạng thái tuy lớn nhưng có thể phát

hiện sớm các nhánh sai – nó vẫn là một công cụ minh họa hữu ích cho tư duy giải quyết theo chiều sâu và loại trừ. Tuy nhiên, nếu không có cơ chế ghi nhớ trạng thái đã duyệt (memoization hoặc visited set), thuật toán này rất dễ bị rơi vào vòng lặp, dẫn đến hiệu năng kém. Về mặt bộ nhớ, Backtracking là một trong những phương pháp tiết kiệm nhất, do chỉ cần lưu trữ đường đi hiện tại. Tuy nhiên, do không có chiến lược ưu tiên rõ ràng hoặc hướng dẫn bởi heuristic, việc tìm ra lời giải ngắn nhất là không đảm bảo, và thời gian chạy có thể không khả thi nếu độ sâu giải pháp quá lớn.

4. Nhóm thuật toán tìm kiếm trong môi trường không xác định

Trong một số bài toán, không gian tìm kiếm không chỉ đơn thuần là cây hay đồ thị với các lựa chọn tách biệt (OR-node), mà còn tồn tại những tình huống đòi hỏi phải đồng thời thỏa mãn nhiều điều kiện con để đạt được mục tiêu – đây chính là đặc trưng của AND-node. Để xử lý hiệu quả những cấu trúc phức tạp này, người ta sử dụng AND/OR Search – một chiến lược mở rộng của tìm kiếm truyền thống, cho phép biểu diễn linh hoạt cả các tình huống lựa chọn (OR) và kết hợp (AND).

Sự phức tạp càng tăng khi tác nhân hoạt động trong môi trường không đầy đủ thông tin – ví dụ như trong Sensorless Search, nơi tác nhân không biết chính xác trạng thái hiện tại, hoặc trong Partial Observation, nơi chỉ có thể quan sát một phần của trạng thái. Trong các môi trường này, tác nhân không chỉ cần lên kế hoạch trong một không gian trạng thái đầy đủ mà còn phải duy trì và cập nhật một tập hợp trạng thái niềm tin (belief state)

Thành phần chính:

- Trạng thái: 8-puzzle
- Hành động and_or_moves (up, down, left, right)
- Cấu trúc AND/OR: Gọi đệ quy theo nhiều nhánh và điều kiện
- Giới hạn độ sâu: max_depth

Solution: Một plan (dạng cây nhánh AND/OR) đưa tới goal_state, hoặc None nếu không đạt.

Thuật toán được sử dụng:

- **AND/OR Search:** Đây là thuật toán tìm kiếm trong các không gian trạng thái có cấu trúc phân rẽ theo kiểu cây hỗn hợp giữa AND và OR. Với mỗi nút, thuật toán xác định nó là loại AND hay OR, sau đó lần lượt giải quyết các nhánh theo logic tương ứng. Nếu là OR-node, chỉ cần tìm một lời giải trong các nhánh con; nếu là AND-node, cần giải quyết tất cả các nhánh con. Trong quá trình tìm kiếm, thuật toán cũng áp dụng cơ chế quay lui và loại bỏ nhánh không khả thi.

    - Độ phức tạp thời gian: Phụ thuộc vào cấu trúc cây AND/OR, trong trường hợp xấu nhất vẫn là O(b^d) nhưng thường nhỏ hơn do loại trừ được nhiều nhánh từ sớm.

    - Độ phức tạp không gian: O(d).

- **Sensorless Search:** Là dạng tìm kiếm trong môi trường không có thông tin cảm biến, tức là tác nhân không thể quan sát trạng thái hiện tại một cách chính xác. Thay vào đó, nó giữ một tập hợp các trạng thái có thể và thực hiện các hành động nhằm giảm dần tập hợp này, với mục tiêu cuối cùng là thu hẹp về một trạng thái duy nhất hoặc đạt được mục tiêu bất kể trạng thái ban đầu chính xác là gì.

    - **Mục tiêu:** Tìm chuỗi hành động sao cho dù bắt đầu ở trạng thái nào trong tập hợp ban đầu, tác nhân vẫn đảm bảo đạt được mục tiêu.

- **Partial Observation:** **T**ác nhân có một phần thông tin về trạng thái hiện tại, thường thông qua cảm biến hạn chế hoặc mập mờ. Tác nhân cần sử dụng suy luận logic và thông tin lịch sử để cập nhật niềm tin về trạng thái thực sự của thế giới. Mục tiêu: Duy trì một mô hình belief state (tập hợp hoặc phân phối xác suất các trạng thái có thể xảy ra).

