

Xây dựng ứng dụng giải bài toán 8 Puzzle

Thông tin đề tài

- **Trường:** HCMC University of Technology and Education
- **Môn học:** Trí Tuệ Nhân Tạo
- **Giảng viên hướng dẫn:** TS. Phan Thị Huyền Trang
- **Mã số sinh viên:** 23110222
- **Tên sinh viên:** Nghiêm Quang Huy

Mục lục

1. [Giới thiệu bài toán 8 Puzzle](#)
2. [Mục tiêu](#)
3. [Tổng quan các thuật toán áp dụng](#)
 - [Uninformed Search](#)
 - [Informed Search](#)
 - [Local Search](#)
 - [Complex Spaces Search](#)
 - [Reinforcement Learning](#)
 - [constraint satisfaction](#)
4. [So sánh hiệu suất](#)
5. [Hướng dẫn sử dụng](#)

Giới thiệu bài toán 8 Puzzle

Bài toán **8 Puzzle** là một trò chơi xếp hình kinh điển, bao gồm một bảng 3x3 với 8 ô được đánh số từ 1 đến 8 và một ô trống (ký hiệu là 0). Người chơi di chuyển ô trống (lên, xuống, trái, phải) để sắp xếp các ô số về trạng thái mục tiêu: [1, 2, 3, 4, 5, 6, 7, 8, 0]

Dự án này phát triển một ứng dụng sử dụng các thuật toán trí tuệ nhân tạo để tự động giải bài toán 8 Puzzle, đồng thời so sánh hiệu quả của các thuật toán dựa trên số bước, thời gian, và bộ nhớ sử dụng.

Mục tiêu

Phát triển ứng dụng giải bài toán **8 Puzzle**, sử dụng và so sánh hiệu quả các nhóm thuật toán tìm kiếm phổ biến trong trí tuệ nhân tạo, gồm:

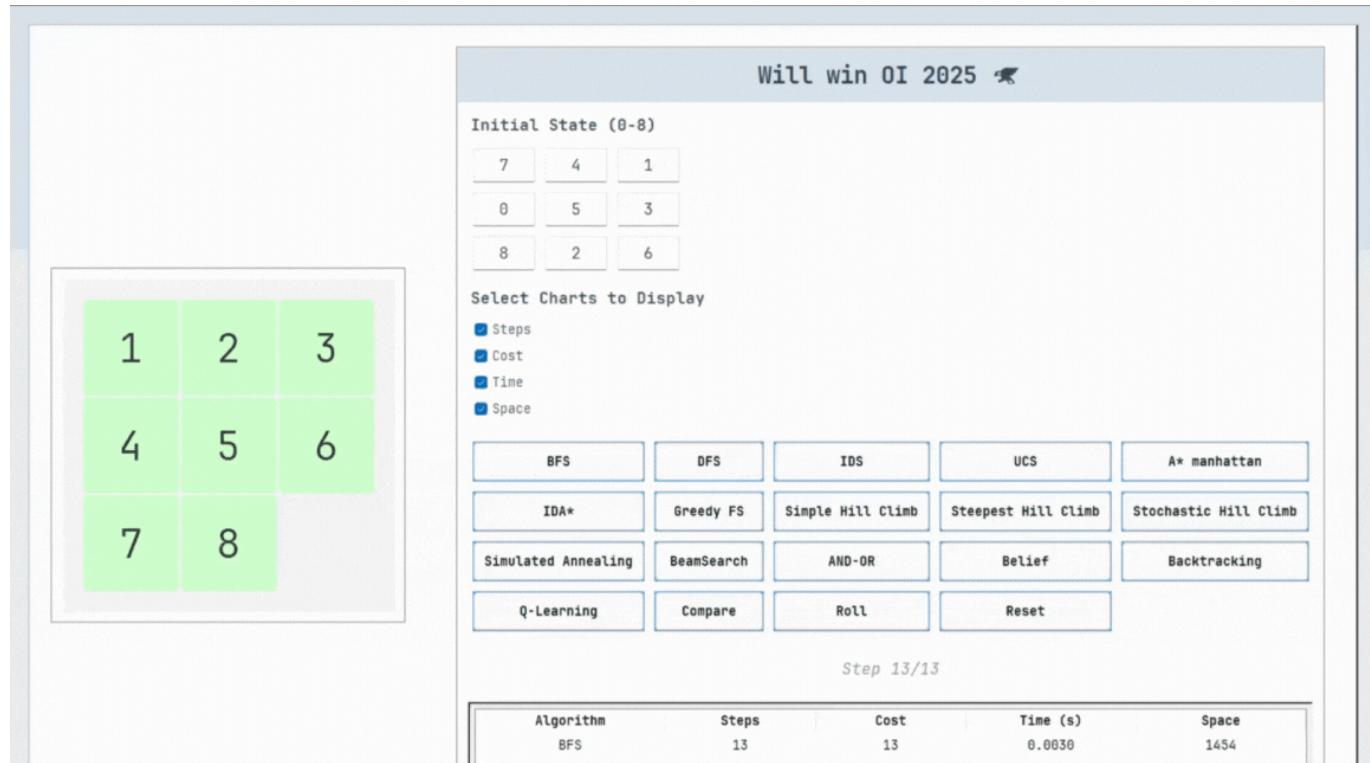
- **Uninformed Search** (Tìm kiếm không thông tin)
- **Informed Search** (Tìm kiếm có thông tin)
- **Local Search** (Tìm kiếm cục bộ)
- **Complex Spaces Search** (Tìm kiếm trong không gian phức tạp)
- **Reinforcement Learning** (Học tăng cường)
- **Constraint Satisfaction** (Tìm kiếm trong môi trường có ràng buộc)

Tổng quan các thuật toán áp dụng

1. Uninformed Search

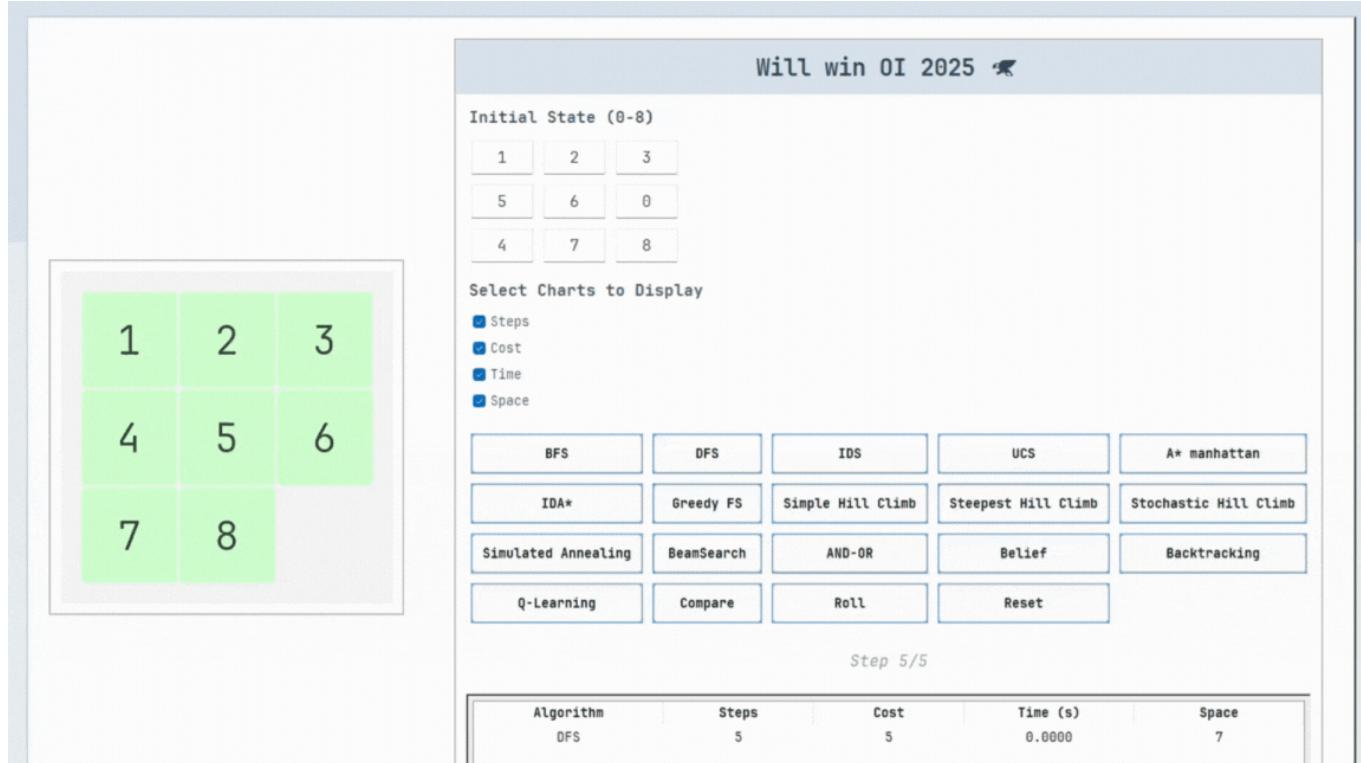
Các thuật toán tìm kiếm không dùng thông tin heuristic, hoạt động dựa trên cấu trúc không gian trạng thái của 8 Puzzle:

➤ Breadth-First Search (BFS)



