8-Puzzle Solver Agent

Trò chơi giải đố 8-puzzle với nhiều nhóm thuật toán tìm kiếm AI

BÁO CÁO TỔNG KẾT DỰ ÁN

1. Mục Tiêu

Dự án 8-Puzzle Solver Game được xây dựng nhằm phát triển một ứng dụng giải đố 8-puzzle kết hợp nhiều thuật toán tìm kiếm AI, với điểm nhấn là giao diện đồ họa dễ sử dụng và khả năng hiển thị trực quan toàn bộ quá trình giải. Các mục tiêu cụ thể bao gồm:

Thiết kế một trò chơi 8-puzzle có giao diện đồ họa sinh động bằng Pygame, cho phép người dùng tùy chỉnh trạng thái đầu vào, theo dõi diễn biến bàn cờ và tương tác với quá trình giải thông qua các nút thuật toán.

Tích hợp đa dạng các thuật toán tìm kiếm thuộc 6 nhóm chính: tìm kiếm không định hướng (Uninformed), có định hướng (Informed), tìm kiếm cục bộ (Local Search), giải quyết bài toán ràng buộc (CSP), môi trường không chắc chắn (Complex Environments) và học tăng cường (Reinforcement Learning). Tất cả đều được áp dụng trên cùng một bài toán để dễ dàng so sánh.

Thực hiện phân tích so sánh chi tiết giữa các thuật toán, chỉ rõ điểm mạnh, điểm yếu và phạm vi ứng dụng phù hợp của từng chiến lược giải.

Hiển thị trực quan từng bước giải thuật trên giao diện: từ quá trình di chuyển từng ô số, trạng thái trung gian, đến tổng số bước và thời gian giải – giúp người học hoặc người trình bày dễ dàng theo dõi và hiểu rõ logic giải quyết vấn đề của từng thuật toán.

2. Nội Dung

2.1. Nhóm 1: Tìm Kiếm Không Có Thông Tin (Uninformed Search)

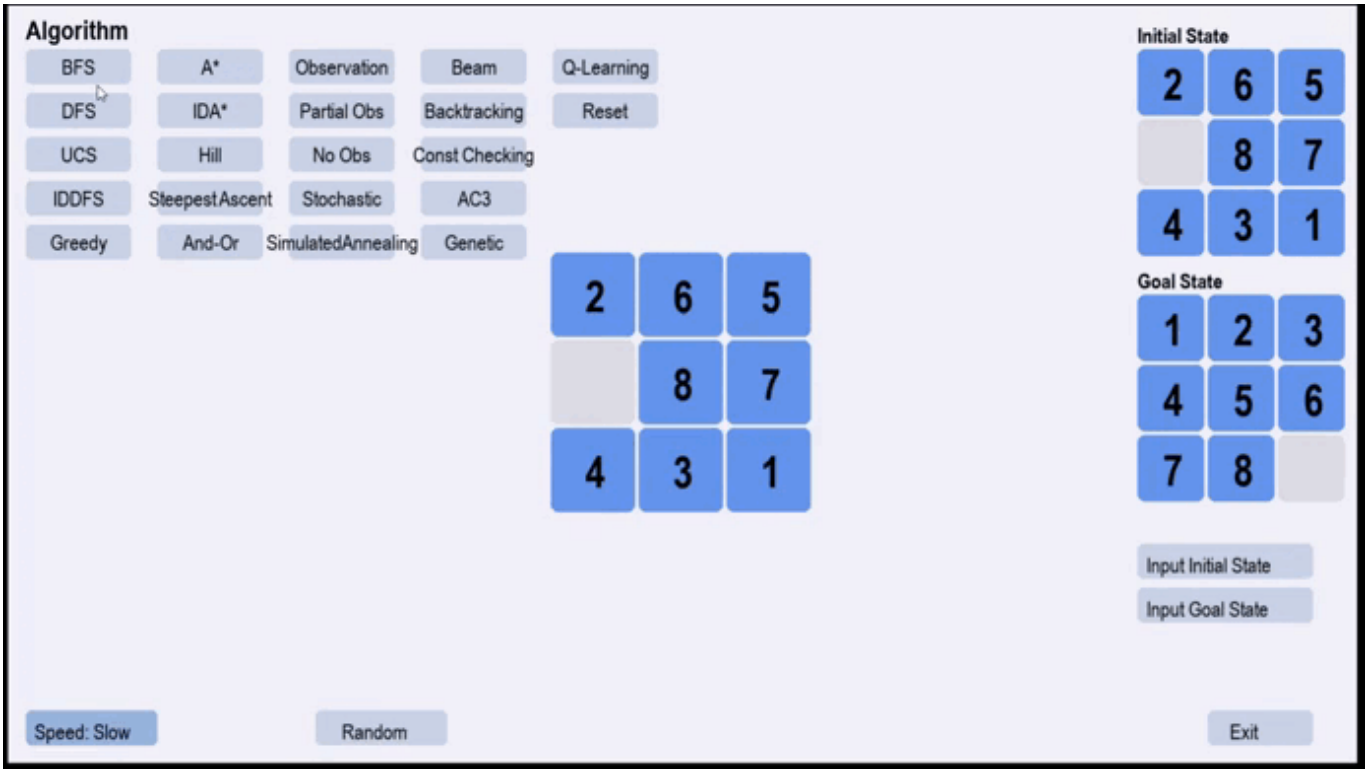
Thành phần chính của bài toán tìm kiếm:

- **Trạng thái ban đầu**: Cấu hình khởi điểm của bảng 8-puzzle, do người dùng tùy ý nhập thông qua giao diện.
- **Trạng thái đích**: Cấu hình mục tiêu cần đạt tới, thường được chuẩn hóa là `(1, 2, 3, 4, 5, 6, 7, 8, 0)`, trong đó `0` đại diện cho ô trống.
- **Hàm chi phí**: Mỗi hành động di chuyển giữa hai trạng thái có chi phí bằng 1 đơn vị.
- **Solution**: Chuỗi các bước di chuyển từ trạng thái ban đầu đến trạng thái đích

Các thuật toán tìm kiếm không có thông tin:

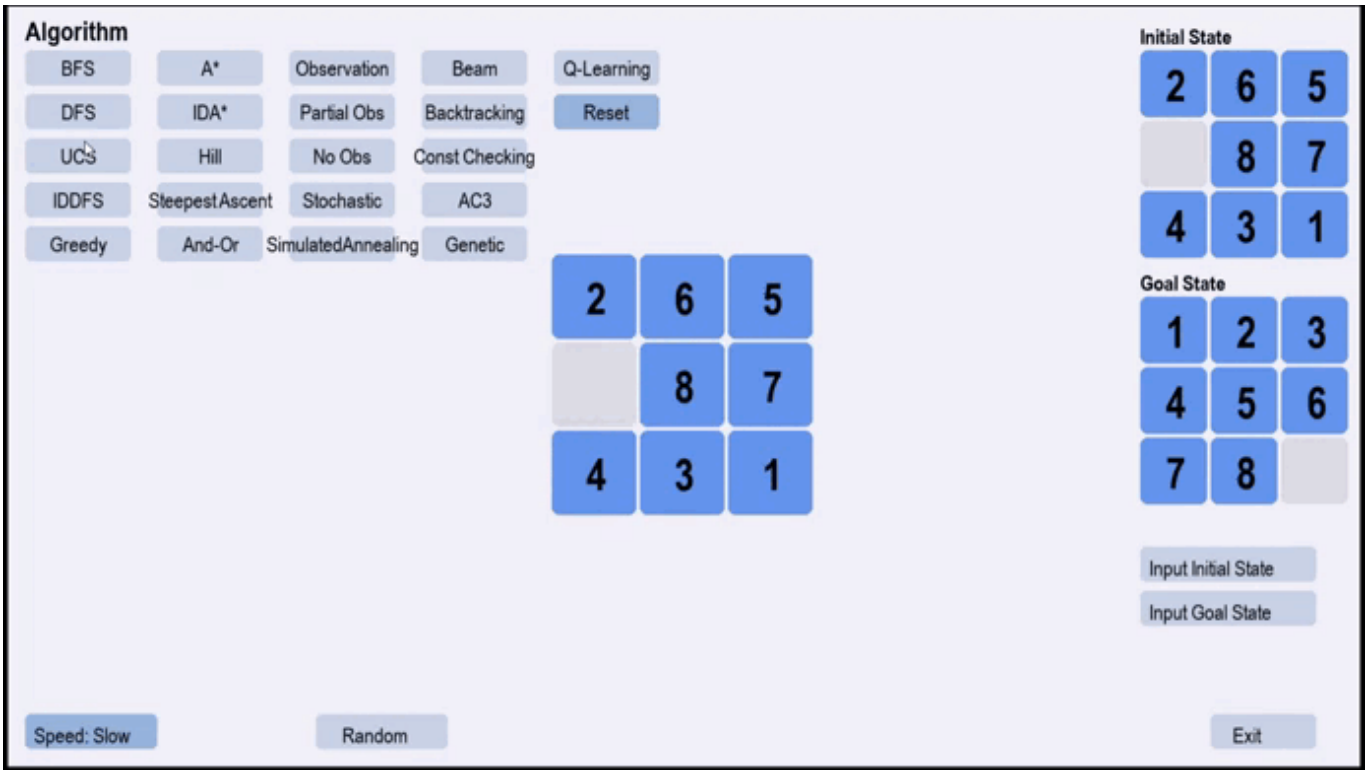
BFS (Breadth-First Search)

- **Mô tả**: Thuật toán Breadth-First Search (BFS)– Tìm kiếm theo chiều rộng là một phương pháp tìm kiếm không có thông tin (Uninformed Search), hoạt động theo nguyên lý duyệt theo chiều rộng– mở rộng lần lượt từng lớp nút trước khi đi sâu hơn. Thuật toán này thường được sử dụng trong môi trường đảm bảo, không có đối kháng, nơi mọi hành động đều dẫn đến trạng thái xác định và không có yếu tố ngẫu nhiên.
- **Minh họa**:



