

TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH
KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO CUỐI KỲ
Môn học: TRÍ TUỆ NHÂN TẠO

Tên đề tài:
THIẾT KẾ VÀ CÀI ĐẶT TRÒ CHƠI N PUZZLE
SỬ DỤNG CÁC THUẬT TOÁN AI

Giảng viên hướng dẫn: PGS.TS. Hoàng Văn Dũng

Danh sách sinh viên thực hiện:

MSSV	Họ và tên	Tỉ lệ hoàn thành (%)
21110335	Nguyễn Trần Văn Trung	100%
21110332	Kiến Đức Trọng	100%

TP. Hồ Chí Minh, tháng 11 năm 2023.

MỤC LỤC

I. MỞ ĐẦU	1
1. Lý do chọn đề tài	1
2. Mục đích nghiên cứu	1
3. Đối tượng nghiên cứu	1
4. Phạm vi nghiên cứu	1
5. Tóm tắt đề tài	2
II. CƠ SỞ LÝ THUYẾT VÀ DÙNG ĐỂ THỰC HIỆN ĐỒ ÁN	3
1. Công nghệ sử dụng	3
2. Các thuật toán tìm kiếm	3
2.1 Tìm kiếm toàn cục	3
2.1.1 Không có thông tin	4
2.1.1.1 BFS (Breadth-First Search)	4
2.1.1.2 DFS (Depth-First Search)	4
2.1.1.3 DLS (Depth-Limited Search)	4
2.1.1.4 Iterative Deepening Depth-First Search	5
2.1.1.5 UCS (Uniform Cost Search)	5
2.1.1.6 BS (Bidirectional Search)	5
2.1.2 Có thông tin	6
2.1.2.1 Greedy Search	6
2.1.2.2 A* Search	6
2.1.2.3 IDA* Search	7
2.2 Tìm kiếm cục bộ	7
2.2.1 Hill Climbing	7
2.2.2 Simulated Annealing	7
2.2.3 Beam Search	7
III. PHÂN TÍCH, THIẾT KẾ GIẢI PHÁP	8
1. Các hàm cài đặt phụ trợ	8
1.1 Hàm tạo ra các bước di chuyển của trạng thái	8
1.2 Các hàm ước lượng chi phí Heuristic	8
1.2.1 Khoảng cách Manhattan	8
1.2.2 Khoảng cách Hamming	9
1.2.3 Xung đột tuyến tính	9
1.2.4 Độ lệch giữa các ô	10
2. Các giải thuật tìm kiếm	10

2.1 Toàn cục.....	10
2.1.1 Không có thông tin	10
2.1.1.1 BFS (Breadth-First Search)	10
2.1.1.2 DFS (Depth-First Search)	11
2.1.1.3 DLS (Depth-Limited Search).....	12
2.1.1.4 IDDFS (Iterative Deepening Depth-First Search)	13
2.1.1.5 UCS (Uniform Cost Search)	13
2.1.1.6 BS (Bidirectional Search).....	14
2.1.1 Có thông tin	15
2.1.1.1 Greedy Search.....	15
2.1.1.2 A* Search.....	16
2.1.1.3 IDA* Search	17
2.2 Cục bộ	18
2.2.1 Hill Climbing	18
2.2.2 Simulated Annealing.....	19
2.2.3 Beam Search	20
IV. THỰC NGHIỆM, ĐÁNH GIÁ, PHÂN TÍCH KẾT QUẢ.....	22
1. Giao diện	22
1.1 Tùy chỉnh kích thước của trò chơi	22
1.2 Hướng dẫn sử dụng	22
1.2.1 Thông số biểu diễn	23
1.2.2 Các nút điều khiển	23
1.2.3 Các ô số.....	23
1.2.3 Lịch sử các bản ghi đã giải	24
2. Đánh giá, phân tích kết quả.....	24
2.1 Ghi chú.....	24
2.1.1 Lựa chọn ước lượng độ dài Manhattan	24
2.1.2 Hill Climbing và Simulated Annealing	24
2.1.3 Đánh giá theo kích thước của bài toán	25
2.2 So sánh, đánh giá dựa trên kích thước 3x3	25
V. KẾT LUẬN.....	28

I. MỞ ĐẦU

1. Lý do chọn đề tài

N Puzzle là một trò chơi giải đố cổ điển, trong đó có các ô số được sắp xếp ngẫu nhiên theo ma trận $N = m \text{ (hàng)} * n \text{ (cột)}$, người chơi sẽ di chuyển các ô số để sắp xếp chúng theo thứ tự từ 1 đến N, đến khi có được trạng thái cuối cùng sẽ chiến thắng.