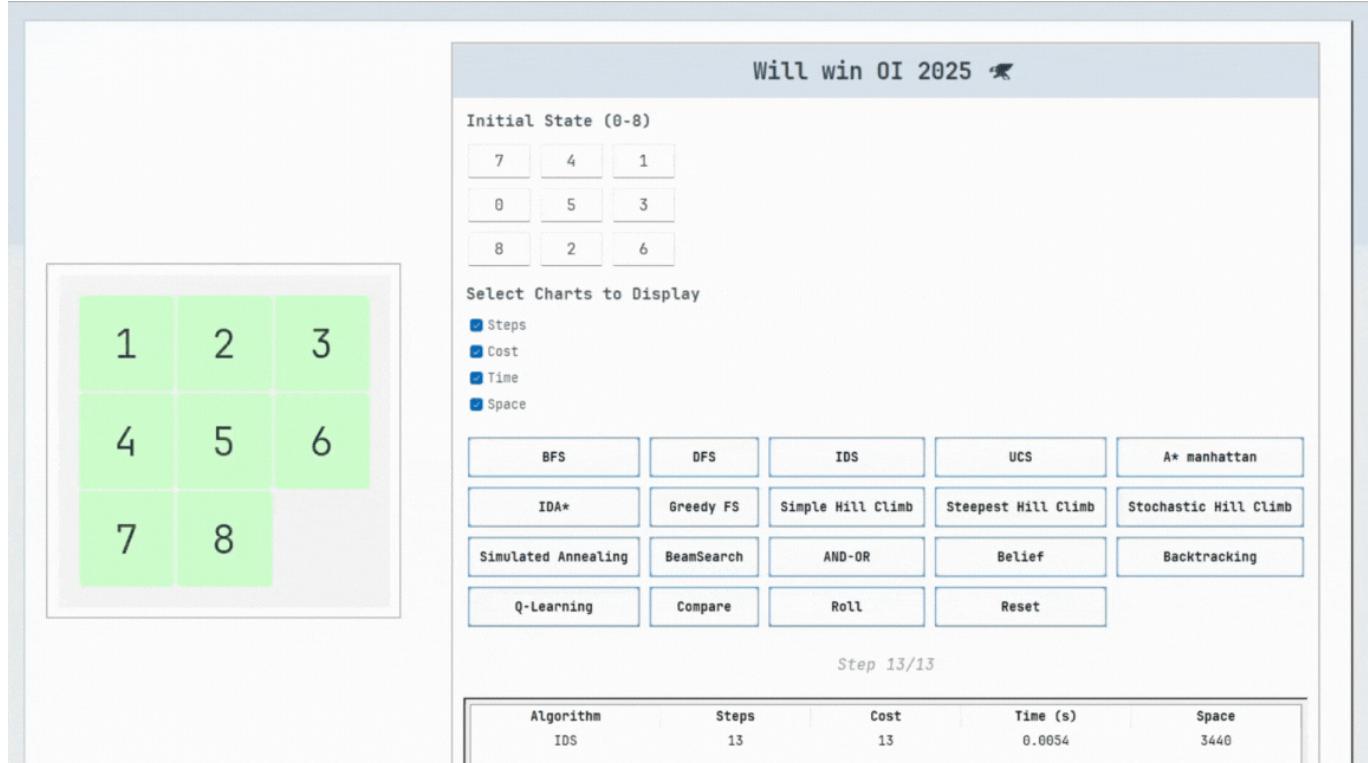
- **Chiến lược:** Mở rộng tất cả các trạng thái ở một mức độ (lớp) trước khi đi sâu xuống mức tiếp theo.
- **Cấu trúc dữ liệu:** Queue.
- **Ưu điểm:** Luôn tìm được lời giải ngắn nhất nếu chi phí giữa các bước là như nhau.
- **Nhược điểm:** Do phải lưu trữ toàn bộ các nút ở một mức độ trước khi chuyển sang mức tiếp theo.
- **Độ phức tạp:**
 - Thời gian: $O(b^d)$
 - Bộ nhớ: $O(b^d)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - d (depth): độ sâu của trạng thái mục tiêu trong cây.

► Depth-First Search (DFS)



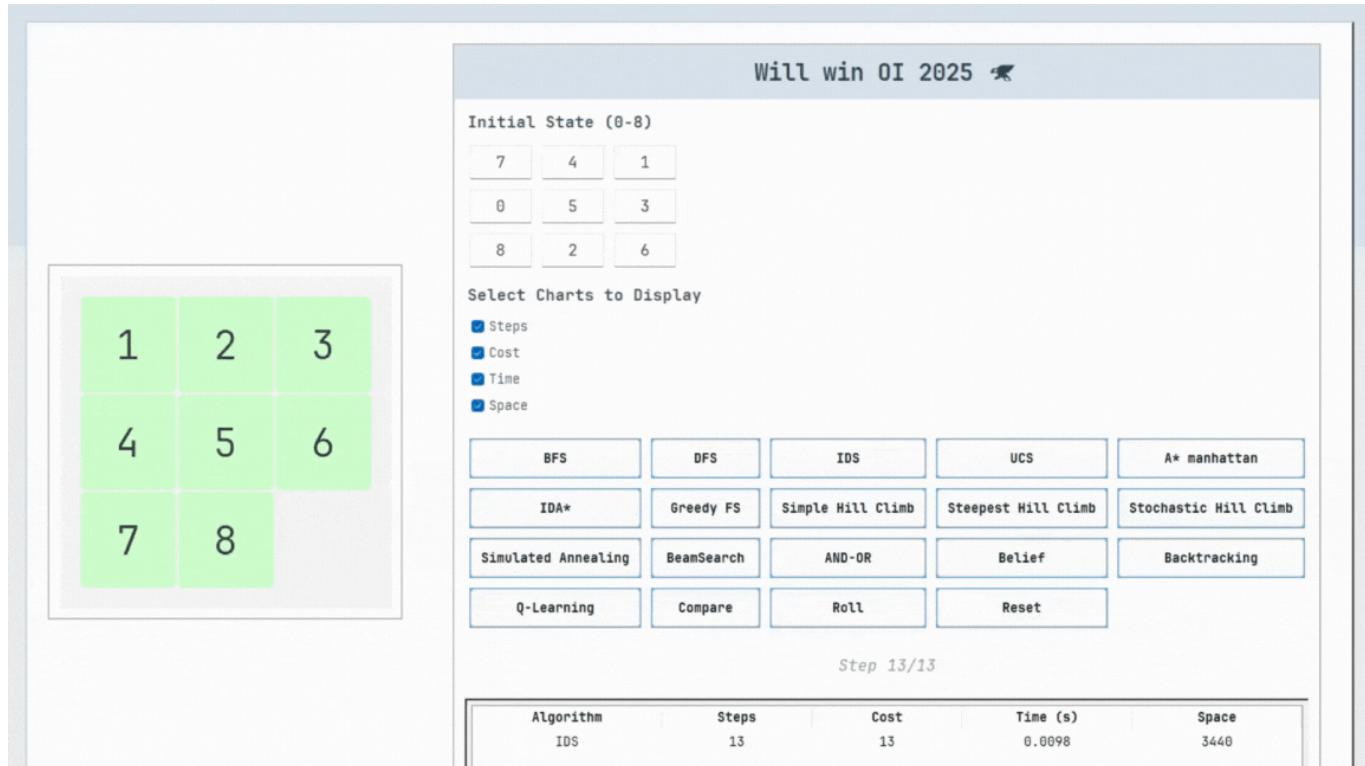
- **Chiến lược:** Duyệt sâu trước, mở rộng hết một nhánh rồi quay lại.
- **Cấu trúc dữ liệu:** Stack.
- **Ưu điểm:** Sử dụng bộ nhớ hiệu quả hơn so với BFS, do không cần lưu trữ tất cả các trạng thái ở cùng một mức độ.
- **Nhược điểm:** Không đảm bảo tìm được lời giải ngắn nhất nếu tồn tại nhiều đường đi đến mục tiêu.
- **Độ phức tạp:**
 - Thời gian: $O(b^m)$
 - Bộ nhớ: $O(bm)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - m (maximum depth): độ sâu lớn nhất mà DFS có thể đi tới trong cây.

► Uniform Cost Search (UCS)



- **Chiến lược:** Mở rộng trạng thái mà có **tổng chi phí thấp nhất** từ gốc đến hiện tại.
- **Cấu trúc dữ liệu:** Priority Queue.
- **Ưu điểm:** Luôn tìm được lời giải tối ưu (chi phí thấp nhất), với điều kiện mọi bước đi đều có chi phí dương.
- **Nhược điểm:** Hiệu năng và bộ nhớ sử dụng tương đương BFS trong các không gian tìm kiếm lớn.
- **Độ phức tạp:**
 - Thời gian: $O(b^{1 + C^*/\varepsilon})$
 - Bộ nhớ: $O(b^{1 + C^*/\varepsilon})$
 - Trong đó:
 - C^* : chi phí tối ưu để tìm lời giải.
 - ε : bước chi phí nhỏ nhất.
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).