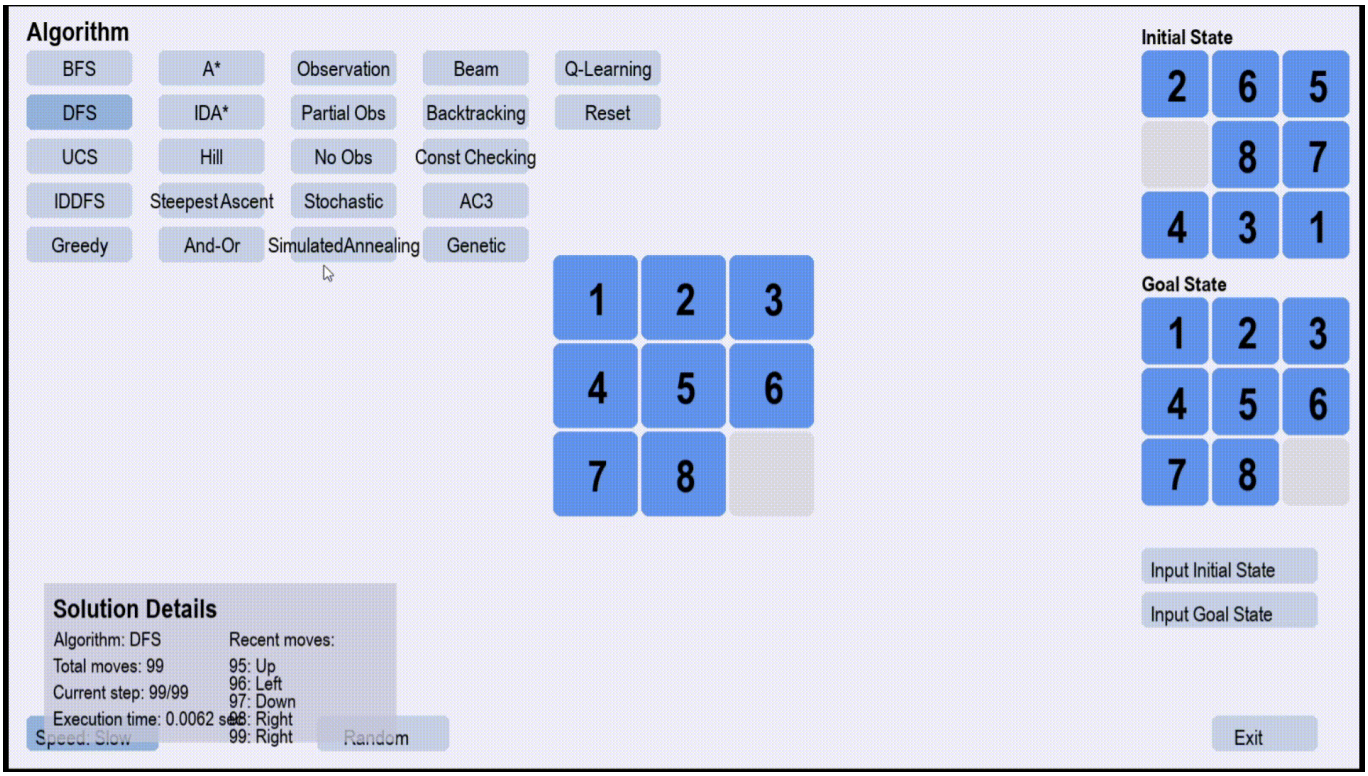
DFS (Depth-First Search)

- **Mô tả**: Thuật toán Depth-First Search (DFS)– Tìm kiếm theo chiều sâu là một phương pháp tìm kiếm không có thông tin (uninformed search), hoạt động theo chiến lược đi càng sâu càng tốt trước khi quay lui để thử nhánh khác.
- **Minh họa**:



UCS (Uniform Cost Search)

- **Mô tả**: Thuật toán Uniform Cost Search (UCS)- Tìm kiếm theo chi phí thống nhất là một phương pháp tìm kiếm không có thông tin trước (uninformed search), hoạt động dựa trên nguyên tắc mở rộng nút có chi phí thấp nhất trước. UCS tương tự như thuật toán Dijkstra, nhưng được áp dụng trong các bài toán tìm kiếm tổng quát.
- **Minh họa**:

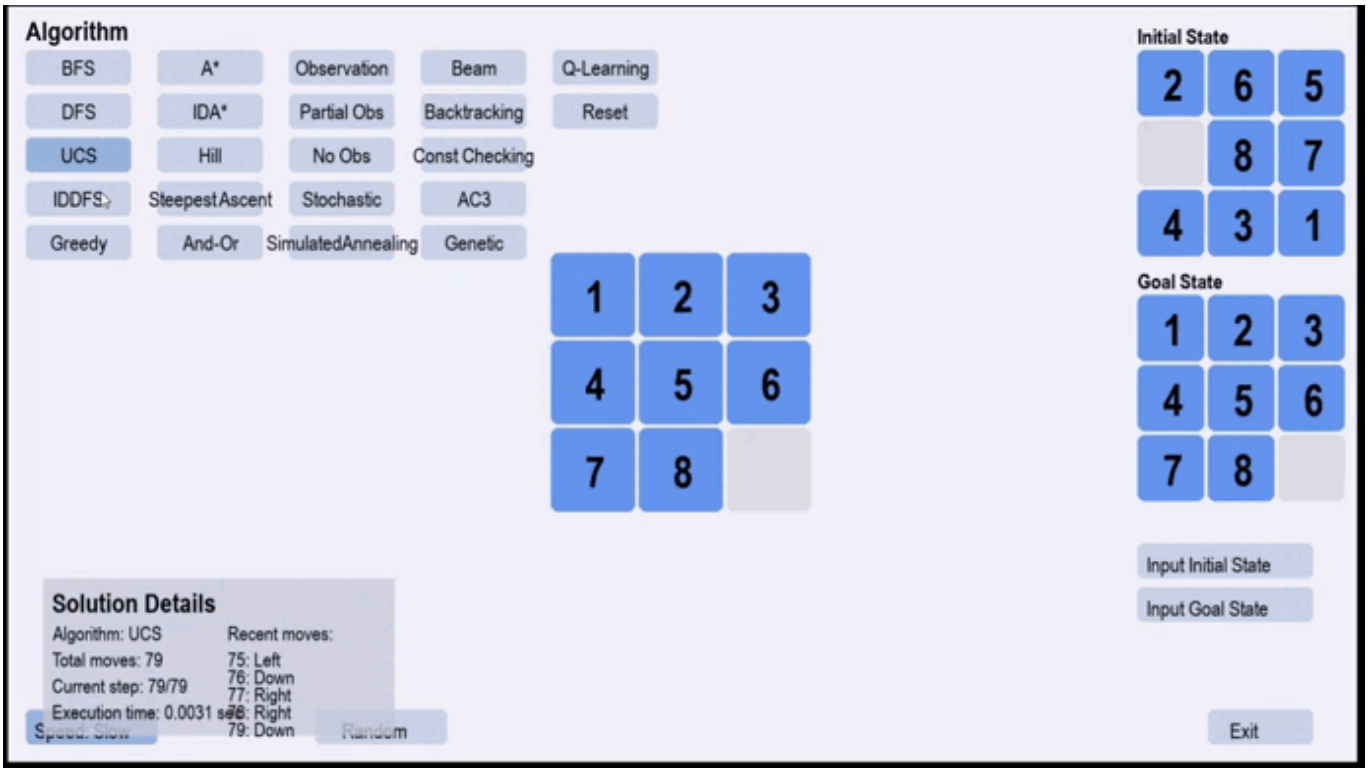


IDDFS (Iterative Deepening Depth-First Search)

- **Mô tả**: DDFS là sự kết hợp giữa hai chiến lược tìm kiếm: DFS (tìm kiếm theo chiều sâu) và BFS (tìm kiếm theo chiều rộng). Thuật toán này thực hiện DFS nhiều lần, mỗi lần với một giới hạn độ sâu tăng dần. Cụ thể, IDDFS bắt đầu bằng việc tìm kiếm từ độ sâu 0, sau đó tăng độ sâu lên 1, rồi 2, rồi 3,... cho đến khi tìm thấy lời giải hoặc đạt độ sâu tối đa.

  Mỗi vòng lặp giới hạn độ sâu sẽ thực hiện DFS đầy đủ ở mức đó, nhưng tránh được nhược điểm của DFS là đi quá sâu vào nhánh không có lời giải. Đồng thời, nó vẫn giữ được ưu điểm của BFS là tìm được lời giải ngắn nhất đầu tiên (trong không gian trạng thái có độ sâu nhỏ).

- **Minh họa**:



So sánh hiệu suất:

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Số bước tối ưu
BFS	100-300	Cao	Luôn tối ưu
DFS	20-100	Thấp	Thường không tối ưu
UCS	120–250	Trung bình	Luôn tối ưu
IDDFS	200–400	Trung bình	Luôn tối ưu

Nhận xét:

- **BFS** đảm bảo tìm được đường đi ngắn nhất nhưng tiêu tốn nhiều bộ nhớ khi độ sâu của giải pháp tăng
- **DFS** tiết kiệm bộ nhớ nhưng không đảm bảo tìm được đường đi ngắn nhất và có thể rơi vào vòng lặp vô hạn
- **UCS** tương tự BFS trong bài toán 8-puzzle (vì mỗi bước di chuyển có chi phí bằng nhau), nhưng hiệu quả hơn trong các bài toán có chi phí khác nhau
- **IDDFS** kết hợp ưu điểm của BFS (đảm bảo tìm được đường đi ngắn nhất) và DFS (tiết kiệm bộ nhớ), nhưng có thể tốn thời gian do phải duyệt lại các nút nhiều lần

2.2. Nhóm 2: Thuật Toán Tìm Kiếm Có Thông Tin (Informed Search)

Thành phần chính của bài toán tìm kiếm:

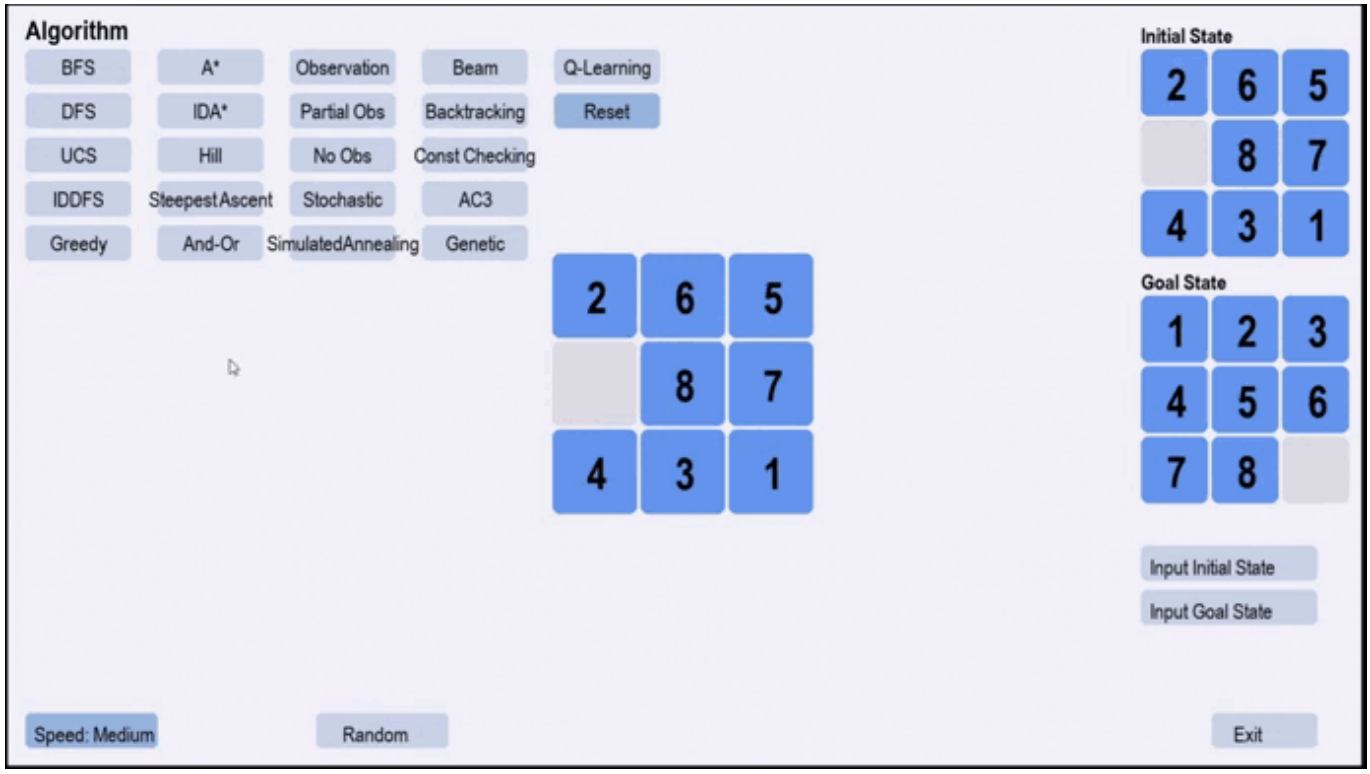
- **Trạng thái ban đầu**: Cấu hình khởi điểm của bảng 8-puzzle, do người dùng tùy ý nhập thông qua giao diện.
- **Trạng thái đích**: Cấu hình mục tiêu cần đạt tới, thường được chuẩn hóa là `(1, 2, 3, 4, 5, 6, 7, 8, 0)`, trong đó `0` đại diện cho ô trống.