Ví dụ: Ma trận 3x3

Initial state

3	7	6
4		2
5	8	1

Goal State

	1	2
3	4	5
6	7	8

Trong quá trình học tập và thực nghiệm các các giải thuật tìm kiếm lời giải, chúng tôi nhận thấy tiềm năng của trò chơi N Puzzle là đề tài phù hợp để áp dụng minh họa những thuật toán tìm kiếm lời giải thông qua duyệt các trạng thái. Vì vậy mà chúng tôi quyết định lựa chọn chủ đề này cho đề án cuối kỳ của bộ môn Trí tuệ nhân tạo.

2. Mục đích nghiên cứu

Thiết kế trò chơi N puzzle trong dự án lần này ứng dụng được nhiều thuật toán AI để giải quyết bài toán N puzzle. Tối ưu hóa giải thuật để giải quyết bài toán N Puzzle hiệu quả. Đánh giá được hiệu suất của các thuật toán dựa trên các tiêu chí: thời gian thực thi, số bước giải, và số trạng thái đã duyệt qua.

3. Đối tượng nghiên cứu

Đối tượng chính của đề tài này là trò chơi N Puzzle, thông qua việc cài đặt và minh họa các thuật toán trí tuệ nhân tạo. Từ đó, chúng tôi có thể đưa ra những nhận định và đánh giá độ hiệu quả của từng thuật toán.

4. Phạm vi nghiên cứu

Thuật toán AI: Nghiên cứu và triển khai các thuật toán AI như thuật toán tìm kiếm gồm 12 thuật toán:

- **Tìm kiếm toàn cục:**
 - **Không có thông tin:**
 - BFS (Breadth-First Search)
 - DFS (Depth-First Search)
 - DLS (Depth-Limited Search)
 - IDDFS (Iterative Deepening Depth-First Search)
 - UCS (Uniform Cost Search)
 - BS (Bidirectional Search)

- **Có thông tin:**
 - Greedy Search
 - A* Search
 - IDA* Search
- **Tìm kiếm cục bộ:**
 - Hill Climbing Search
 - Simulated Annealing Search
 - Beam Search

Các hàm ước lượng đường đi - Heuristics: Đối với các giải thuật tìm kiếm có thông tin và tìm kiếm cục bộ, chúng tôi có cài đặt một số hàm ước lượng heuristics để cung cấp thông tin.

- Khoảng cách Manhattan
- Khoảng cách Hamming
- Xung đột tuyến tính: Linear conflict
- Độ lệch của các ô số: Misplaced tiles

Kích thước trò chơi N Puzzle: Số lượng các ô số là $N = m * n$. Với m, n lần lượt là số hàng và số cột của hình vuông Puzzle.

5. Tóm tắt đề tài

Đề án cuối kỳ lần này sẽ bao gồm hai phần:

- Giới thiệu về cơ sở lý thuyết của các thuật toán tìm kiếm trí tuệ nhân tạo, sơ tầm và sơ lược nội dung chính.
- Phân tích, thiết kế và cài đặt trò chơi N Puzzle sử dụng các thuật toán AI, từ đó đưa ra những nhận định đánh giá độ hiệu quả của từng thuật toán khác nhau.

II. CƠ SỞ LÝ THUYẾT VÀ DÙNG ĐỂ THỰC HIỆN ĐỒ ÁN

1. Công nghệ sử dụng

- **Ngôn ngữ lập trình:** Python (v3.11)
- **Cấu trúc mã nguồn:** Functional
- **IDE:** Visual Studio Code
- **Thư viện:**
 - Đồ họa:
 - **tkinter:** đồ họa giao diện người dùng, xây dựng GUI
 - **ttk:** cung cấp các widget mở rộng cho tkinter (sử dụng Combobox)
 - **PIL:** xử lý đồ họa và hình ảnh
 - **filedialog:** mở cửa sổ chọn file đồ họa
 - **messagebox:** hiển thị hộp thoại thông báo
 - Hệ thống:
 - **threading:** đa luồng (chạy thuật toán, cập nhật giao diện trên nhiều luồng)
 - **sys:** cung cấp quyền truy cập vào các biến/hàm liên quan đến interpreter Python
 - **os:** tương tác với hệ điều hành (operating system)
 - Cấu trúc dữ liệu:
 - **queue:** xử lý hàng đợi
 - **deque:** cấu trúc dữ liệu deque (double-ended queue)
 - **heapq:** cấu trúc dữ liệu heap
 - Khác:
 - **random:** tạo ngẫu nhiên (tìm các hướng đi ngẫu nhiên, xáo trộn các ô số)
 - **time:** xử lý thời gian (cài đặt độ trễ cho cập nhật giao diện, tính toán thời gian xử lý của các thuật toán)