▶ Iterative Deepening Search (IDS)



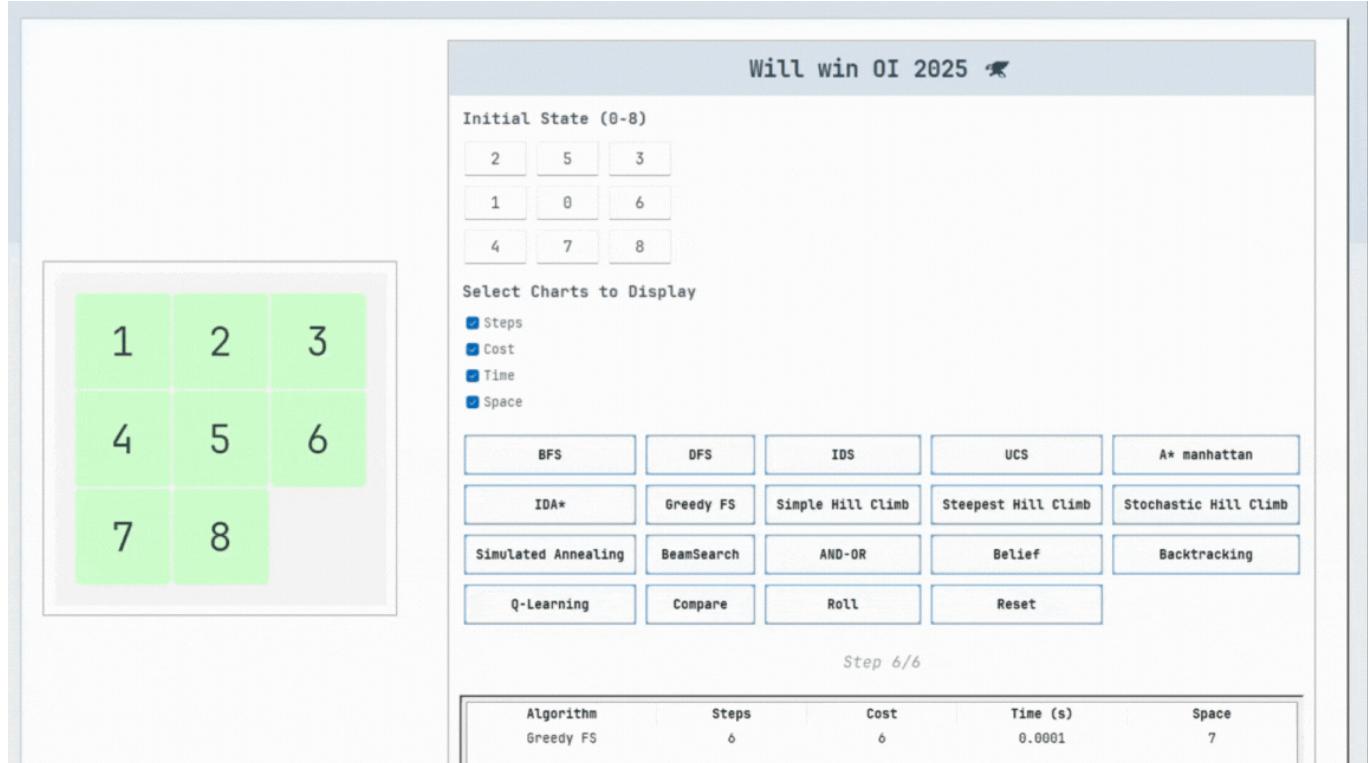
- **Chiến lược:** Kết hợp ưu điểm của DFS và BFS bằng cách thực hiện DFS lặp lại nhiều lần với giới hạn độ sâu tăng dần (depth limit). Mỗi lần lặp, thuật toán tìm kiếm trên cây đến một độ sâu nhất định rồi quay lại từ gốc với giới hạn mới.
- **Ưu điểm:** Tìm được lời giải ngắn nhất như BFS nhưng tốn ít bộ nhớ như DFS.
- **Nhược điểm:** Thời gian bị lãng phí do phải lặp lại việc duyệt các trạng ở các độ sâu nhỏ nhiều lần.
- **Độ phức tạp:**
 - Thời gian: $O(b^d)$
 - Bộ nhớ: $O(bd)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - d (depth): độ sâu của trạng thái mục tiêu trong cây.

2. Informed Search

Các thuật toán sử dụng heuristic để ước lượng chi phí từ trạng thái hiện tại đến đích, ưu tiên mở rộng các trạng thái có khả năng dẫn đến lời giải nhanh hơn. Trong bài toán **8 Puzzle**, các thuật toán nhóm Informed Search bao gồm:

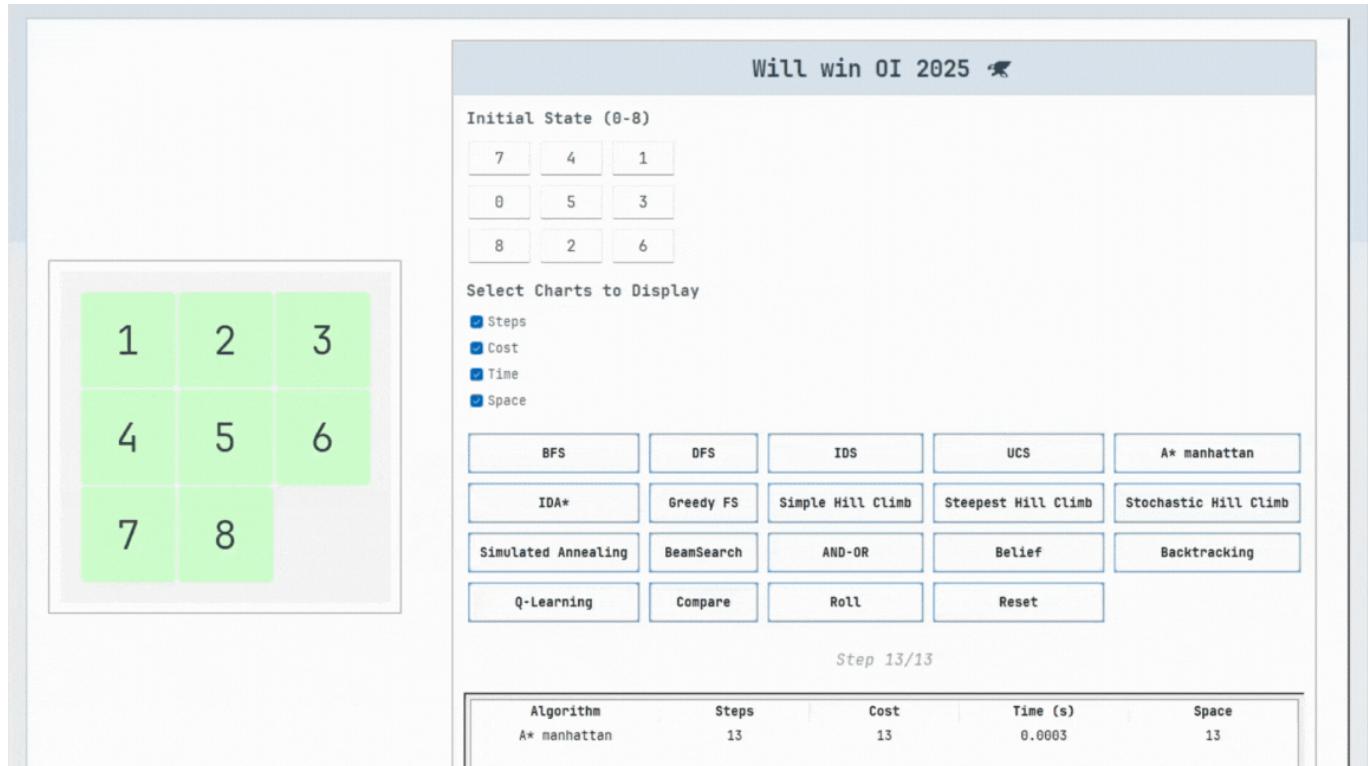
- Greedy Best-First Search
- A* Search
- Iterative Deepening A* (IDA*)

► Greedy Best-First Search (GBFS)



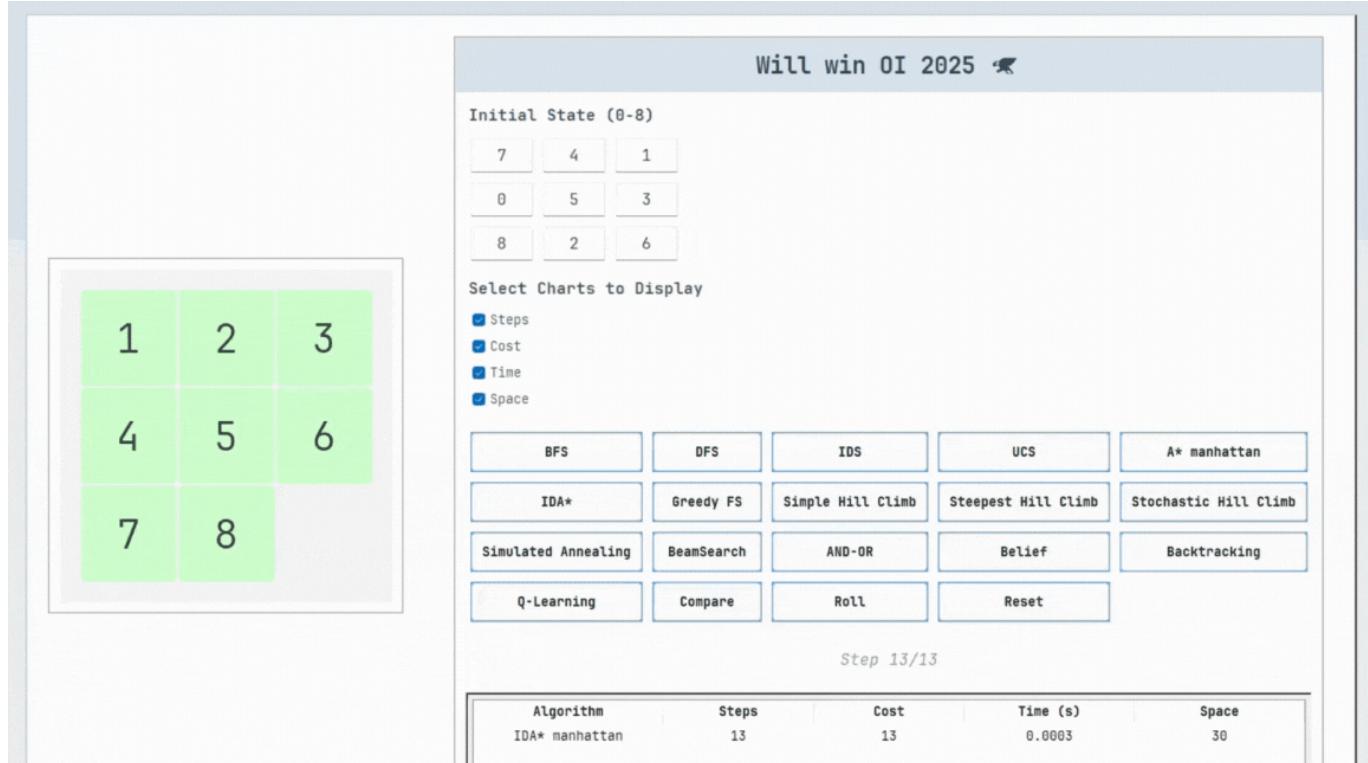
- **Chiến lược:** Mở rộng trạng thái lân cận có giá trị heuristic $h(n)$ nhỏ nhất (sử dụng Manhattan Distance).
- **Cấu trúc dữ liệu:** Priority Queue (min heap).
- **Ưu điểm:**
 - Tìm kiếm nhanh, tận dụng thông tin heuristic để đi thẳng đến đích.
 - Tiết kiệm bộ nhớ hơn so với BFS hay UCS nếu heuristic tốt.
- **Nhược điểm:**
 - Không đảm bảo tìm ra lời giải tối ưu (ngắn nhất).
 - Có thể bị mắc kẹt nếu heuristic không chính xác.
- **Độ phức tạp:**
 - Thời gian và bộ nhớ trong trường hợp xấu nhất: $O(b^m)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - m (maximum depth): độ sâu lớn nhất mà thuật toán có thể đi tới.