- **Hàm chi phí**: Mỗi hành động di chuyển giữa hai trạng thái có chi phí bằng 1 đơn vị.
- **Hàm heuristic (h(n))**: Ước lượng chi phí còn lại từ trạng thái hiện tại (x1, y1) đến trạng thái đích, đóng vai trò dẫn đường cho quá trình tìm kiếm.
    - **Ví dụ heuristic phổ biến**:
        - **Manhattan distance**: Tổng khoảng cách Manhattan của mỗi ô từ vị trí hiện tại (x1, y1) đến vị trí đích (x2, y2), với công thức là |x1 - x2| + |y1 - y2|.
- **Solution (Lời giải)**: Là chuỗi các hành động hợp lệ dẫn từ trạng thái ban đầu đến trạng thái đích với tổng chi phí thấp nhất theo đánh giá của thuật toán.

Các thuật toán tìm kiếm có thông tin:

Greedy Search

- **Mô tả**: Thuật toán Greedy (Tìm kiếm tham lam) là một phương pháp tìm kiếm có thông tin, thường được sử dụng trong môi trường xác định, có thể ước lượng khoảng cách đến mục tiêu bằng heuristic mà cụ thể là dựa trên khoảng cách Manhattan
- **Minh họa**:

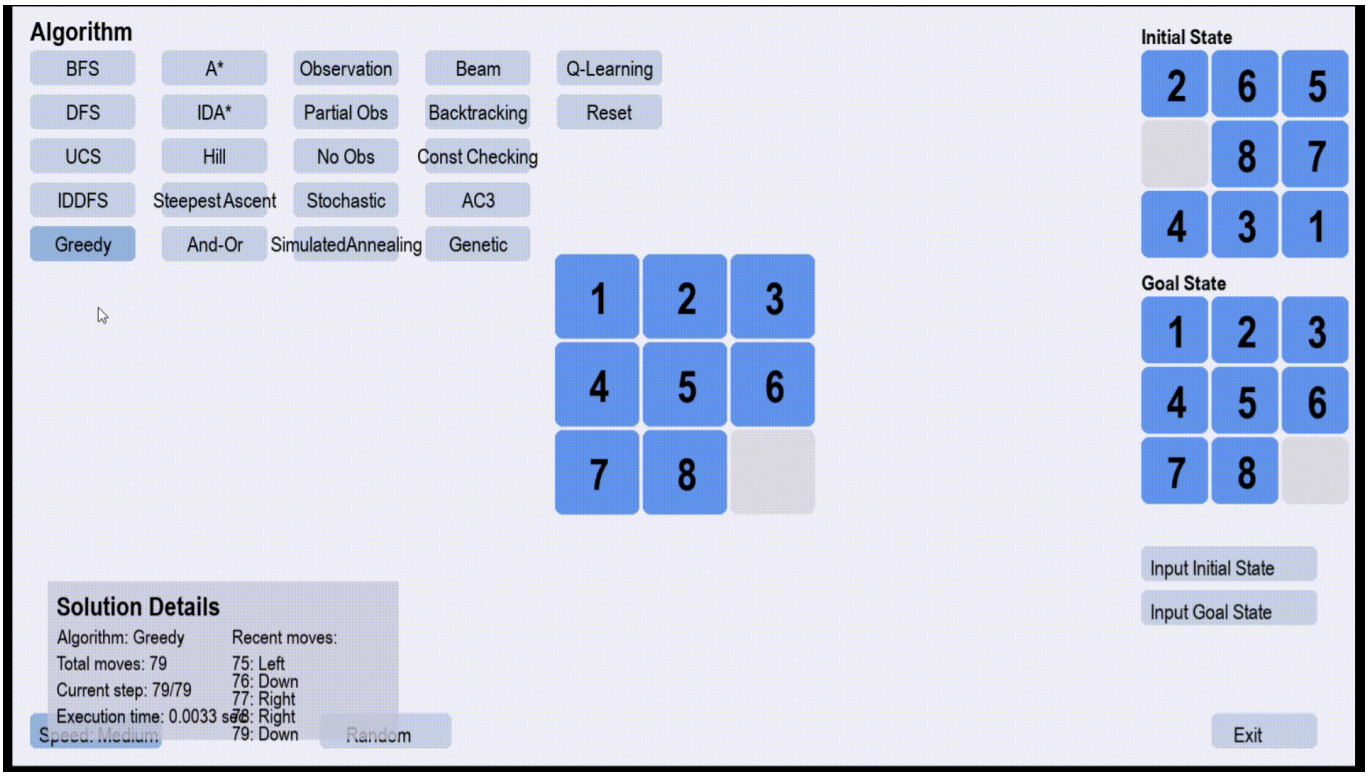


A* (A Star)

- **Mô tả**:Thuật toán A* là một phương pháp tìm kiếm có thông tin trước (informed search), kết hợp giữa chi phí thực tế đã đi (g(n)) và chi phí ước lượng còn lại đến đích (h(n), heuristic).

  Khác với UCS chỉ dựa vào chi phí thực tế, A* sử dụng công thức f(n) = g(n) + h(n) để mở rộng các nút có tiềm năng tốt nhất, tức là vừa gần điểm xuất phát vừa gần mục tiêu.

- **Minh họa**:



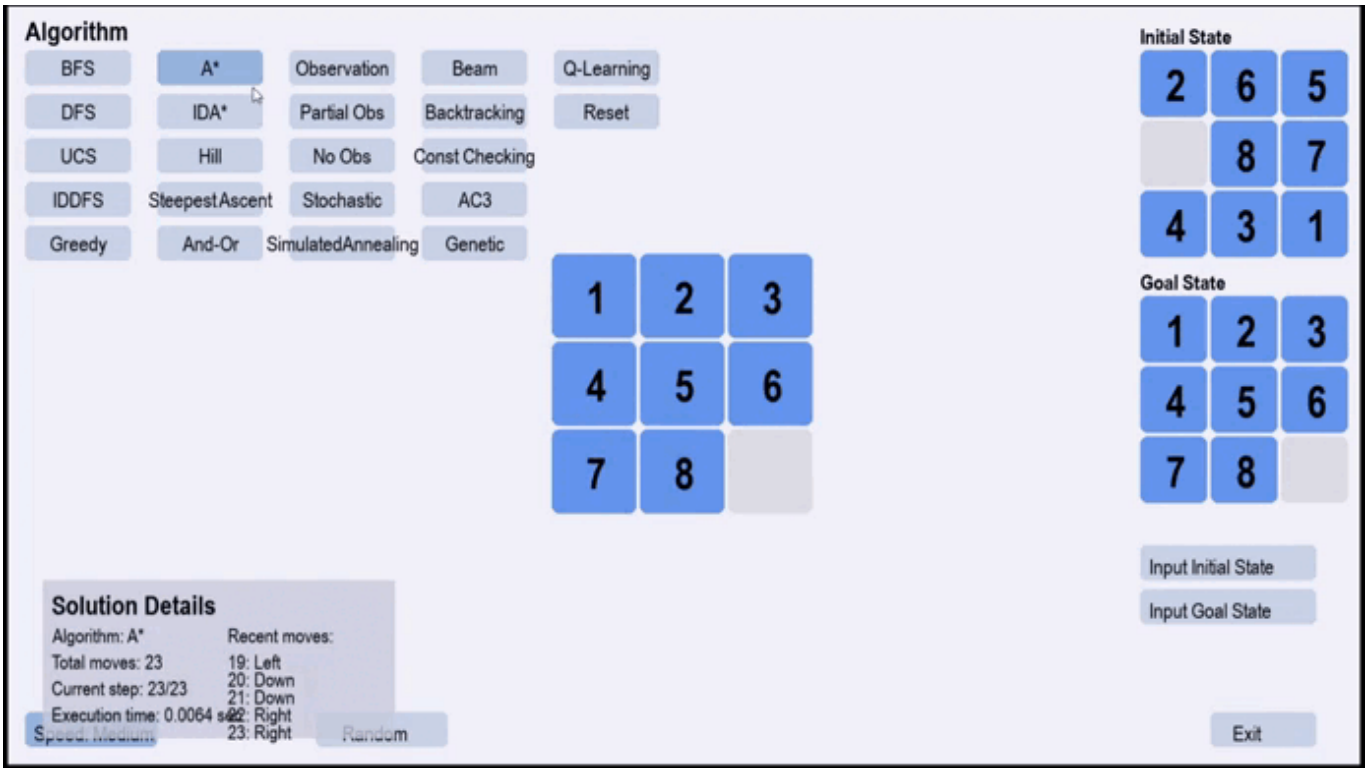
IDA* (Iterative Deepening A Star)

- **Mô tả**: IDA* là phiên bản kết hợp giữa hai chiến lược mạnh mẽ: A* (tìm kiếm tối ưu theo chi phí) và DFS có giới hạn độ sâu, nhằm khắc phục nhược điểm về bộ nhớ của A*.

  Thuật toán hoạt động theo nguyên tắc tìm kiếm theo tầng với giới hạn f(n) – tức là tổng chi phí thực tế từ đầu đến đỉnh n (g(n)) cộng với ước lượng chi phí còn lại đến đích (h(n)). Ban đầu, thuật toán sử dụng giá trị f(start) = g(start) + h(start) làm ngưỡng giới hạn, và thực hiện tìm kiếm theo chiều sâu trong phạm vi giới hạn đó.

  Nếu không tìm thấy lời giải trong ngưỡng này, IDA* sẽ nâng ngưỡng lên bằng giá trị f(n) nhỏ nhất vượt quá giới hạn trong lần lặp trước, và lặp lại quá trình tìm kiếm. Cứ như vậy, thuật toán sẽ tiến dần đến lời giải, đảm bảo tìm ra đường đi tối ưu (giống A*) nhưng chỉ cần dùng bộ nhớ tuyến tính như DFS, thay vì lưu toàn bộ hàng đợi ưu tiên như A*.