Tóm tắt thuật toán:

Thuật toán	**Chiến lược duyệt**	**Ưu điểm**	**Nhược điểm**	**Độ phức tạp thời gian**	**Độ phức tạp không gian**
AND/OR Search	DFS có phân	Mô hình hóa được các bài	Không đơn giản để cài	O(b^d) (trong	O(d)

Thuật toán	Chiến lược duyệt	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
	nhánh AND và OR	toán phân rã phức tạp, tiết kiệm bộ nhớ	đặt, khó trực quan hóa lời giải	TH xấu)	

Nhận xét:

Thuật toán AND/OR Search phù hợp với những bài toán có cấu trúc mục tiêu con phụ thuộc lẫn nhau, ví dụ như các hệ thống lập kế hoạch đa bước hoặc trò chơi chiến lược. Trong khi tìm kiếm thông thường chỉ cần xét đến một hành động tiếp theo tại mỗi bước, thì AND/OR Search yêu cầu một cái nhìn bao quát hơn để giải quyết toàn bộ tập con các mục tiêu trong một số tình huống. Thuật toán không đảm bảo tìm lời giải tối ưu nếu không có sự hỗ trợ của heuristic hoặc chi phí. Trong bối cảnh trò chơi đơn giản như 8 ô, AND/OR Search ít khi được sử dụng trực tiếp, nhưng về mặt lý thuyết, nó là một mô hình tìm kiếm mạnh cho các môi trường cần xử lý các mục tiêu ràng buộc đồng thời hoặc hệ thống phân cấp.

5. Nhóm thuật toán tìm kiếm cục bộ

Tìm kiếm cục bộ là nhóm thuật toán thường được sử dụng khi không gian trạng thái quá lớn để có thể duyệt toàn bộ. Thay vì duyệt từ trạng thái gốc qua các bước trung gian cho đến trạng thái mục tiêu như các phương pháp tìm kiếm truyền thống, thuật toán tìm kiếm cục bộ chỉ duy trì một (hoặc vài) trạng thái hiện tại, và cải tiến nó thông qua việc tìm trạng thái "lân cận" tốt hơn. Nhóm thuật toán này thường được áp dụng cho các bài toán tối ưu hóa hoặc bài toán không có cấu trúc rõ ràng về đường đi, như bài toán tối ưu hàm, bài toán người du lịch, hoặc sắp xếp lịch trình.

Khác với nhóm không có thông tin, tìm kiếm cục bộ sử dụng heuristic để đánh giá mức độ tốt của trạng thái hiện tại nhằm hướng dẫn quá trình tìm kiếm đến lời giải tối ưu.

Thành phần chính:

- Trạng thái: 8-puzzle

- Hàm đánh giá: manhattan_distance()
- Chiến lược:
  - Leo đồi (tăng dần điểm)
  - Tìm điểm tối ưu cục bộ
  - Giải pháp quần thể (GA)

Solution: Trả về đường đi (chuỗi trạng thái) từ start_state đến goal_state, nhưng có thể không tối ưu toàn cục.

Các thuật toán được sử dụng:

- **Simple Hill Climbing:** Tại mỗi bước, thuật toán chọn trạng thái lân cận đầu tiên tốt hơn trạng thái hiện tại và chuyển đến nó. Nếu không có trạng thái nào tốt hơn, thuật toán dừng lại. Dễ cài đặt, nhanh với không gian tìm kiếm nhỏ.
  - Độ phức tạp thời gian: O(b × m)
  - Độ phức tạp không gian: O(1)
- **Steepest Ascent Hill Climbing:** Giống như Hill Climbing đơn giản, nhưng tại mỗi bước, thuật toán xét tất cả các trạng thái lân cận, chọn ra trạng thái tốt nhất và chuyển đến đó. Điều này giúp tránh rơi vào bẫy sớm hơn. Cải thiện so với Simple Hill Climbing.
  - Độ phức tạp thời gian: O(b × m)
  - Độ phức tạp không gian: O(1)
- **Stochastic Hill Climbing:** Tương tự Steepest Ascent, nhưng chọn ngẫu nhiên một trong các trạng thái lân cận tốt hơn. Điều này giúp tăng khả năng thoát khỏi vùng bằng hoặc local maximum. Tránh bẫy tốt hơn nhờ yếu tố ngẫu nhiên.
  - Độ phức tạp thời gian: O(b × m)
  - Độ phức tạp không gian: O(1)