2. Các thuật toán tìm kiếm

2.1 Tìm kiếm toàn cục

Những thuật toán tìm kiếm toàn cục sẽ bao gồm những thuật toán có xu hướng duyệt trạng thái tìm kiếm dựa trên toàn bộ trạng thái của thể đạt được từ trạng thái ban đầu. Trong tìm kiếm cục bộ sẽ chia ra làm hai nhóm đó là tìm kiếm có thông tin và tìm kiếm không có thông tin.

2.1.1 Không có thông tin

Những thuật toán tìm kiếm không có thông tin thường được gọi là tìm kiếm mù, chúng không được cung cấp thông tin về chi phí cũng như ước lượng đường đi nên sẽ có xu hướng duyệt nhiều trạng thái hơn các thuật toán đã được tính toán đường đi như những thuật toán tìm kiếm có thông tin. Vì các trạng thái duyệt này được mô hình theo kỹ thuật vét cạn nên có thể sẽ luôn tìm ra kết quả với điều kiện đáp án phải có trong phạm vi được duyệt tìm kiếm.

2.1.1.1 BFS (Breadth-First Search)

“Tìm kiếm theo chiều rộng (BFS) là một thuật toán để duyệt đồ thị hoặc cây. BFS sử dụng cấu trúc dữ liệu hàng đợi (queue) để tìm đường đi ngắn nhất trong biểu đồ.”¹

Thuật toán sử dụng một cấu trúc dữ liệu hàng đợi để lưu trữ thông tin trung gian thu được trong quá trình tìm kiếm:

Chèn đỉnh gốc vào hàng đợi (đang hướng tới)
Lấy ra đỉnh đầu tiên trong hàng đợi và quan sát nó
Nếu đỉnh này chính là đỉnh đích, dừng quá trình tìm kiếm và trả về kết quả.
Nếu không phải thì chèn tất cả các đỉnh kề với đỉnh vừa thăm nhưng chưa được quan sát trước đó vào hàng đợi.
Nếu hàng đợi là rỗng, thì tất cả các đỉnh có thể đến được đều đã được quan sát – dừng việc tìm kiếm và trả về "không thấy".
Nếu hàng đợi không rỗng thì quay về bước 2.

2.1.1.2 DFS (Depth-First Search)

“Tìm kiếm theo chiều sâu (DFS) là một thuật toán để duyệt qua hoặc tìm kiếm cấu trúc dữ liệu dạng cây hoặc đồ thị. Thuật toán bắt đầu tại nút gốc (chọn một số nút tùy ý làm nút gốc trong trường hợp đồ thị) và kiểm tra từng nhánh càng xa càng tốt trước khi quay lui. Để thực hiện duyệt theo DFS, chúng ta cần sử dụng cấu trúc dữ liệu ngăn xếp có kích thước tối đa bằng tổng số đỉnh trong biểu đồ.”²

Các bước thực hiện tìm kiếm của DFS tương tự với kỹ thuật của BFS nhưng thay vì dùng hàng đợi như BFS thì DFS sẽ sử dụng ngăn xếp.

2.1.1.3 DLS (Depth-Limited Search)

“Là một thuật toán tìm kiếm không xác định tương tự như Tìm kiếm theo chiều sâu (DFS). Nó có thể được coi là tương đương với DFS với giới hạn độ sâu được xác định trước 'l'. Các nút ở độ sâu l được coi là các nút không có nút kế thừa. Tìm kiếm giới hạn độ sâu có thể được coi là một giải pháp cho vấn đề đường dẫn vô hạn của DFS;

¹ <https://viblo.asia/p/data-structure-algorithm-graph-algorithms-breadth-first-search-bfs-gwd43kMM4X9>