- **Minh họa**:



So sánh hiệu suất:

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Số bước tối ưu
Greedy	100-150	Thấp	Thường không tối ưu
A*	100-200	Trung bình	Luôn tối ưu
IDA*	150-300	Thấp	Luôn tối ưu

Nhận xét:

- **Greedy Search** rất nhanh nhưng không đảm bảo tìm được đường đi ngắn nhất
- **A*** kết hợp hiệu quả giữa UCS và Greedy, đảm bảo tìm được đường đi ngắn nhất nếu heuristic admissible
- **IDA*** có hiệu suất bộ nhớ tốt hơn A* nhưng có thể chậm hơn do phải duyệt lại các nút

2.3. Nhóm 3: Thuật Toán Tìm Kiếm Cục Bộ (Local Search)

Thành phần chính của bài toán tìm kiếm:

- **Trạng thái ban đầu**: Cấu hình khởi điểm của bảng 8-puzzle, được người dùng nhập tùy ý thông qua giao diện.
- **Trạng thái đích**: Cấu hình mục tiêu cần đạt tới, thường là `(1, 2, 3, 4, 5, 6, 7, 8, 0)`, trong đó `0` đại diện cho ô trống.
- **Hàng xóm (Neighbors)**: Tập các trạng thái có thể sinh ra từ trạng thái hiện tại bằng một bước di chuyển hợp lệ. Local Search chỉ xét trạng thái hàng xóm trực tiếp thay vì xây dựng toàn bộ cây tìm kiếm.
- **Hàm đánh giá (Evaluation function)**: Hàm dùng để đánh giá "độ tốt" của một trạng thái hiện tại, thường dựa trên khoảng cách đến trạng thái đích.

    - **Ví dụ phổ biến**:
        - **Manhattan distance**: Tổng khoảng cách Manhattan của tất cả các ô (trừ ô trống) từ vị trí hiện tại đến vị trí đúng trong trạng thái đích, tính theo công thức: `|x1 - x2| + |y1 - y2|`.
- **Solution (Lời giải)**: Là một trạng thái gần với mục tiêu hoặc đạt được mục tiêu, được tìm thông qua quá trình cải thiện dần từ trạng thái ban đầu sang trạng thái tốt hơn trong không gian hàng xóm.

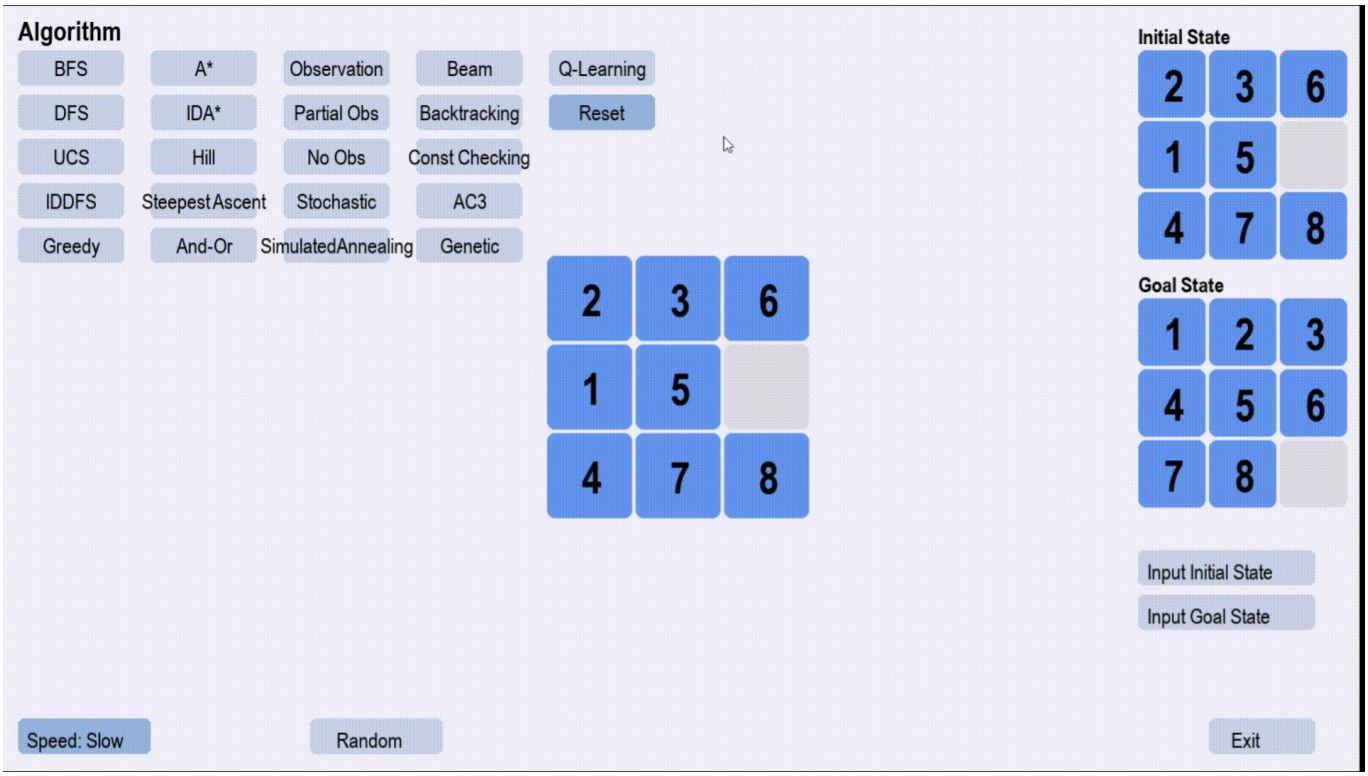
Các thuật toán tìm kiếm cục bộ:

Hill Climbing

- **Mô tả**: Hill Climbing là một thuật toán tìm kiếm cục bộ đơn giản và trực quan, mô phỏng quá trình leo lên đỉnh của một ngọn đồi bằng cách luôn di chuyển từ trạng thái hiện tại sang trạng thái hàng xóm có giá trị tốt hơn dựa trên một hàm đánh giá (heuristic).
- **Minh họa**:

Steepest-Ascent Hill Climbing

- **Mô tả**: Steepest-Ascent Hill Climbing là một thuật toán tìm kiếm cục bộ (local search) dựa trên nguyên tắc leo đồi, trong đó Pacman luôn chọn bước đi tiếp theo có giá trị heuristic cao nhất trong số các lựa chọn lân cận. Thuật toán chỉ quan tâm đến vùng lân cận hiện tại (không nhớ trạng thái toàn cục).
- **Minh họa**:



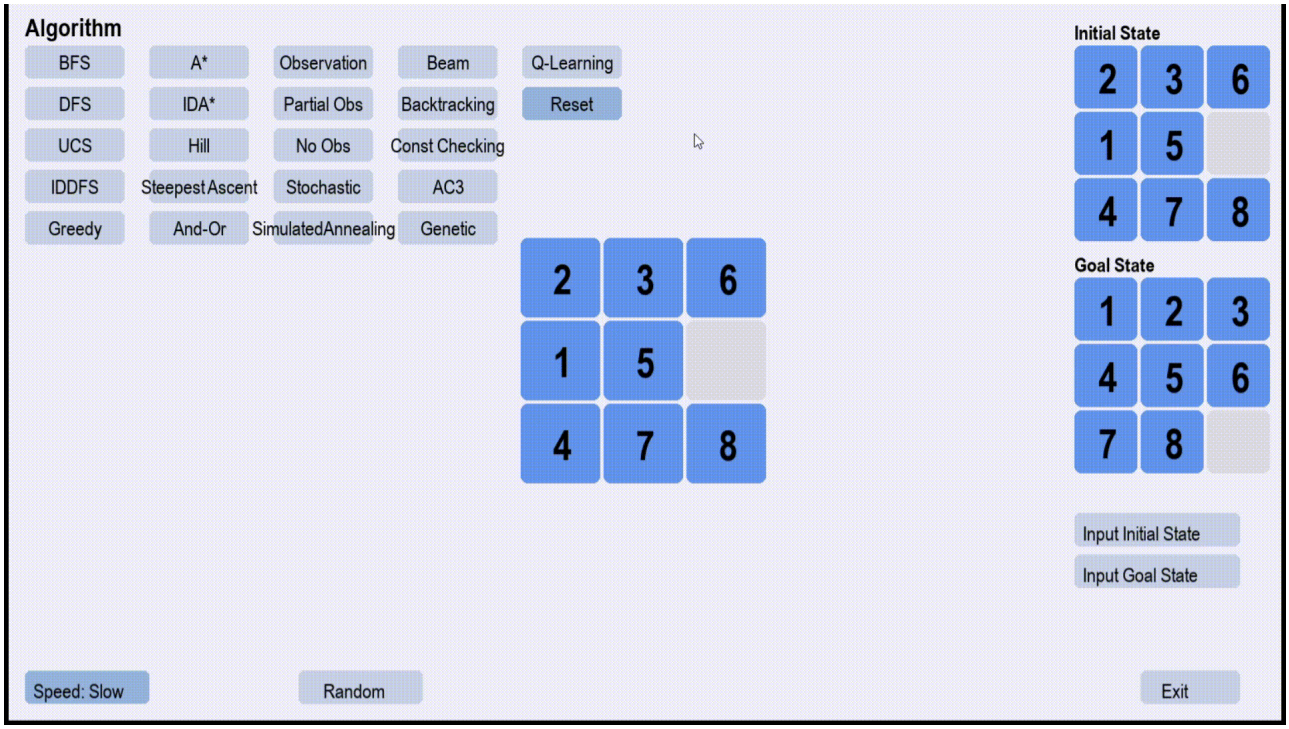
Stochastic Hill Climbing

- **Mô tả**: Stochastic Hill Climbing là một biến thể của Hill Climbing nhằm giảm nguy cơ mắc kẹt ở cực trị cục bộ bằng cách chọn ngẫu nhiên một trong các trạng thái hàng xóm tốt hơn thay vì luôn chọn trạng thái tốt nhất.

  Cụ thể, tại mỗi bước, thuật toán đánh giá tất cả các trạng thái hàng xóm sinh ra từ trạng thái hiện tại. Sau đó, nó lọc ra các trạng thái có giá trị heuristic tốt hơn (tức là gần trạng thái đích hơn). Trong số các trạng thái này, một trạng thái sẽ được chọn ngẫu nhiên để tiếp tục quá trình tìm kiếm. Bằng cách không luôn chọn trạng thái "tốt nhất", thuật toán này có thể vượt qua các vùng bằng phẳng (plateaus) hoặc tránh bị kẹt tại các đỉnh cục bộ (local maxima) như thuật toán Hill Climbing truyền thống.