► A* Search



- **Chiến lược:** Kết hợp chi phí thực tế từ gốc đến hiện tại $g(n)$ và ước lượng chi phí đến đích heuristic $h(n)$ để đánh giá node theo $f(n) = g(n) + h(n)$ (sử dụng Manhattan Distance hoặc Linear Conflict).
- **Ưu điểm:**
 - Tìm được lời giải tối ưu nếu heuristic là **đúng và không vượt quá thực tế**.
 - Hiệu quả hơn so với tìm kiếm không thông tin.
- **Nhược điểm:** Tốn bộ nhớ lớn khi không gian tìm kiếm rộng.
- **Độ phức tạp:**
 - Thời gian: $O(b^d)$ (phụ thuộc vào heuristic).
 - Bộ nhớ: $O(b^d)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - d (depth): độ sâu của trạng thái mục tiêu trong cây.

▶ Iterative Deepening A* (IDA*)



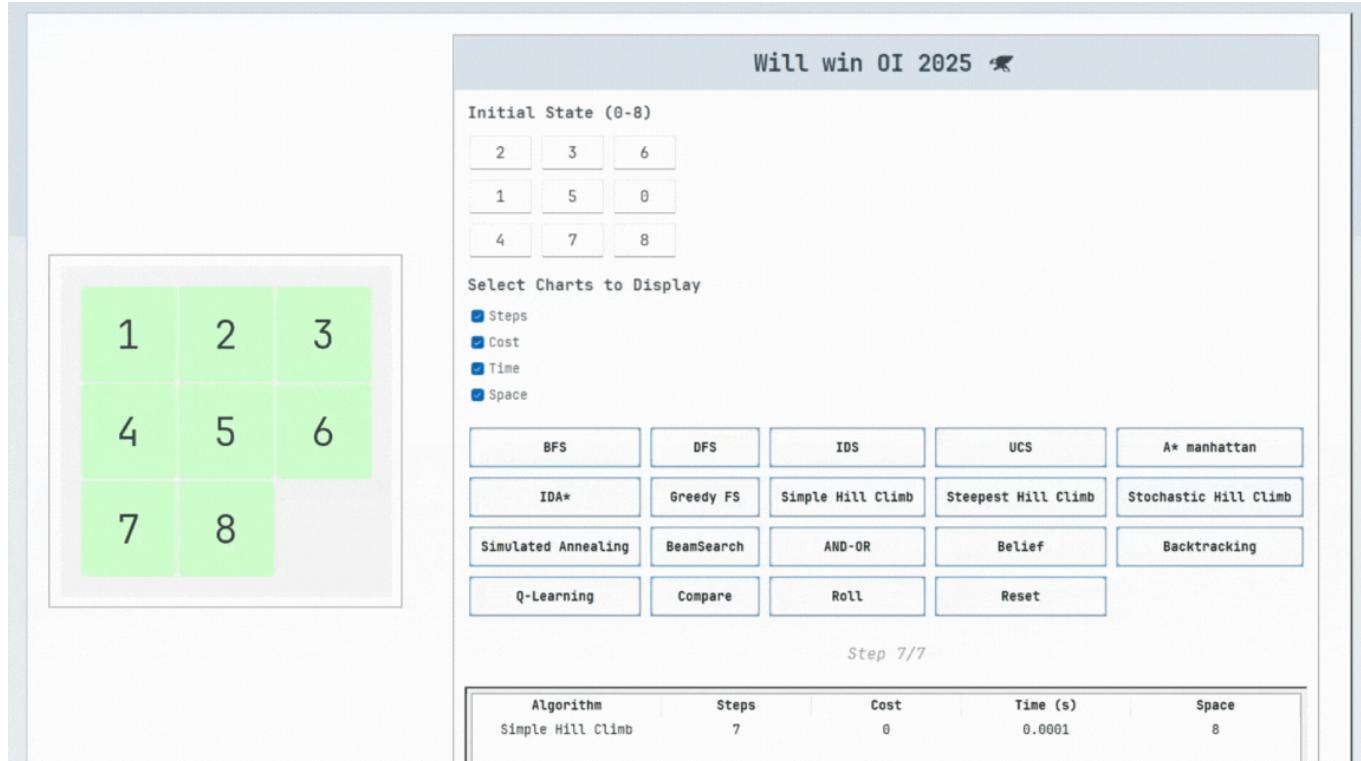
- Chiến lược:** Kết hợp IDS và A* bằng cách lặp lại tìm kiếm sâu với ngưỡng $f(n)$ tăng dần, chỉ mở rộng các nút có $f(n) \leq$ **ngưỡng hiện tại** (sử dụng Manhattan Distance).
- Ưu điểm:**
 - Giảm bộ nhớ sử dụng so với A*.
 - Vẫn giữ được tính tối ưu của A*.
- Nhược điểm:** Có thể lặp lại mở rộng node nhiều lần, gây tốn thời gian.
- Độ phức tạp:**
 - Thời gian: $O(b^d)$
 - Bộ nhớ: $O(d)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - d (depth): độ sâu của trạng thái mục tiêu trong cây.

3. Local Search

Tìm kiếm theo hướng cải thiện trạng thái hiện tại mà không cần lưu toàn bộ đường đi, phù hợp với không gian trạng thái lớn của 8 Puzzle. Các thuật toán nhóm Local Search bao gồm:

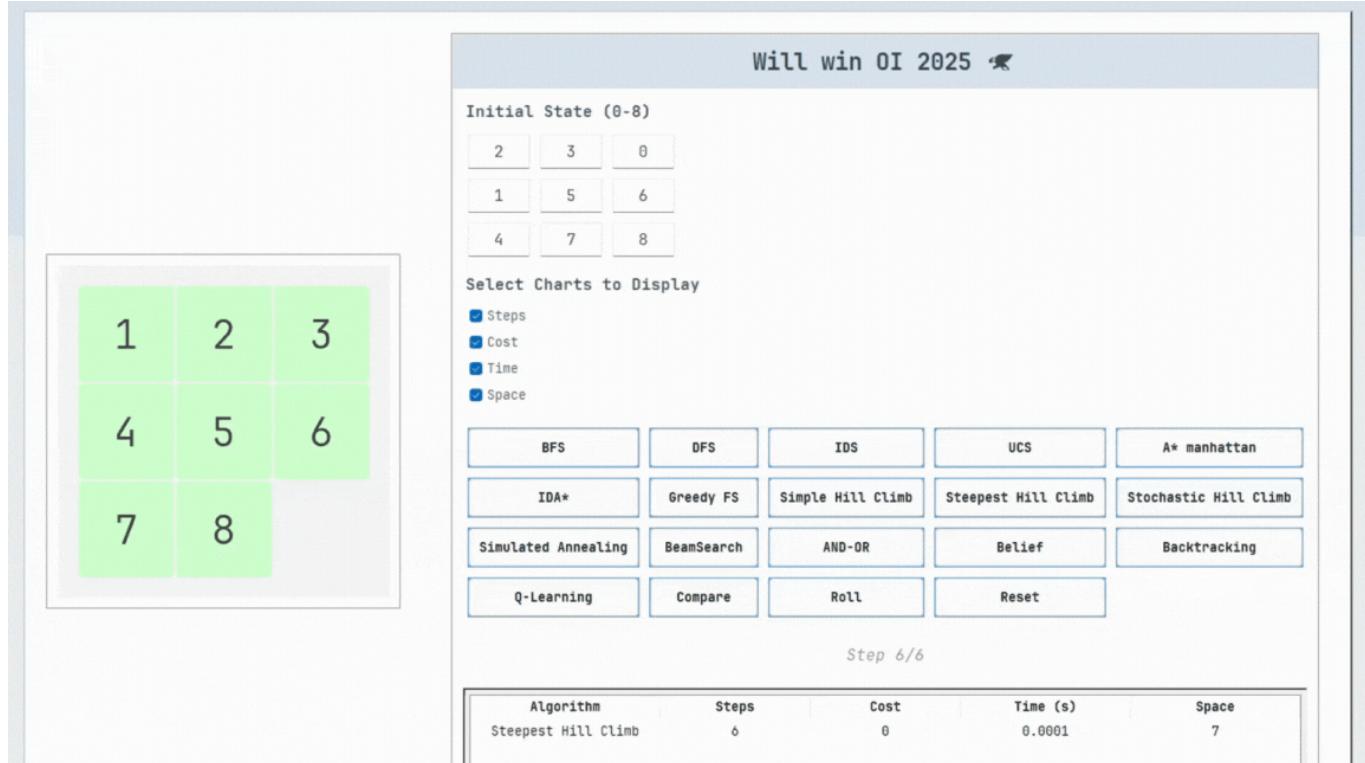
- Hill Climbing (Simple, Steepest, Stochastic)
- Simulated Annealing
- Beam Search

► Simple Hill Climbing



- **Chiến lược:** Chọn trạng thái lân cận đầu tiên có giá trị heuristic tốt hơn (sử dụng Manhattan Distance).
- **Ưu điểm:** Cài đặt đơn giản, tốc độ nhanh.
- **Nhược điểm:** Dễ kẹt ở local optimum hoặc vùng phẳng (plateau).
- **Độ phức tạp:**
 - Thời gian: $O(b^m)$
 - Bộ nhớ: $O(1)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - m (maximum depth): độ sâu lớn nhất mà thuật toán có thể đi tới.