- **Simulated Annealing:** Thuật toán cho phép chọn các trạng thái lân cận kém hơn với xác suất giảm dần theo thời gian (dựa trên tham số nhiệt độ – temperature). Nhờ đó, nó có thể vượt qua các cực trị cục bộ và có xác suất tiến đến lời giải tối ưu toàn cục. Có khả năng tìm lời giải tốt hơn trong không gian phức tạp.
  - Độ phức tạp thời gian: O(b × m) (trong T vòng lặp)
  - Độ phức tạp không gian: O(1)
- **Beam Search:** Giống như BFS nhưng chỉ giữ lại k trạng thái tốt nhất ở mỗi mức, giới hạn độ rộng tìm kiếm. Điều này giúp tiết kiệm bộ nhớ và thời gian hơn BFS. Hiệu quả hơn BFS trong không gian lớn.
  - Độ phức tạp thời gian: O(k × b × d)
  - Độ phức tạp không gian: O(k × b)
- **Genetic Algorithm:** Dựa trên tiến hóa sinh học, thuật toán khởi tạo một tập hợp các lời giải (quần thể), áp dụng các phép crossover và mutation để tạo ra lời giải mới. Sau mỗi thế hệ, những lời giải tốt nhất được chọn lọc để tiếp tục quá trình tiến hóa. Phù hợp với các bài toán tối ưu phức tạp.
  - Độ phức tạp thời gian: O(p × g × c)
  - Độ phức tạp không gian: O(p)

So sánh giữa các thuật toán tìm kiếm cục bộ:

Thuật toán	Chiến lược chính	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
Simple Hill Climbing	Chọn trạng thái lân cận đầu tiên tốt hơn	Đơn giản, dễ triển khai	Mắc kẹt ở cực trị cục bộ	O(b × m)	O(1)

Thuật toán	Chiến lược chính	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
Steepest Ascent Hill Climbing	Xét tất cả lân cận, chọn tốt nhất	Chính xác hơn Simple HC	Tốn công xét toàn bộ lân cận	O(b × m)	O(1)
Stochastic Hill Climbing	Chọn ngẫu nhiên trong các lân cận tốt hơn	Tránh bẫy nhờ yếu tố ngẫu nhiên	Kết quả không ổn định	O(b × m)	O(1)
Simulated Annealing	Cho phép chọn trạng thái xấu theo xác suất	Có thể đạt cực trị toàn cục	Cần tinh chỉnh tham số nhiệt độ	O(b × m) (trong T vòng lặp)	O(1)
Beam Search	Duyệt song song nhiều trạng thái (k trạng thái)	Cân bằng BFS và tiết kiệm tài nguyên	Có thể bỏ sót lời giải tốt	O(k × b × d)	O(k × b)
Genetic Algorithm	Tiến hóa dựa trên quần thể, lai và đột biến	Hiệu quả với bài toán phức tạp	Tham số phức tạp, không đảm bảo tối ưu toàn cục	O(p × g × c)	O(p)

- **Ghi chú về các ký hiệu:**

- b: số lượng trạng thái lân cận (branching factor)
- m: số bước tối đa (số lần cải tiến)
- d: độ sâu của lời giải
- k: số lượng trạng thái được giữ lại ở mỗi mức trong Beam Search
- T: số vòng lặp tối đa (Simulated Annealing)
- b: kích thước quần thể (population) trong Genetic Algorithm
- g: số thế hệ (generations)
- c: số cá thể con được tạo ra mỗi thế hệ

Nhận xét:

Tìm kiếm cục bộ là công cụ mạnh mẽ khi không gian tìm kiếm quá lớn hoặc không thể xây dựng đường đi rõ ràng giữa các trạng thái. Tuy nhiên, hiệu quả của các thuật toán trong nhóm này phụ thuộc nhiều vào đặc tính của bài toán, cách thiết kế hàm đánh giá (heuristic) và các tham số điều chỉnh.

6. Nhóm học củng cố

Q-learning là một thuật toán học củng cố cho phép tác nhân học cách hành động trong môi trường bằng cách tương tác và rút ra chiến lược tối ưu dựa trên phần thưởng nhận được. Thay vì duyệt toàn bộ không gian trạng thái như các thuật toán tìm kiếm truyền thống, Q-learning học dần qua trải nghiệm, giúp nó áp dụng tốt cho các môi trường không biết trước hoặc có tính ngẫu nhiên.