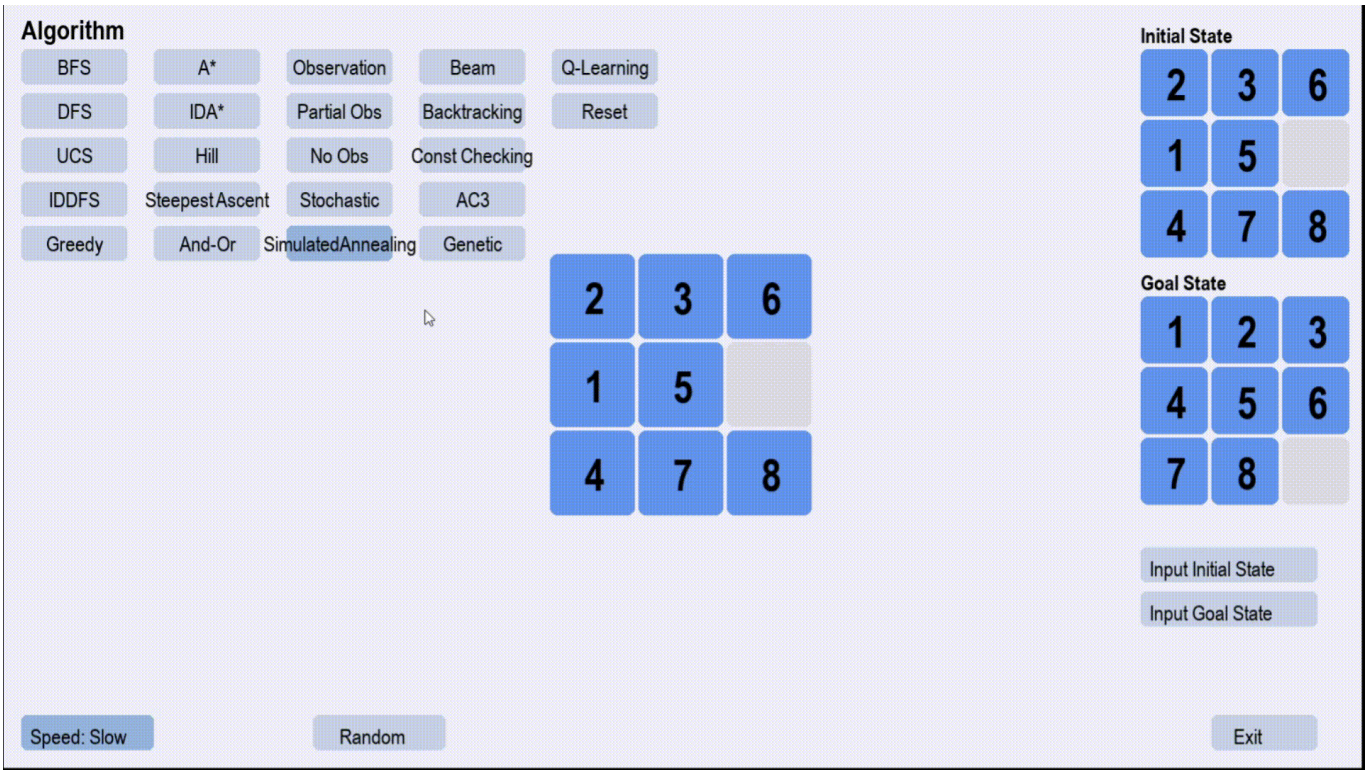
  Stochastic Hill Climbing giúp đa dạng hóa hành vi tìm kiếm, có khả năng khám phá không gian trạng thái rộng hơn, tuy nhiên vẫn không đảm bảo tìm được lời giải tối ưu. Độ hiệu quả phụ thuộc nhiều vào may mắn trong việc chọn hướng đi.

- **Minh họa**:



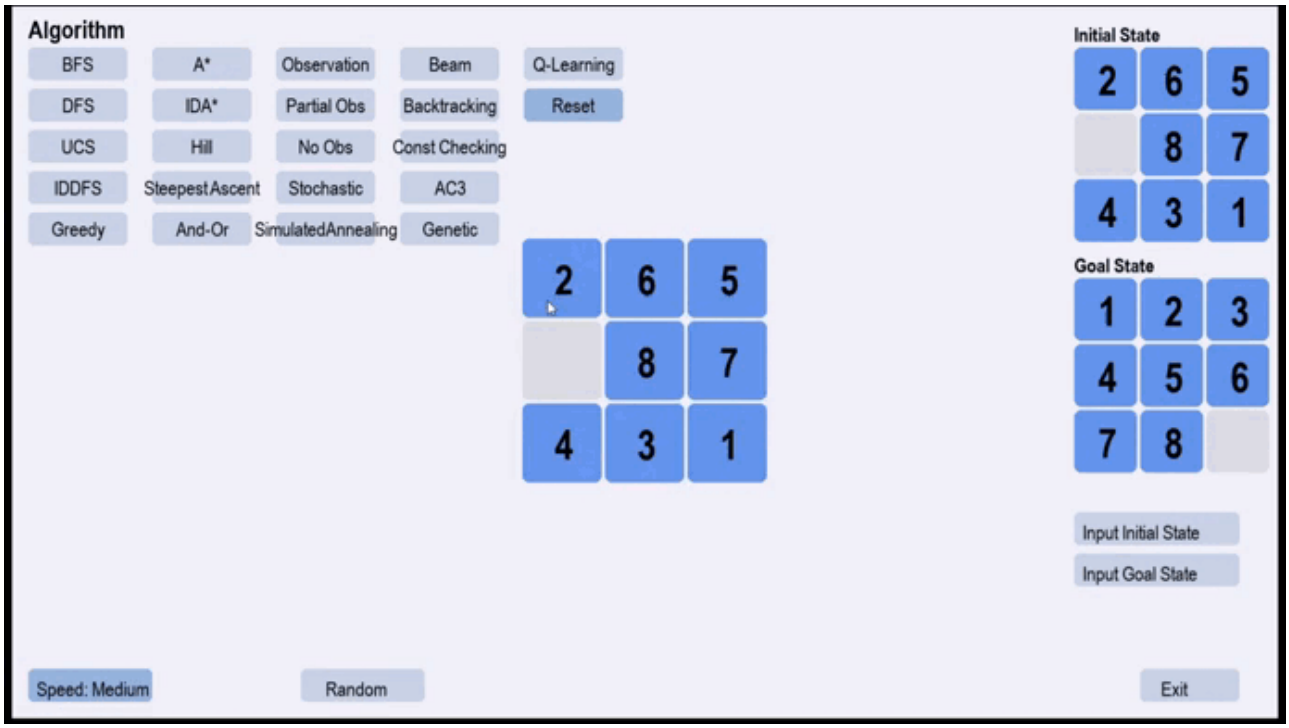
Simulated Annealing

- **Mô tả**: Simulated Annealing (SA) là thuật toán tìm kiếm cục bộ lấy cảm hứng từ quá trình tôi luyện kim loại trong vật lý, nơi kim loại được nung nóng rồi làm nguội dần để đạt được cấu trúc tối ưu. Trong 8-puzzle, SA được sử dụng để thoát khỏi cực trị cục bộ (local optimum) – tình huống mà mọi trạng thái lân cận đều tệ hơn, khiến các thuật toán như Hill Climbing bị "mắc kẹt".
- **Minh họa**:



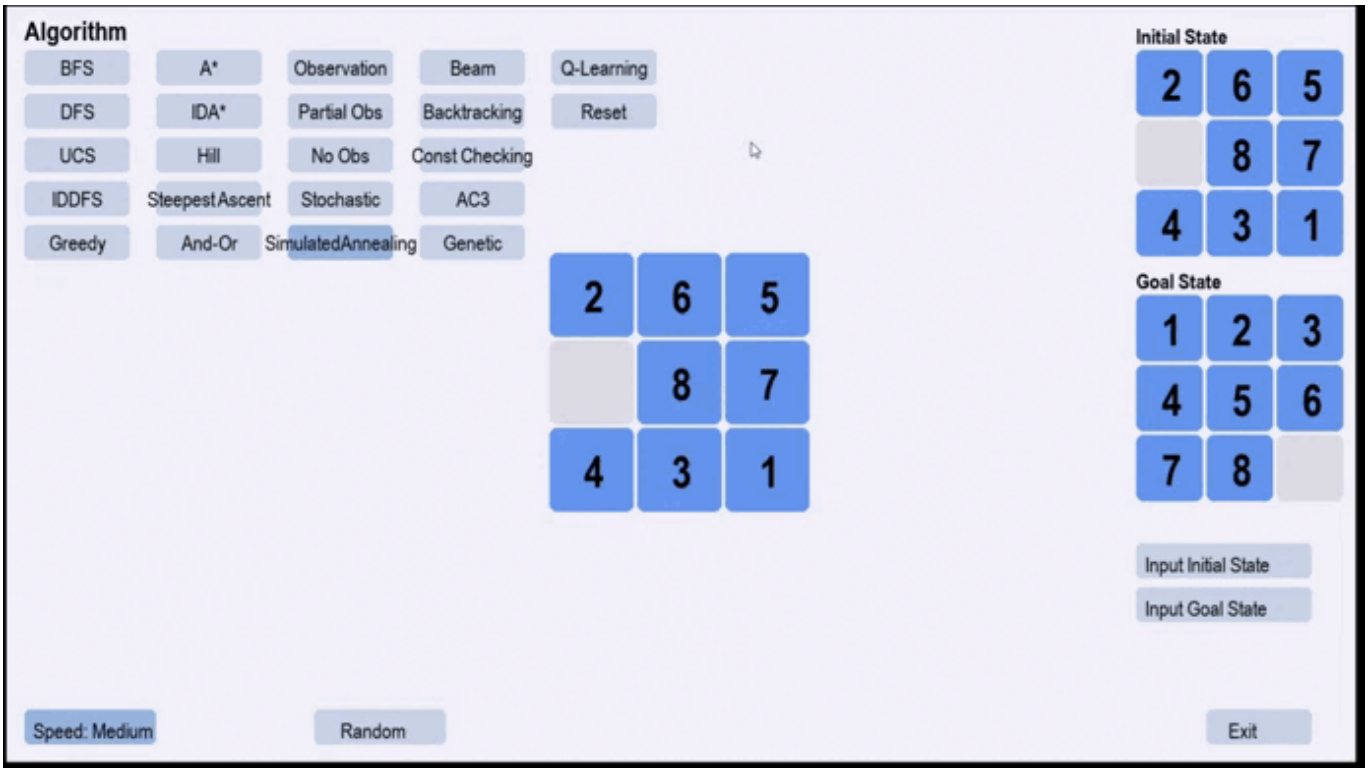
Genetic Algorithm

- **Mô tả**: Sử dụng các nguyên tắc di truyền để tiến hóa dần đến giải pháp tốt
- **Minh họa**:



Beam Search

- **Mô tả**: Duy trì beam_width trạng thái tốt nhất tại mỗi cấp độ
- **Minh họa**:



So sánh hiệu suất:

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Khả năng tìm lời giải

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Khả năng tìm lời giải
Hill Climbing	50-100	Rất thấp	Có thể bị kẹt
Steepest-Ascent HC	50-100	Rất thấp	Có thể bị kẹt
Stochastic HC	60-120	Rất thấp	Có thể bị kẹt
Simulated Annealing	100-150	Rất thấp	Thường gần tối ưu
Genetic Algorithm	200-350	Trung bình	Có thể tìm ra giải pháp tốt
Beam Search	100-250	Trung bình	Tìm được giải pháp tốt nhất

Nhận xét:

- Các thuật toán **Hill Climbing** rất nhanh và ít tốn bộ nhớ, nhưng dễ bị kẹt ở cực trị cục bộ
- **Simulated Annealing** giải quyết được vấn đề kẹt ở cực trị cục bộ nhưng có thể mất nhiều thời gian hơn
- **Genetic Algorithm** đa dạng trong việc tìm kiếm không gian trạng thái nhưng phức tạp hơn và tốn thời gian
- **Beam Search** cho tốc độ tốt nhưng không đảm bảo tìm được đường đi tối ưu nếu beam_width quá nhỏ

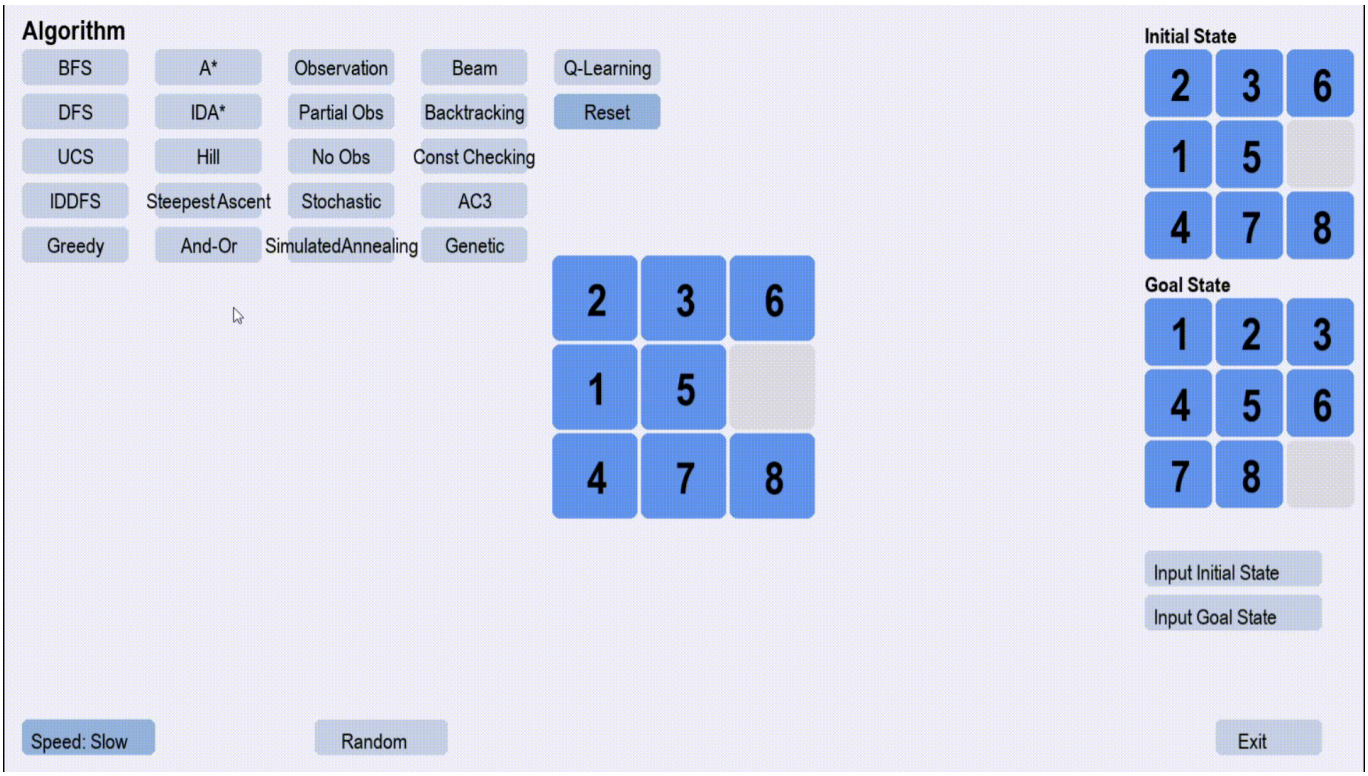
2.4. Nhóm 4: Thuật Toán Tìm Kiếm Trong Môi Trường Phức Tạp

Thành phần chính của bài toán tìm kiếm:

- **Trạng thái ban đầu**: Không còn là một trạng thái xác định duy nhất, mà là một **tập hợp các trạng thái niềm tin (belief state)** do không có đủ thông tin ban đầu.
- **Hành động (Action)**: Có thể mang tính **bất định**, nghĩa là một hành động thực hiện từ một trạng thái có thể dẫn đến nhiều kết quả khác nhau, tùy vào điều kiện môi trường.
- **Quan sát (Observation)**: Là thông tin gián tiếp thu được sau khi thực hiện hành động, dùng để **cập nhật lại tập hợp belief state** và thu hẹp khả năng nhận diện trạng thái hiện tại.
- **Solution (Lời giải)**: Không đơn thuần là một chuỗi hành động tuyến tính, mà là một **kế hoạch có cấu trúc cây (AND-OR plan)** hoặc một chiến lược hành động phù hợp cho mọi khả năng xảy ra, bất chấp việc thiếu thông tin quan sát đầy đủ hoặc môi trường thay đổi không đoán trước.

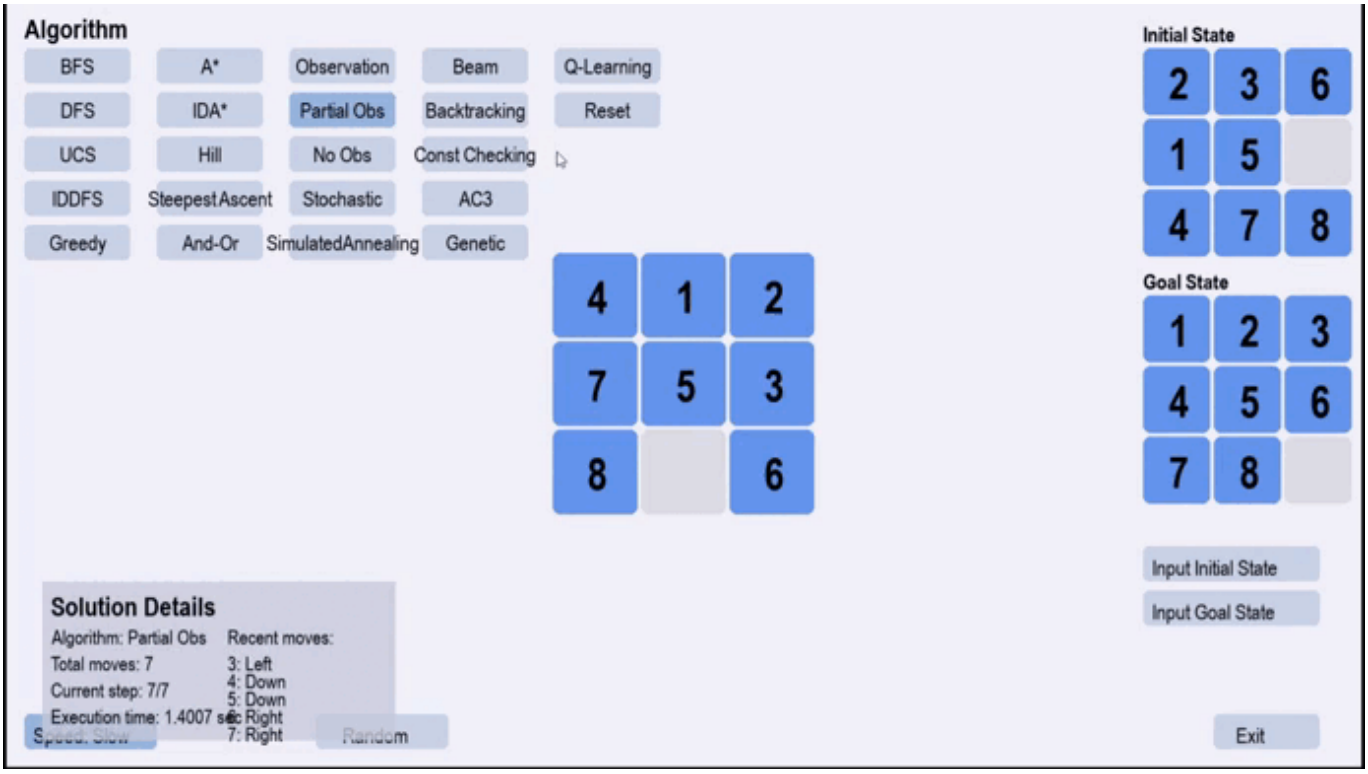
And-Or Search

- **Mô tả**: Phù hợp cho bài toán có nhiều khả năng lựa chọn và rẽ nhánh
- **Minh họa**:



No Observation Search

- **Mô tả**: Giải trong điều kiện không biết rõ trạng thái ban đầu
- **Minh họa**:



Partial Observable Search

- **Mô tả**: Xử lý bài toán khi chỉ biết một phần trạng thái môi trường
- **Minh họa**:

So sánh hiệu suất:

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Đặc điểm
And-Or Search	400-500	Cao	Tìm giải pháp tối ưu trong cây AND-OR
No Observation	300-400	Thấp	Đưa ra kết quả chính xác trong điều kiện không quan sát
Partial Observable	250-300	Trung bình	Đưa ra kết quả với thông tin quan sát một phần

Nhận xét:

- **And-Or Search** hiệu quả cho các bài toán có nhiều khả năng lựa chọn nhưng tốn nhiều bộ nhớ
- **No Observation** và **Partial Observable** giải quyết được các bài toán với thông tin không đầy đủ
- **Partial Observable** đặc biệt hữu ích trong môi trường mà người giải không thể biết chính xác trạng thái hiện tại, buộc phải dựa vào tập hợp các trạng thái khả thi (belief state) để ra quyết định

2.5. Nhóm 5: Thuật Toán Tìm Kiếm Trong Môi Trường Có Ràng Buộc (CSP)

Thành phần chính của bài toán tìm kiếm:

- **Biến (Variables)**: Bài toán 8-Puzzle bao gồm 9 ô vuông, mỗi ô được coi là một biến cần gán giá trị, đại diện cho các vị trí từ 0 đến 8 trên lưới 3x3. Các biến này tương ứng với các vị trí cụ thể trên bảng, được ký hiệu từ X0 đến X8.
- **Miền giá trị (Domain)**: Mỗi biến nhận một giá trị duy nhất trong tập {0, 1, 2, ..., 8}, trong đó 0 đại diện cho ô trống. Miền giá trị của mỗi biến được khởi tạo ngẫu nhiên và sau đó được kiểm tra để đảm bảo

không có giá trị trùng lặp trên toàn bộ bảng.