► Steepest Hill Climbing



- **Chiến lược:** So sánh toàn bộ trạng thái lân cận và chọn trạng thái có giá trị heuristic tốt nhất (sử dụng Manhattan Distance).
- **Ưu điểm:** Tăng khả năng tránh lựa chọn sai như Simple Hill Climbing.
- **Nhược điểm:** Tốn thời gian để đánh giá toàn bộ lân cận; vẫn dễ bị kẹt ở local optimum.
- **Độ phức tạp:**
 - Thời gian: $O(b^m)$
 - Bộ nhớ: $O(1)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - m (maximum depth): độ sâu lớn nhất mà thuật toán có thể đi tới.

► Stochastic Hill Climbing

The screenshot shows the Will win OI 2025 software interface. On the left, a 3x3 grid displays the initial state of an 8-puzzle:

1	2	3
4	5	6
7	8	

On the right, the software interface includes:

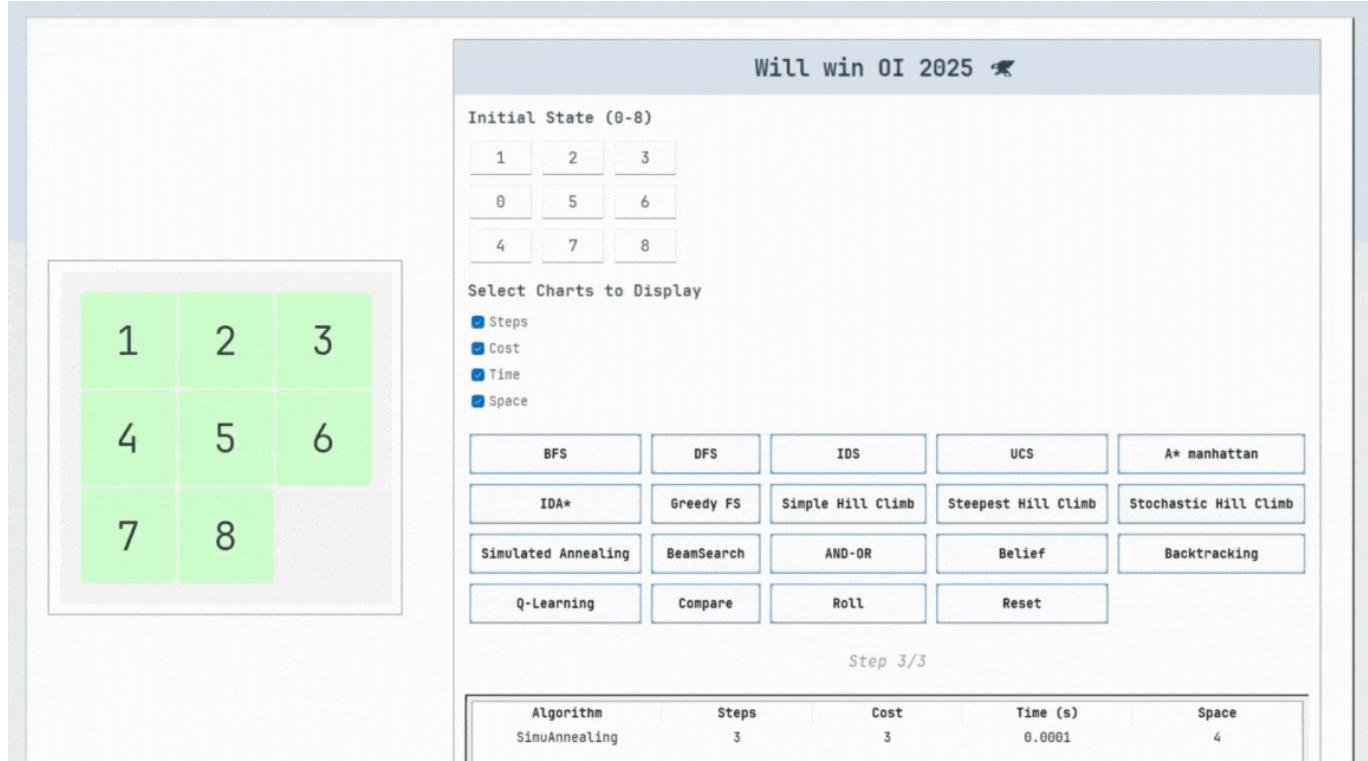
- Initial State (8-8)**: Shows the 3x3 grid above.
- Select Charts to Display**: A checkbox group with four options: Steps, Cost, Time, and Space.
- Algorithm Selection**: A grid of buttons for different search algorithms:

BFS	DFS	IDS	UCS	A* manhattan
IDA*	Greedy FS	Simple Hill Climb	Steepest Hill Climb	Stochastic Hill Climb
Simulated Annealing	BeamSearch	AND-OR	Belief	Backtracking
Q-Learning	Compare	Roll	Reset	
- Step 6/6**: A status message indicating the current step.
- Performance Data**: A table showing the results for the Steepest Hill Climb algorithm:

Algorithm	Steps	Cost	Time (s)	Space
Steepest Hill Climb	6	0	0.0001	7

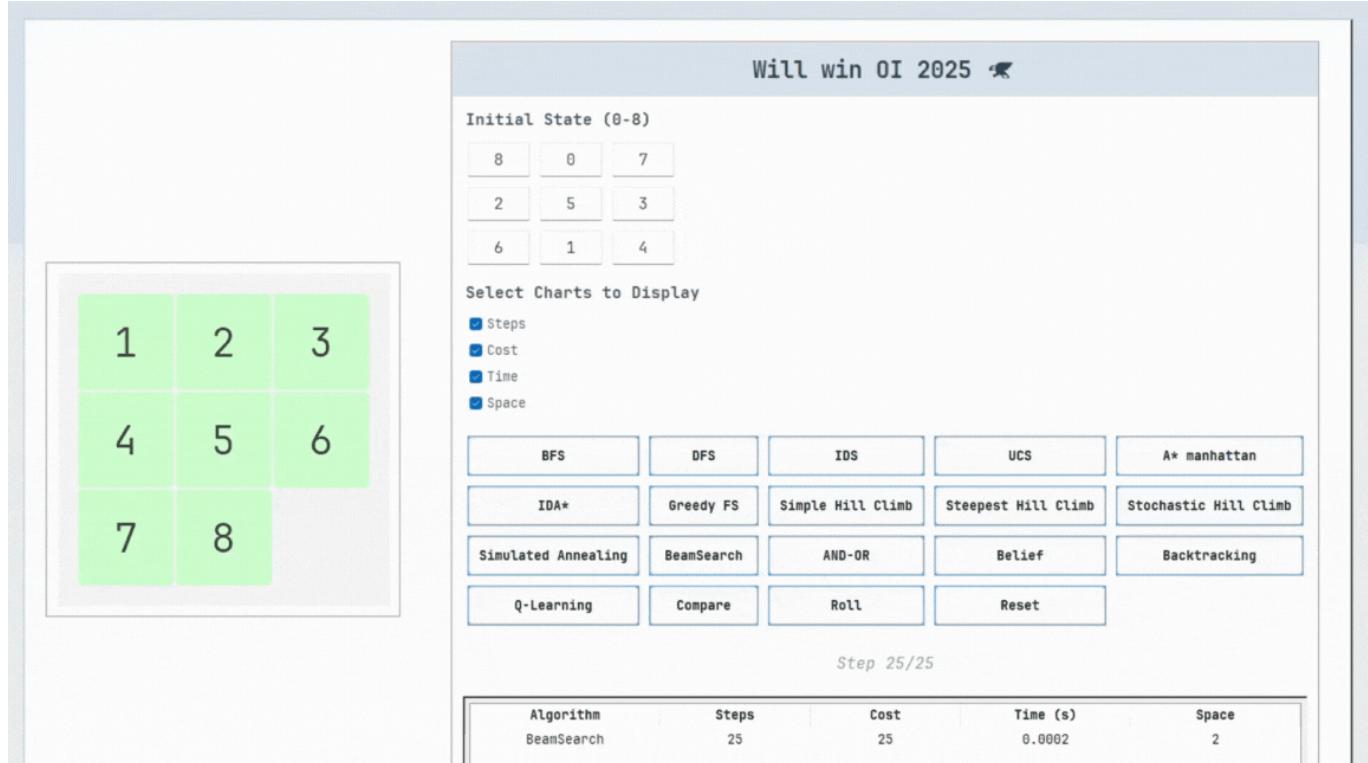
- **Chiến lược**: Chọn ngẫu nhiên một trong các trạng thái lân cận có cải thiện heuristic (sử dụng Manhattan Distance).
- **Ưu điểm**: Tăng khả năng thoát local optimum, tránh rơi vào vùng phẳng.
- **Nhược điểm**: Kết quả không ổn định, phụ thuộc vào ngẫu nhiên.
- **Độ phức tạp**:
 - Thời gian: $O(bm)$
 - Bộ nhớ: $O(1)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - m (maximum depth): độ sâu lớn nhất mà thuật toán có thể đi tới.