Trong trò chơi 8 ô, Q-learning có thể áp dụng bằng cách ánh xạ mỗi trạng thái thành một giá trị Q cho từng hành động có thể thực hiện. Thông qua quá trình thử-sai, sẽ cập nhật bảng Q để học ra chiến lược tốt nhất nhằm đạt đến trạng thái đích.

Thành phần chính:

- Trạng thái: 8-puzzle
- Hành động: actions_q = ['U','D','L','R']

- Hàm giá trị: Bảng Q: Q[state][action]
- Chiến lược học Q-Learning: Cập nhật theo công thức Q
- Chính sách chọn hành động: choose_action() – theo epsilon-greedy

Solution: Sau khi học (train_q_learning()), lời giải là đường đi tốt nhất từ trạng thái ban đầu đến goal_state_q, được tạo bằng hàm play().

Thuật toán Q-Learning:

- Khởi tạo bảng Q với các giá trị bằng 0 hoặc ngẫu nhiên.
- Ở mỗi trạng thái hiện tại s, chọn hành động a theo chính sách ε-greedy (tức là:
  - Với xác suất ε: chọn hành động ngẫu nhiên (khám phá – explore)
  - Với xác suất 1−ε: chọn hành động tốt nhất theo Q hiện tại (khai thác – exploit))
- Thực hiện hành động a, nhận phần thưởng r và trạng thái mới s'
- Cập nhật Q-value theo công thức:
  - $Q(s, a) \leftarrow Q(s, a) + \alpha \times [r + \gamma \times \max_{a'} Q(s', a') - Q(s, a)]$
    - α: tốc độ học (learning rate)
    - γ: hệ số chiết khấu (discount factor)
- Cập nhật trạng thái hiện tại thành s', lặp lại đến khi kết thúc.

Tóm tắt thuật toán:

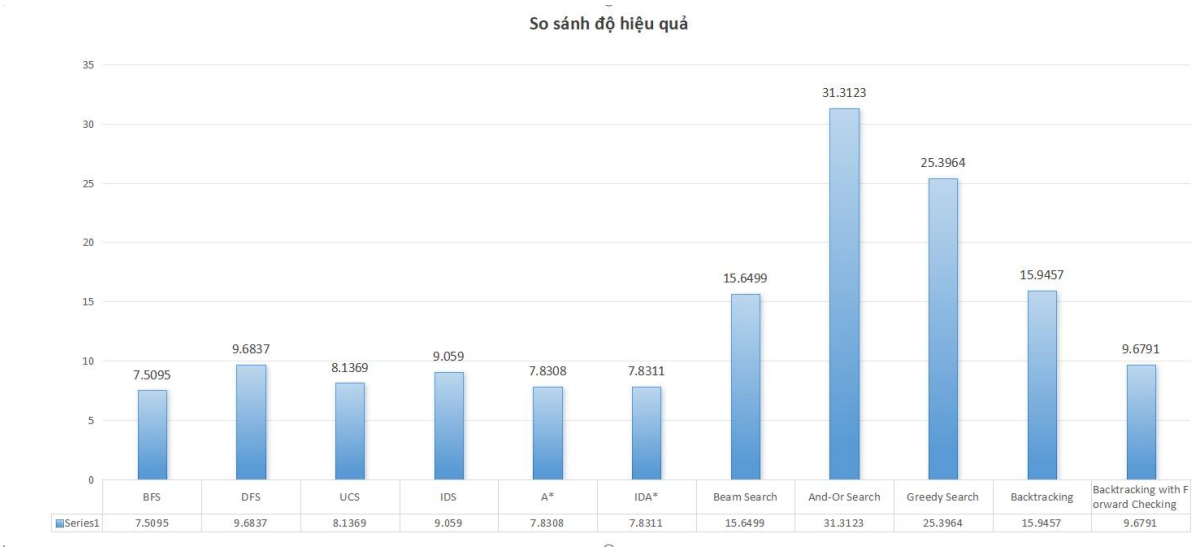
Thuật toán	**Loại học**	**Ưu điểm**	**Nhược điểm**	**Độ phức tạp thời gian**	**Độ phức tạp không gian**
Q-Learning	Học củng cố	Học chiến lược tối ưu thông	Cần thời gian huấn luyện	O(E × A) (E: số	O(S × A) (S: số