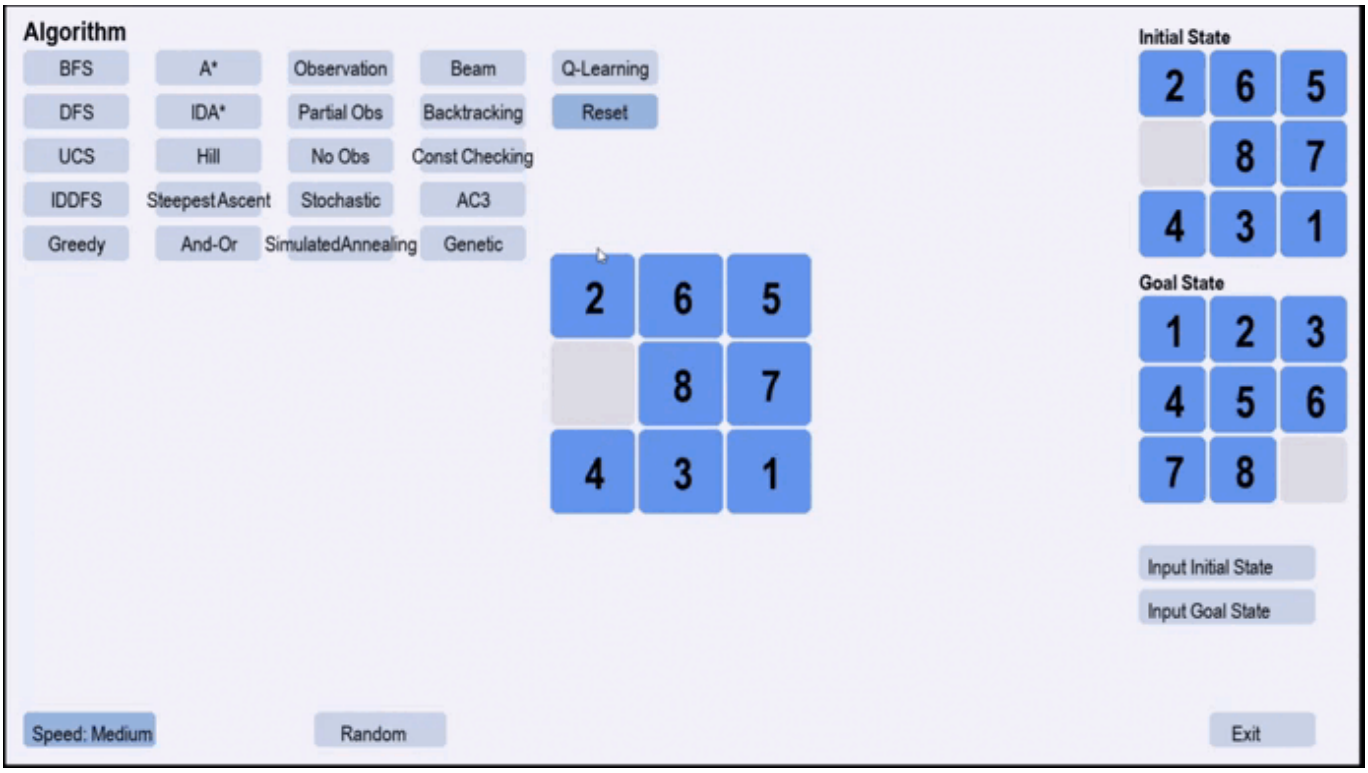
- **Ràng buộc (Constraints)**:
  - **Ràng buộc ngang**: Ràng buộc không trùng lặp (AllDifferent): Không có hai biến nào được gán cùng một giá trị. Đây là ràng buộc toàn cục, đảm bảo tính hợp lệ của trạng thái như một hoán vị hợp lệ của các số từ 0 đến 8.
  - **Ràng buộc khả thi**: Sau khi gán xong toàn bộ giá trị cho 9 biến, thuật toán sẽ kiểm tra khả năng giải bằng cách sử dụng hàm is_solvable(). Trạng thái chỉ được chấp nhận nếu nó có thể được giải theo quy tắc 8-Puzzle.
- **Kiểm tra khả năng giải (solvability check)**: Sau khi hoàn tất việc gán giá trị cho 9 biến, trạng thái cuối cùng sẽ được kiểm tra tính khả thi bằng hàm `is_solvable()`. Trạng thái chỉ hợp lệ nếu có thể giải được theo luật 8-puzzle.
- **Solution**: Gán giá trị cho 9 biến X1 đến X9, thỏa mãn các ràng buộc (ngang, dọc, không giá trị) và tạo thành một trạng thái có khả năng thực hiện đến trạng thái mục tiêu.

Backtracking

- **Mô tả**: Lần lượt gán giá trị cho từng biến và quay lui nếu phát hiện mâu thuẫn.
- **Minh họa**:

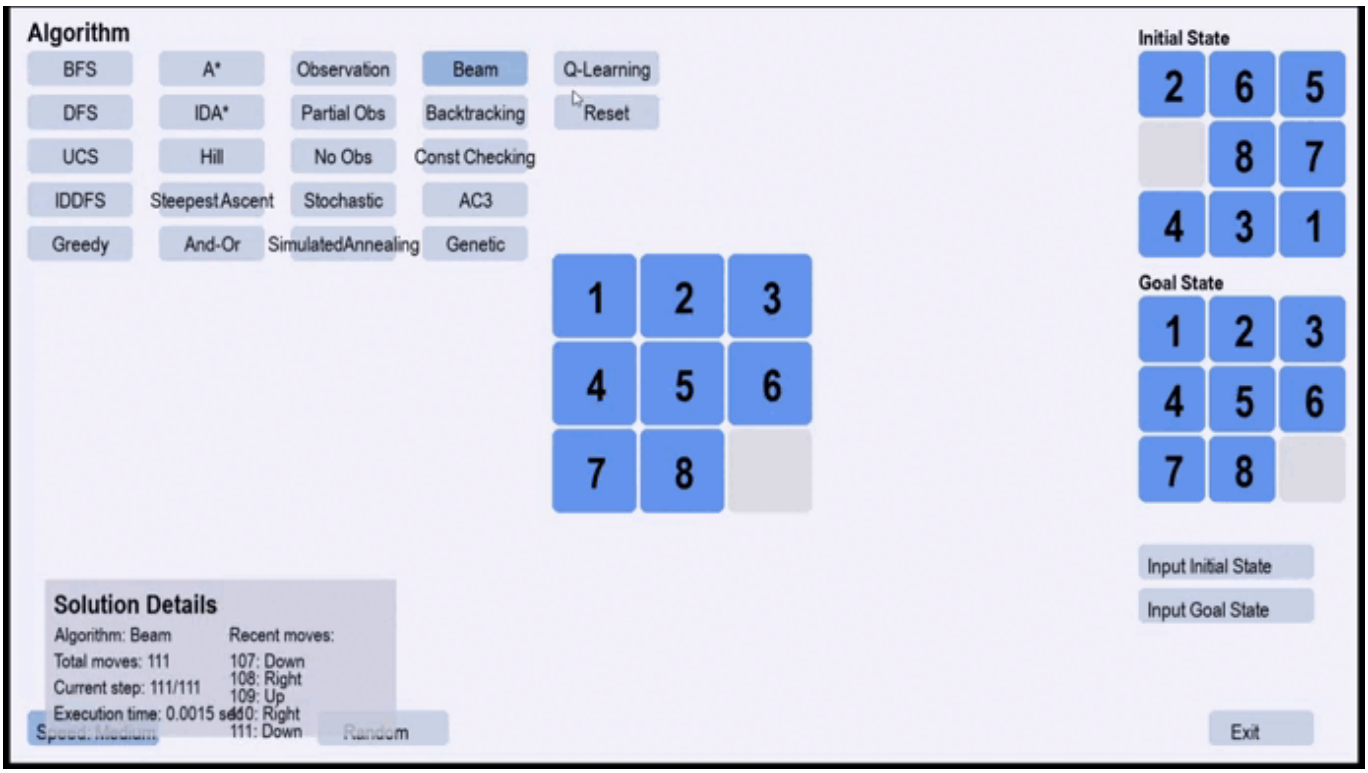
AC3 (Arc Consistency Algorithm #3)

- **Mô tả**: Lọc các giá trị không hợp lệ ra khỏi miền giá trị dựa trên tính nhất quán cung (arc consistency) giữa các biến.
- **Minh họa**:



Constraint Checking

- **Mô tả**: Kiểm tra ràng buộc đơn giản trong quá trình duyệt trạng thái.
- **Minh họa**:



So sánh hiệu suất:

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Đặc điểm

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Đặc điểm
Backtracking	50-150	Thấp	Tìm tất cả các giải pháp khả thi
AC3	150-200	Trung bình	Cải thiện hiệu suất tìm kiếm bằng cách loại bỏ giá trị không hợp lệ
Constraint Checking	100-150	Thấp	Kiểm tra tính hợp lệ của các trạng thái

Nhận xét:

- **Backtracking** đơn giản và hiệu quả cho các bài toán nhỏ, nhưng có thể chậm với không gian trạng thái lớn
- **AC3** và **Constraint Checking** cải thiện hiệu suất tìm kiếm bằng cách loại bỏ sớm các giá trị không hợp lệ

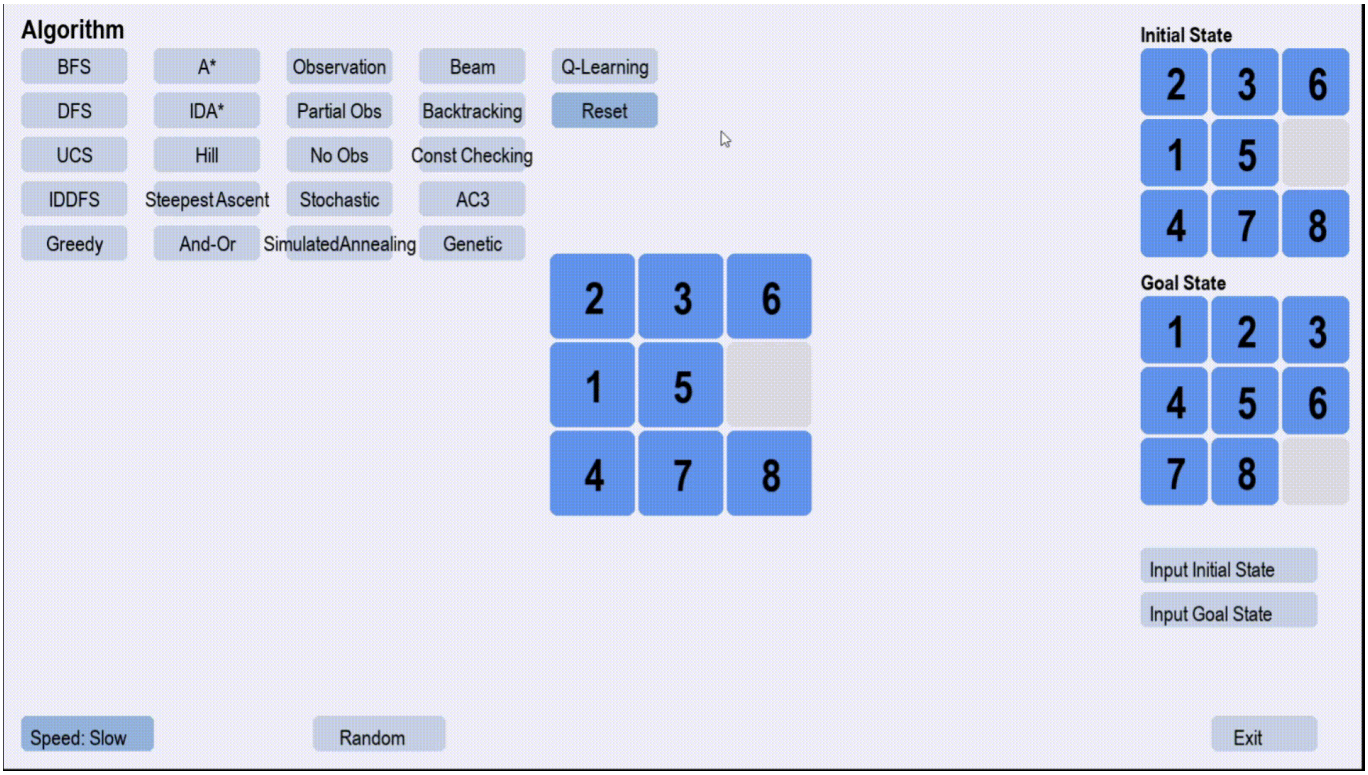
2.6. Nhóm 6: Thuật Toán Học Tăng Cường (Reinforcement Learning)