► Simulated Annealing



- **Chiến lược:** Cho phép chọn trạng thái xấu hơn với xác suất giảm dần theo thời gian, tránh mắc kẹt ở local optimum (sử dụng Manhattan Distance).
- **Ưu điểm:** Có thể thoát local optimum và tìm lời giải tốt hơn.
- **Nhược điểm:** Hiệu quả phụ thuộc vào cách giảm nhiệt độ (cooling schedule).
- **Độ phức tạp:**
 - Thời gian: $O(b^m)$
 - Bộ nhớ: $O(1)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - m (maximum depth): độ sâu lớn nhất mà thuật toán có thể đi tới.

▶ Beam Search



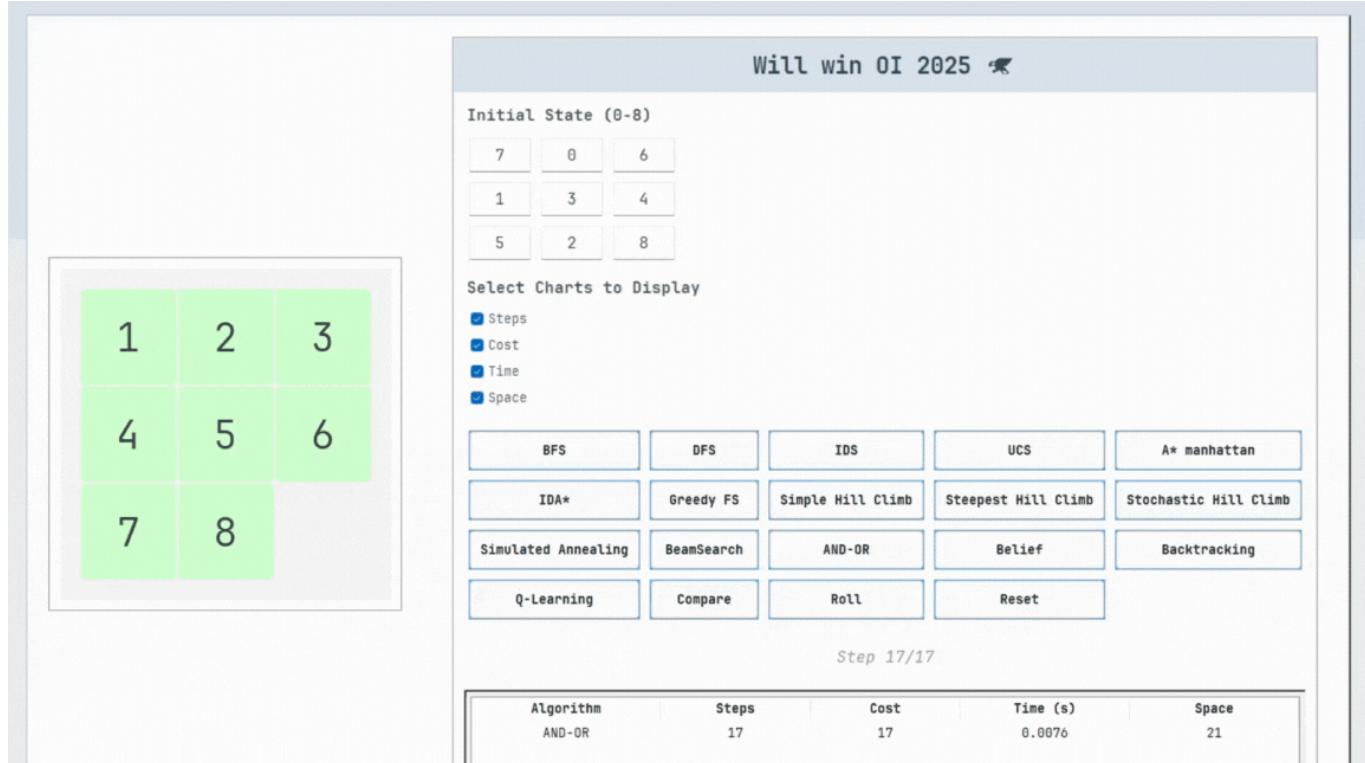
- **Chiến lược:** Giữ lại k trạng thái tốt nhất tại mỗi bước (k gọi là beam width) thay vì chỉ một (sử dụng Manhattan Distance).
- **Ưu điểm:** Dễ mở rộng, tránh kẹt local optimum tốt hơn Hill Climbing.
- **Nhược điểm:** Không đảm bảo tối ưu, dễ bỏ sót lời giải nếu k quá nhỏ.
- **Độ phức tạp:**
 - Thời gian: $O(kbm)$
 - Bộ nhớ: $O(k)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - m (maximum depth): độ sâu lớn nhất mà thuật toán có thể đi tới.
 - k (beam width): số trạng thái tốt nhất được giữ lại.

4. Complex Spaces Search

Các kỹ thuật dành cho không gian tìm kiếm lớn hoặc có cấu trúc phức tạp trong 8 Puzzle. Các thuật toán nhóm Complex Spaces Search bao gồm:

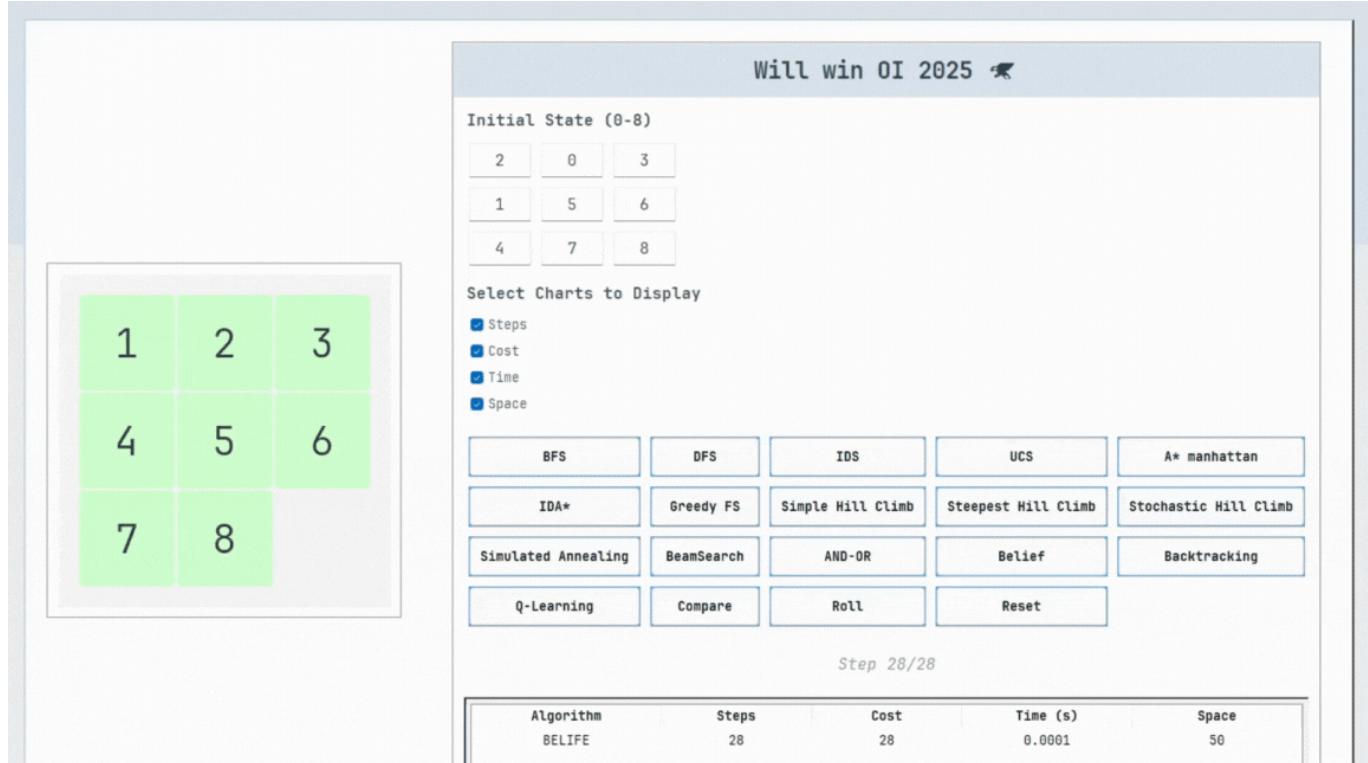
- AND-OR Graph Search
- Belief Search

► AND-OR Graph Search



- Chiến lược:** Kết hợp node "AND" (cần tất cả con) và "OR" (chỉ cần một con), sử dụng Manhattan Distance để ưu tiên.
- Ưu điểm:** Tốt cho các hệ thống có nhiều kết quả khả thi phụ thuộc vào hành động.
- Nhược điểm:** Cấu trúc phức tạp, khó triển khai với không gian lớn như 8 Puzzle.
- Độ phức tạp:**
 - Thời gian: $O(b^m)$
 - Bộ nhớ: $O(b^d)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - d (depth): độ sâu của trạng thái mục tiêu trong cây.

► Belief Search



- **Chiến lược:** Tìm kiếm với ràng buộc belief state (ví dụ: hàng đầu tiên là [1, 2, 3]), sử dụng DFS.
- **Ưu điểm:** Áp dụng được trong môi trường không chắc chắn.
- **Nhược điểm:** Không phù hợp với bài toán xác định như 8 Puzzle, giới hạn không gian tìm kiếm.
- **Độ phức tạp:**
 - Thời gian: $O(b^m)$
 - Bộ nhớ: $O(b^m * N)$
 - Trong đó:
 - b (branching factor): số lượng trạng thái con trung bình (~4 trong 8 Puzzle).
 - m (maximum depth): độ sâu lớn nhất mà thuật toán có thể đi tới.
 - N : số lượng trạng thái trong không gian ban đầu (với 8 Puzzle là $9! = 362,880$).