Thuật toán	Loại học	Ưu điểm	Nhược điểm	Độ phức tạp thời gian	Độ phức tạp không gian
	không mô hình	qua trải nghiệm, xử lý tốt môi trường lớn	dài, bảng Q rất lớn nếu trạng thái nhiều	episode, A: số hành động)	trạng thái, A: số hành động)

Nhận xét:

Q-learning là một thuật toán học mạnh mẽ, đặc biệt phù hợp với các môi trường mà trạng thái không được biết trước hoàn toàn hoặc có yếu tố ngẫu nhiên. Tuy nhiên, đối với trò chơi như 8 ô, không gian trạng thái rất lớn (≈ 9! = 362,880 trạng thái có thể), nên việc lưu trữ bảng Q trở nên tốn kém bộ nhớ. Các cải tiến như Deep Q-Learning dùng mạng nơron thay cho bảng Q truyền thống có thể giúp mở rộng khả năng áp dụng cho không gian liên tục hoặc cực lớn. Dù vậy, về mặt học thuật, Q-learning vẫn là một ví dụ điển hình minh họa khả năng học hành vi tối ưu qua trải nghiệm thay vì lập trình trước.

IV. Phân tích kết quả



Kết quả cho thấy:

Các thuật toán có hiệu suất cao nhất là A* (7.8308s), IDA* (7.8311s) và BFS (7.5095s). Đây là các thuật toán đảm bảo tìm lời giải ngắn nhất, trong đó A* và IDA*

sử dụng heuristic để định hướng tìm kiếm, còn BFS duyệt toàn bộ theo từng mức nên không bỏ sót lời giải tối ưu.

- Greedy Best-First Search (25.3964s) và AND-OR Search (31.3123s) là hai thuật toán có thời gian giải lâu nhất. Nguyên nhân là:
  - Greedy chỉ dùng heuristic (h(n)) mà bỏ qua chi phí thực, dễ lạc hướng.
  - AND-OR phải xử lý nhiều nhánh lựa chọn, gây bùng nổ trạng thái trong môi trường không xác định.
- Các thuật toán tìm kiếm truyền thống như DFS, UCS, IDS có thời gian trung bình, dao động từ 8–10s. Trong đó:
  - DFS có thể rơi vào nhánh sai và cần thời gian quay lui.
  - UCS mở rộng theo chi phí nên duyệt rộng hơn DFS, nhưng chậm hơn BFS nếu tất cả chi phí là đồng đều.
- Backtracking (15.9457s) mất nhiều thời gian do có thể lặp lại trạng thái và không có hướng dẫn heuristic. Tuy nhiên, khi kết hợp với Forward Checking (9.6791s), thời gian được cải thiện đáng kể nhờ loại trừ nhánh sai từ sớm.
- Beam Search (15.6499s) hoạt động hiệu quả hơn một số thuật toán không định hướng do chỉ giữ lại k trạng thái tốt nhất, nhưng vẫn có nguy cơ bỏ lỡ lời giải tối ưu.

Nhìn chung, các thuật toán định hướng tốt bằng heuristic và có cơ chế kiểm soát mở rộng như A* và IDA* là lựa chọn tối ưu nhất cho trò chơi 8 ô. Trong khi đó, các thuật toán thiếu chiến lược hoặc xử lý nhiều nhánh (như Greedy hoặc AND-OR) có thể mất thời gian đáng kể dù logic đúng.

V. Kết luận

Việc triển khai và thử nghiệm nhiều nhóm thuật toán tìm kiếm trên trò chơi 8 ô đã cung cấp cái nhìn trực quan và sâu sắc về hiệu quả của từng chiến lược trong các điều kiện thông tin khác nhau. Mỗi thuật toán mang những ưu và nhược điểm riêng, và phát huy hiệu quả trong từng hoàn cảnh cụ thể – từ các thuật toán không dùng heuristic,

đến những phương pháp cục bộ, có ràng buộc hoặc hoạt động trong môi trường không xác định.

Thông qua quá trình thực nghiệm, bài toán không chỉ giúp củng cố kiến thức về tìm kiếm trong trí tuệ nhân tạo, mà còn rèn luyện khả năng phân tích, so sánh và lựa chọn giải pháp phù hợp. Đây cũng là cơ sở quan trọng để ứng dụng vào các lĩnh vực thực tế, lập kế hoạch hành động và các hệ thống thông minh trong tương lai.