² <https://viblo.asia/p/data-structure-algorithm-graph-algorithms-depth-first-search-dfs-qPoL7zyXJvk>

trong thuật toán tìm kiếm giới hạn độ sâu, DFS được chạy ở độ sâu hữu hạn 'l', trong đó 'l' là giới hạn độ sâu. Trước khi chuyển sang đường dẫn tiếp theo, Tìm kiếm theo chiều sâu bắt đầu từ nút gốc và đi theo từng nhánh đến nút sâu nhất. Vấn đề với DFS là điều này có thể dẫn đến vòng lặp vô hạn. Bằng cách kết hợp một giới hạn cụ thể được gọi là giới hạn độ sâu, Thuật toán tìm kiếm giới hạn độ sâu sẽ loại bỏ vấn đề về đường dẫn vô hạn của thuật toán DFS.”³

Đây là một loại giải thuật tìm kiếm dựa trên DFS nhưng có sự cải tiến hơn vì nó sẽ có giới hạn độ sâu tùy chỉnh. Trong giai đoạn giải thuật toán nếu độ sâu đạt một ngưỡng nhất định sẽ quay trở lại duyệt hết tất cả những trạng thái đã lưu vào hàng đợi trước đó, điều này cho phép giải thuật có độ hiệu quả cao hơn DFS, tuy nhiên vẫn sẽ không thể lời giải nếu ở ngưỡng độ sâu được thiết lập không tồn tại lời giải.

2.1.1.4 Iterative Deepening Depth-First Search

“Iterative Deepening Depth-First Search (IDDFS) kết hợp tính hiệu quả về không gian của tìm kiếm theo chiều sâu và tìm kiếm nhanh của tìm kiếm theo chiều rộng (đối với các nút gần gốc hơn).”⁴

Ở trường hợp này, thuật toán IDDFS là một trường hợp cải tiến của DLS, nó sẽ duyệt tìm kiếm từ độ sâu giới hạn nhỏ nhất là 1 và tăng dần lên. Nếu ở độ sâu hiện tại không có lời giải, nó tiếp tục tăng độ sâu giới hạn cho đến khi tìm kiếm được lời giải cuối cùng sẽ kết thúc thuật toán.

2.1.1.5 UCS (Uniform Cost Search)

“Là một thuật toán tìm kiếm sử dụng chi phí tích lũy thấp nhất để tìm đường đi từ nguồn đến đích. Các nút được mở rộng, bắt đầu từ gốc, theo chi phí tích lũy tối thiểu. Việc tìm kiếm chi phí thống nhất sau đó được triển khai bằng Hàng đợi ưu tiên.”⁵

UCS là giải thuật tìm kiếm theo chiều rộng nhưng có tính giá trị chi phí của từng bước di chuyển của trạng thái. Và đối với phạm vi của bài toán N Puzzle này thì mỗi bước duyệt trạng thái được tính là một đơn vị nên về cấu trúc giải thuật sẽ tương tự BFS vì nó sẽ duyệt theo chiều rộng. Vì vậy nên chúng tôi xếp loại thuật toán này vào “Tìm kiếm không có thông tin”.

2.1.1.6 BS (Bidirectional Search)

“Trong tìm kiếm biểu đồ thông thường bằng BFS/DFS, ta bắt đầu tìm kiếm theo một hướng thường từ đỉnh nguồn tới đỉnh đích, nhưng điều gì sẽ xảy ra nếu chúng ta bắt đầu tìm kiếm từ cả hai hướng cùng một lúc. Tìm kiếm hai chiều là một thuật toán

³ <https://iq.opengenus.org/depth-limited-search/>

⁴ <https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/>

⁵ <https://www.educative.io/answers/what-is-uniform-cost-search>

tìm kiếm đồ thị tìm đường đi nhỏ nhất từ đỉnh nguồn đến đỉnh đích. Nó chạy hai tìm kiếm đồng thời từ hai hướng”.⁶

Giải thuật BS này có thể áp dụng cả kỹ thuật hàng đợi hoặc ngăn xếp, tương tự như BFS và DFS, nhưng mỗi lần duyệt sẽ duyệt đường đi từ cả hai trạng thái (trạng thái đầu và trạng thái đích). Nếu hai lần duyệt đó gặp nhau tại một trạng thái, sẽ cộng gộp lại đường đi của cả 2 trạng thái và trả về lời giải.