Thành phần chính của bài toán tìm kiếm:

- **Trạng thái (State)**: Đại diện cho một cấu hình cụ thể của bảng 8-Puzzle, thể hiện vị trí hiện tại của các số từ 0 đến 8, trong đó 0 là ô trống. Mỗi trạng thái là một điểm trong không gian trạng thái rộng lớn mà agent cần khám phá.
- **Hành động (Action)**: Là các phép di chuyển hợp lệ của ô trống trong bảng (lên, xuống, trái, phải), được thực hiện bởi agent để thay đổi trạng thái hiện tại sang một trạng thái mới.
- **Phần thưởng (Reward)**: Một giá trị số phản ánh mức độ tốt hoặc xấu của một hành động. Agent nhận được phần thưởng dương khi tiến gần đến trạng thái mục tiêu, và phần thưởng âm (hoặc 0) nếu thực hiện hành động không hiệu quả hoặc lặp lại..
- **Chính sách (Policy)**: Là chiến lược ra quyết định được agent học được sau nhiều lần thử nghiệm. Chính sách giúp xác định hành động tối ưu nên chọn tại mỗi trạng thái để tối đa hóa phần thưởng tích lũy theo thời gian.
- **Solution (Lời giải)**: Là chuỗi các hành động do agent lựa chọn theo chính sách đã học, giúp dẫn từ trạng thái ban đầu đến trạng thái đích mà không cần phải duyệt toàn bộ cây trạng thái như các thuật toán tìm kiếm truyền thống.

Q-Learning

- **Mô tả**: Thuật toán học tăng cường để tìm chiến lược tối ưu
- **Minh họa**:



Temporal Difference (TD) Learning

- **Mô tả**: Thuật toán học giá trị trạng thái dựa trên sự khác biệt tạm thời (temporal difference) giữa giá trị hiện tại và giá trị kế tiếp.
- **Minh họa**:

So sánh hiệu suất:

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Đặc điểm
Q-Learning	1000 - vài giây	Cao	Học dần dần chiến lược tối ưu

Thuật toán	Thời gian giải (ms)	Bộ nhớ sử dụng	Đặc điểm
TD-Learning	500 – 1500	Cao	Cập nhật nhanh, không cần mô hình môi trường

Nhận xét:

- **Q-Learning** có khả năng học và cải thiện hiệu suất theo thời gian, nhưng đòi hỏi nhiều tài nguyên
- **TD Learning** có thể học nhanh và nhẹ hơn do không cần lưu bảng Q đầy đủ, thích hợp với môi trường không xác định rõ mô hình.

3. Kết Luận

Kết quả đạt được

- Triển khai thành công 21 thuật toán tìm kiếm, bao gồm đầy đủ 6 nhóm: từ tìm kiếm không có thông tin, có thông tin, cục bộ, cho đến học tăng cường và môi trường ràng buộc phức tạp.
- Xây dựng một giao diện trực quan và dễ sử dụng bằng Pygame, giúp người dùng theo dõi trực tiếp quá trình giải quyết bài toán theo từng bước.
- Tổ chức lại toàn bộ nội dung thuật toán theo nhóm, kết hợp minh hoạ hình ảnh và bảng hiệu suất để trực quan và dễ tiếp cận.
- Phân tích chi tiết điểm mạnh và hạn chế của từng thuật toán khi áp dụng cụ thể vào trò chơi 8-puzzle.

Nhận xét tổng quát

- CNhóm thuật toán không có thông tin như BFS, DFS, UCS, và IDDFS phù hợp với bài toán kích thước nhỏ và yêu cầu giải chính xác, nhưng có thể gặp hạn chế về hiệu suất nếu không kiểm soát tốt số lượng trạng thái mở rộng trong không gian lớn.
- Tìm kiếm có thông tin (Informed Search) như A*, IDA*, và Greedy tận dụng tốt các hàm heuristic để dẫn hướng quá trình giải. Trong đó, A* cho kết quả rất chính xác và ổn định, còn Greedy tuy nhanh nhưng có thể bỏ sót lời giải tối ưu.
- Tìm kiếm cục bộ (Local Search) hoạt động hiệu quả về mặt tốc độ và bộ nhớ, nhưng thường bị mắc kẹt tại cực trị địa phương. Việc sử dụng các biến thể như Simulated Annealing hoặc Genetic Algorithm giúp cải thiện khả năng thoát khỏi các điểm nghẽn này.
- Tìm kiếm trong môi trường phức tạp (như AND-OR Search, Partial Observable, No Observation) là giải pháp phù hợp cho các bài toán không xác định hoặc không đầy đủ thông tin, nơi mà việc xác định một trạng thái cụ thể là không thể hoặc hành động có nhiều kết quả có thể xảy ra.
- Bài toán ràng buộc (CSP) như các giải pháp Backtracking, AC3 và Constraint Checking cho phép rút gọn không gian tìm kiếm thông qua kiểm tra tính hợp lệ sớm, từ đó tăng khả năng tìm được lời giải hợp lệ nhanh hơn.
- Học tăng cường (Reinforcement Learning) tiêu biểu là Q-Learning và TD-Learning, cho thấy khả năng học chính sách giải tối ưu thông qua trải nghiệm và phản hồi từ môi trường thay vì duyệt tuần tự. Đây là hướng tiếp cận mới mẻ và tiềm năng cho các hệ thống tự động thông minh.

Hướng phát triển

-Tối ưu hóa hiệu năng: Nâng cao tốc độ xử lý và tiết kiệm bộ nhớ bằng cách cải tiến thuật toán tìm kiếm, giảm thiểu số lần mở rộng trạng thái trùng lặp và áp dụng các cấu trúc dữ liệu hiệu quả hơn như bảng băm hoặc priority queue cải tiến.

-Bổ sung đa dạng hàm heuristic: Cho phép người dùng tùy chọn giữa nhiều loại hàm heuristic (Misplaced Tiles, Manhattan, Linear Conflict, v.v.), đồng thời đánh giá ảnh hưởng của mỗi loại đến hiệu suất và chất lượng lời giải.

-Phát triển phiên bản mở rộng: Mở rộng ứng dụng để hỗ trợ các biến thể khó hơn như 15-Puzzle, 24-Puzzle, từ đó kiểm tra khả năng mở rộng (scalability) của thuật toán và giao diện.

-Tích hợp trực quan hóa giải thuật: Cung cấp chế độ trình bày từng bước (step-by-step), giải thích logic tại mỗi bước đi (ví dụ: lý do chọn node này, giá trị h/g/f tương ứng...), hỗ trợ việc giảng dạy và học thuật.

-Hỗ trợ người dùng nâng cao: Cho phép người dùng nhập code tùy chỉnh để tự thử nghiệm thuật toán mới, hoặc cho phép nạp thuật toán bên ngoài vào hệ thống để đánh giá và so sánh.

-Lưu và tải trạng thái: Thêm tính năng lưu lại tiến trình giải, hoặc xuất log các bước để người dùng có thể tải về, chia sẻ hoặc kiểm chứng lại quá trình tìm kiếm.

-Tích hợp benchmark và thống kê tự động: Ghi nhận thời gian, số bước, số node mở rộng và so sánh các thuật toán trên cùng một tập hợp trạng thái đầu vào để giúp người dùng lựa chọn giải pháp tối ưu hơn.

-Hỗ trợ đa nền tảng: Phát triển phiên bản web bằng framework như React/PyScript hoặc phiên bản mobile để tiện sử dụng và chia sẻ rộng rãi.

Cài Đặt và Chạy Game

Yêu Cầu

- Python 3.x
- Thư viện **Pygame** (Cài đặt qua `pip`):

```
pip install pygame
```

Cách Tải và Cài Đặt

1. Clone dự án về máy của bạn:

```
https://github.com/Nguyneee/DoAn8Puzzle0702.git
```

2. Chạy ứng dụng:

```
python -m DoAn8Puzzle.DoAnCaNhan
```

Hướng Dẫn Chơi 🎮

1. **Chỉnh Sửa Trạng Thái Ban Đầu**:
   - Nhấp vào các ô hoặc cuộn con lăn chuột để thay đổi giá trị. Ô trống sẽ là số `0`.
   - Bạn có thể nhấp chuột phải để thay đổi giá trị của ô trống từ 8 đến 0.
2. **Chọn Thuật Toán**:
   - Chọn thuật toán từ danh sách để giải bài toán (ví dụ: BFS, A*, hoặc Simulated Annealing).
   - Sau khi chọn thuật toán, ứng dụng sẽ bắt đầu giải quyết và hiển thị số bước đi và thanh tiến trình.
3. **Reset** 🔄:
   - Bạn có thể nhấn "Reset" để quay lại trạng thái ban đầu của puzzle.
4. **Hiển Thị Tiến Trình** 📊:
   - Số bước đi sẽ được cập nhật trong giao diện khi thuật toán đang chạy.
   - Thanh tiến trình sẽ cho bạn thấy tiến độ giải bài toán.

🧩 Cấu trúc dự án

```
DOAN8PUZZLE0702/
├── DoAn8Puzzle/
│   ├── __init__.py
│   ├── DoAnCaNhan.py    # Điểm vào chính của ứng dụng, đồ họa
│   ├── algorithms.py    # Các thuật toán giải 8-puzzle
│   └── utils.py         # Các hàm tiện ích
├── Logo_8puzzle/              # Hình ảnh logo
└── README.md            # Tài liệu dự án
```

🤝 Đóng góp

Mọi đóng góp đều được hoan nghênh! Nếu bạn muốn đóng góp, vui lòng:

1. Fork repository
2. Tạo branch mới (`git checkout -b feature/amazing-feature`)
3. Commit thay đổi (`git commit -m 'Add some amazing feature'`)
4. Push lên branch (`git push origin feature/amazing-feature`)
5. Mở Pull Request

👤 Tác giả

- **Họ tên**: Trịnh Nguyễn Hoàng Nguyên
- **MSSV**: 23110272
- **Môn học**: Trí Tuệ Nhân Tạo
- **Trường**: Đại học Sư phạm Kỹ thuật TP.HCM (HCMUTE)
- **GVHD**: TS.Phan Thị Huyền Trang

Liên hệ

Hoàng Nguyên - HoangNguyen

Project Link: https://github.com/Nguyneee/DoAn8Puzzle0702

Về đầu trang