5. Reinforcement Learning

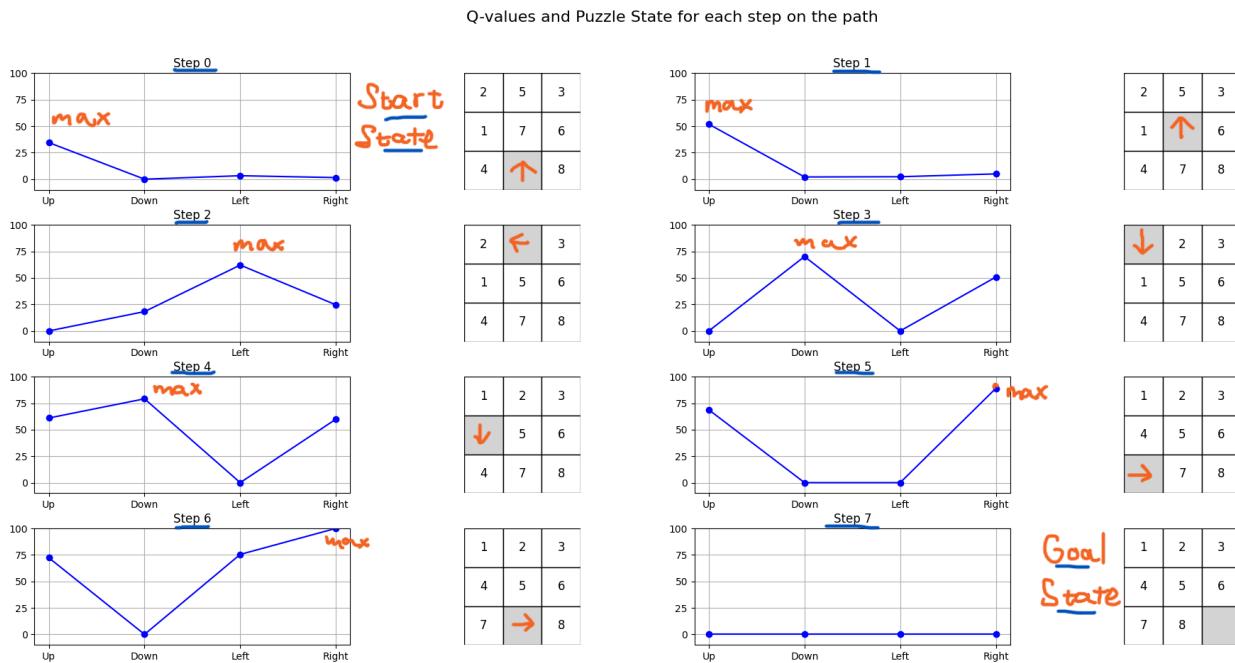
Tìm chính sách hành động tối ưu thông qua tương tác với môi trường 8 Puzzle:

► Q-Learning

- **Chiến lược:**

Học chính sách hành động bằng cách cập nhật bảng Q (Q-table) dựa trên kinh nghiệm thu thập được từ tương tác với môi trường.

Sử dụng chính sách **epsilon-greedy** để cân bằng giữa **khám phá** (exploration) và **khai thác** (exploitation).



- **Công thức cập nhật Q-value:**

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

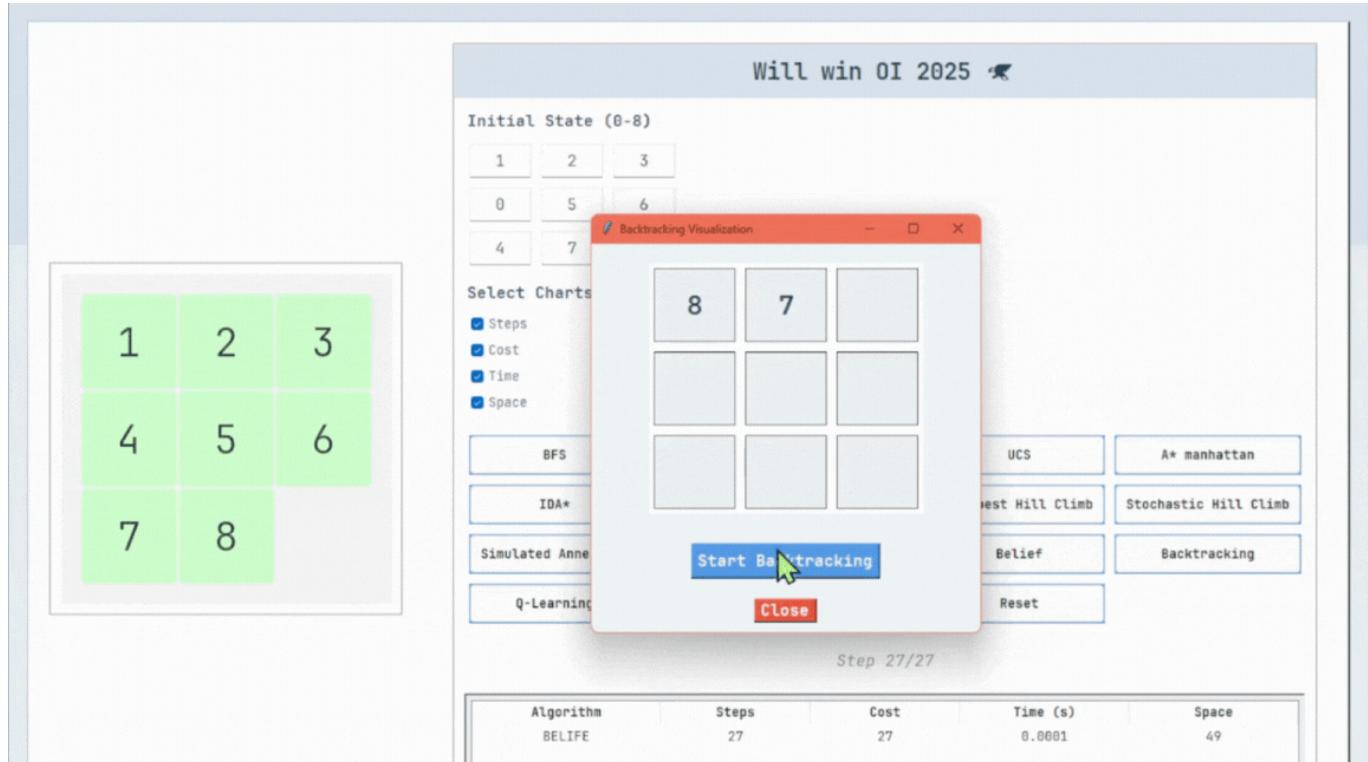
Trong đó:

- $Q(s, a)$: Giá trị Q hiện tại tại trạng thái s khi thực hiện hành động a
- α (alpha): Tốc độ học (learning rate), $0 < \alpha \leq 1$
- r : Phần thưởng nhận được sau khi thực hiện hành động
- γ (gamma): Hệ số chiết khấu (discount factor), thể hiện mức độ ưu tiên phần thưởng tương lai, $0 \leq \gamma \leq 1$
- s' : Trạng thái mới sau khi thực hiện hành động a
- $\max(Q(s', a'))$: Giá trị Q lớn nhất có thể đạt được từ trạng thái mới s'

- hành động a là thao tác di chuyển ô trống [lên, xuống, trái, phải]
- **Ưu điểm:**
 - Có thể học từ môi trường mà không cần heuristic hay mô hình trạng thái.
 - càng về sau khả năng khai thác càng cao.
 - Áp dụng được trong môi trường không xác định (model-free).
- **Nhược điểm:**
 - Tốn thời gian huấn luyện, đặc biệt khi không gian trạng thái lớn.
 - Phụ thuộc mạnh vào tham số (α , γ , ϵ).
 - tốn thời gian train trong bài toán 8 Puzzle do không gian trạng thái quá lớn ($9! / 2 \sim 181,440$ trạng thái hợp lệ).
- **Độ phức tạp:**
 - **Thời gian:** Phụ thuộc vào số lượng episode và số bước tối đa mỗi episode.
 - **Bộ nhớ:** Tăng theo kích thước Q-table, có thể lên đến hàng trăm nghìn cặp (s, a) trong 8 Puzzle.

6. Constraint Satisfaction

Tìm trạng thái hợp lệ thỏa mãn tất cả các ràng buộc bằng cách gán giá trị cho các biến, sử dụng thuật toán **Backtracking**.



► Chiến lược:

- Đại diện bài toán như một tập hợp các **biến** (9 ô trong ma trận 3x3 của 8 Puzzle).
- Mỗi biến có **màu giá trị** (domain) là các số từ 1 đến 8 (vì 1 ô luôn là **0** đại diện ô trống).
- Gán từng giá trị cho các biến theo thứ tự từ trái sang phải, **không lặp lại** các giá trị đã dùng. **Các ràng buộc được áp dụng:**

1. Ràng buộc toàn cục (Global Constraint):

- Mỗi số từ 1 đến 8 chỉ được gán **một lần duy nhất**.
- Trạng thái cuối cùng phải khớp với trạng thái **mục tiêu (goal state)**.

2. Ràng buộc cục bộ (Local Constraint) – tùy chọn tăng độ chính xác và cắt nhánh:

- Khi gán giá trị mới, kiểm tra hiệu tuyệt đối với giá trị vừa gán trước đó phải nhỏ hơn 2:
`abs(value - last_value) < 2`
- Điều này giúp giảm số nhánh không cần thiết trong không gian tìm kiếm.

► Ưu điểm:

- Ràng buộc rõ ràng giúp **giảm mạnh không gian tìm kiếm**.
- Đơn giản, dễ cài đặt và dễ trực quan hóa quá trình hoạt động.
- Không cần heuristic hay mô hình môi trường – chỉ cần mô tả ràng buộc.
- Có thể **giải quyết các biến thể của bài toán 8 Puzzle** có tính ràng buộc.

► Nhược điểm:

- **Không tìm ra lời giải tối ưu**, chỉ trả về lời giải đầu tiên hợp lệ.
- **Không đảm bảo luôn tìm được lời giải**, nếu ràng buộc quá chặt có thể không tồn tại lời giải.
- Dễ bị **bùng nổ tổ hợp** nếu không có thêm chiến lược cắt tỉa ràng buộc (constraint propagation).

► **Độ phức tạp:**

- **Thời gian:**
 - Trung bình là $O(d^n)$, với:
 - $d = 8$ (số giá trị cần gán)
 - $n = 8$ (số ô chứa số)
 - Tệ nhất duyệt hết tất cả hoán vị: ($8! = 40,320$) trạng thái.
 - Việc áp dụng ràng buộc cục bộ giúp **giảm đáng kể số lần thử**.
- **Bộ nhớ:**
 - Tối đa $O(n)$ cho ngăn xếp đệ quy (tối đa 8 cấp).
 - Không cần lưu trữ toàn bộ không gian.

So sánh hiệu suất

🔍 Bảng so sánh các thuật toán tìm kiếm

Thuật toán	Tối ưu?	Hoàn tất?	Có dùng hàm chi phí?	Dựa vào heuristic?	Dễ implement?	Bộ nhớ	Ghi chú
Breadth-First Search	✓	✓	✗	✗	✓	✗ Cao	Tìm lời giải ngắn nhất
Depth-First Search	✗	✗	✗	✗	✓✓	✓ Thấp	Dễ bị lặp, có thể đi sâu vô tận
Uniform Cost Search	✓	✓	✓	✗	✓	✗ Cao	Ưu tiên đường đi rẻ nhất
Iterative Deepening Search	✓	✓	✗	✗	✓	✓ Thấp	Kết hợp DFS + BFS
Greedy Best-First Search	✗	✗	✗	✓	✓	✗ Cao	Nhanh, dễ đi sai hướng
A* Search	✓	✓	✓	✓	✗ Trung bình	✗ Cao	Cần hàm heuristic tốt
Iterative Deepening A*	✓	✓	✓	✓	✗ Khó hơn	✓ Tốt	A* tiết kiệm RAM
Simple Hill Climbing	✗	✗	✗	✓	✓	✓ Thấp	Dễ kẹt tại cực trị địa phương
Steepest Hill Climbing	✗	✗	✗	✓	✓	✓ Thấp	Chọn hướng tốt nhất tại mỗi bước
Stochastic Hill Climbing	✗	✗	✗	✓	✓	✓ Thấp	Chọn ngẫu nhiên từ các hướng tốt
Simulated Annealing	✗	✓	✗	✓	✗ Trung bình	✓ Thấp	Tránh cực trị bằng cách giảm nhiệt độ dần
Beam Search	✗	✗	✗	✓	✗ Trung bình	✗ Giới hạn	Chỉ giữ K node tốt nhất
AND-OR Graph Search	✓	✓	✓	✓	✗ Khó	✗ Cao	Cho bài toán có phân nhánh logic (phi tuyến)

Thuật toán	Tối ưu?	Hoàn tất?	Có dùng hàm chi phí?	Dựa vào heuristic?	Dễ implement?	Bộ nhớ	Ghi chú
Belief Search	✗	✗	✓	✓	✗ Khó	✗ Cao	Môi trường không chắc chắn
Q-Learning (Reinforcement)	✓ dần	✓ dần	✓	✗	✗ Phức tạp	✗ Cao	Học từ tương tác môi trường

➡ Hướng dẫn sử dụng nút Compare để so sánh các thuật toán

📁 File: puzzleApp.py

Trong hàm `compare_algorithms`, bạn có thể chọn các thuật toán cần so sánh bằng cách bật (bỏ comment #) hoặc tắt (comment lại) tại danh sách `algorithms`.

```
algorithms = [
    # Uniform Cost Search và các thuật toán không sử dụng heuristic
    # ( bfs, "BFS"),
    # ( dfs, "DFS"),
    # ( ids, "IDS"),
    # ( ucs, "UCS"),

    # Heuristic-based Search (có sử dụng hàm đánh giá)
    # ( a_star_manhattan, "A* Manhattan"),
    # ( ida_star_manhattan, "IDA* Manhattan"),
    # ( greedy_FS, "Greedy FS"),

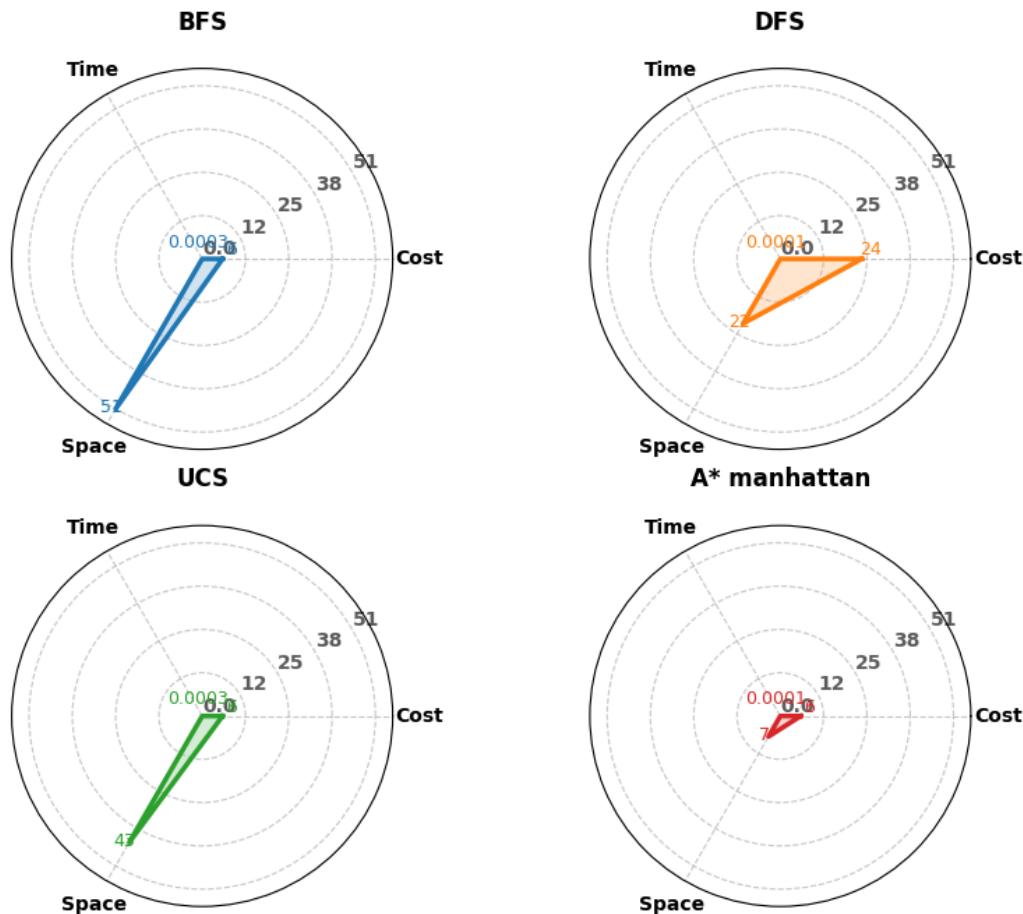
    # Local Search (tìm kiếm cục bộ)
    ( simple_hill_climbing, "Simple Hill Climb"),
    ( steepest_hill_climbing, "Steepest Hill Climb"),
    ( stochastic_hill_climbing, "Stochastic Hill Climb"),

    # Các thuật toán khác (bật nếu muốn)
    # ( simulated_annealing, "SimuAnnealing"),
    # ( beam_search, "BeamSearch"),
    # ( and_or_search, "AND-OR"),
    # ( belief, "BELIEF"),
    # ( lambda state: q_learning(state, episodes=1000), "Q-Learning"),
]
```

📊 So sánh trực quan các thuật toán bằng biểu đồ radar

Hình dưới đây thể hiện so sánh 4 thuật toán: **BFS**, **DFS**, **UCS**, và **A Manhattan*** theo 3 tiêu chí:

- ⌚ **Time**: Thời gian thực thi
- 💾 **Space**: Bộ nhớ sử dụng (số node đã mở)
- ✳️ **Cost**: Độ dài đường đi (cost của lời giải)



🔍 Phân tích biểu đồ:

Thuật toán	Time (s)	Cost	Space	Nhận xét chung
BFS	~0.0063	6	51	Tìm được đường đi ngắn, nhưng tốn nhiều bộ nhớ
DFS	~0.0010	24	22	Rất nhanh, dùng ít bộ nhớ, nhưng không tối ưu đường đi
UCS	~0.0063	6	43	Tối ưu chi phí như BFS nhưng tiết kiệm bộ nhớ hơn
A star	~0.0018	6	7	Hiệu quả nhất: nhanh, ít bộ nhớ, chi phí thấp nhất

☑ Kết luận:

- **DFS** rất nhanh nhưng **không đảm bảo lời giải tối ưu** (Cost cao nhất).
- **A Manhattan*** có hiệu suất tốt nhất về cả ba tiêu chí, nhờ heuristic thông minh.
- **BFS** và **UCS** tìm lời giải tối ưu nhưng tốn thời gian và bộ nhớ hơn A*.

Hướng dẫn sử dụng

Cài đặt

1. Tải và cài Python:

- Tải Python 3.6+ từ python.org.
- Cài đặt, tích chọn **Add Python to PATH**.
- Kiểm tra: `python --version` trong terminal (Command Prompt hoặc Terminal).

2. Cài thư viện:

```
pip install matplotlib numpy pygame
```

3. Clone project từ GitHub:

```
git clone https://github.com/HuyinCP/8-Puzzle-AI-solver-.git
```

4. Chạy chương trình trong terminal:

```
python main.py
```