2.1.2 Có thông tin

Đối với những thuật toán tìm kiếm có thông tin, sự chuyển giao giữa hai trạng thái cụ thể nào đó từ trạng thái đầu đến trạng thái đích sẽ thông qua một hàm ước lượng chi phí tổng quát $f = g + h$. Trong đó, giá trị g đại diện cho chi phí mỗi lần duyệt chuyển trạng thái, còn h là giá trị hàm ước lượng chi phí sẽ phải đến trạng thái tiếp theo. Sau mỗi lần duyệt trạng thái sẽ kiểm tra và so sánh giá trị f này giữa hai trạng thái chuyển giao, nếu chi phí ước lượng thấp hơn sẽ ưu tiên chuyển qua trạng thái đó hơn. Vì vậy, các giải thuật tìm kiếm có thông tin thường có xu hướng duyệt ít trạng thái hơn những thuật toán tìm kiếm mù.

2.1.2.1 Greedy Search

“Greedy Search là một mô hình thuật toán xây dựng từng giải pháp một, luôn chọn bước tiếp theo mang lại lợi ích rõ ràng và ngay lập tức nhất. Vì vậy, các bài toán chọn tối ưu cục bộ cũng dẫn đến giải pháp toàn cục là phù hợp nhất cho Greedy.”⁷

Thuật toán "greedy search" chọn hướng đi dựa trên việc ước tính chi phí gần nhất đến mục tiêu tốt nhất tại thời điểm đó, mà không cần xem xét các hành động tiếp theo có thể dẫn đến giải pháp tối ưu nhất.

2.1.2.2 A* Search

Là thuật toán kết hợp giữa UCS và Greedy. Tức nó vừa nhìn lại để tính toán chi phí như UCS, nhưng cũng ước tính được bước đi tốt nhất ở tương lai. Là một thuật toán gần như là lựa chọn tốt nhất.

A* kết hợp giữa chi phí thực tế đã di chuyển từ điểm xuất phát đến điểm hiện tại (g), và ước tính chi phí còn lại đến điểm đích (h), thông qua việc sử dụng một hàm heuristic (hàm đánh giá) để ước tính chi phí còn lại. Thuật toán A* sử dụng một giá trị $f = g + h$ để đánh giá mỗi nút trong đồ thị, và sau đó lựa chọn nút có giá trị f nhỏ nhất để tiếp tục tìm kiếm.

⁶ <https://www.geeksforgeeks.org/bidirectional-search/>

⁷ <https://www.geeksforgeeks.org/greedy-algorithms/>

2.1.2.3 IDA* Search

Là thuật toán kết hợp A* với IDDFS. IDA* lặp lại quá trình này với các giới hạn sâu tăng dần cho đến khi tìm được giải pháp và IDA* sử dụng chiến lược tìm kiếm thông minh A* để đánh giá các trạng thái và quyết định hướng đi tiếp theo.

Thuật toán IDA* kết hợp giữa kỹ thuật duyệt đệ quy và việc sử dụng một hàm heuristic để ước tính chi phí còn lại đến đích. Thay vì duyệt qua toàn bộ cây tìm kiếm, IDA* duyệt từng đường đi theo độ sâu tăng dần, mỗi lần tăng dần độ sâu này thấp hơn một ngưỡng hàm heuristic cụ thể (được gọi là "threshold") cho đến khi tìm ra đường đi mục tiêu.

2.2 Tìm kiếm cục bộ

2.2.1 Hill Climbing

Hill Climbing là một thuật toán tìm kiếm cục bộ (Local Search) được sử dụng trong trí tuệ nhân tạo để giải quyết bài toán tối ưu hóa. Mục tiêu của thuật toán là di chuyển qua các trạng thái của không gian tìm kiếm và tìm ra trạng thái mà có giá trị (điểm số) cao nhất trong số các trạng thái lân cận. Tức là tìm bước tiếp theo có chi phí nhỏ nhất.

2.2.2 Simulated Annealing

Được lấy cảm hứng từ quy trình nung nóng và làm nguội kim loại trong công nghiệp. Chọn ngẫu nhiên các trạng thái lân cận để di chuyển, sẽ chuyển sang trạng thái đó nếu nó tốt hơn trạng thái hiện tại, nếu không sẽ chỉ chuyển sang trạng thái với một xác suất nào đó theo công thức $e = \Delta/T$. Với Δ = “chi phí của trạng thái sẽ đi” - “chi phí của trạng thái hiện tại”. T là nhiệt độ được giảm sau mỗi lần duyệt các trạng thái lân cận, T lớn thì khả năng chọn trạng thái “tồi” càng cao.

2.2.3 Beam Search

Beam search chọn ra k ứng viên tốt nhất cho mỗi bước tính toán trong những bước đi có thể đi. Và xử lý chạy dựa trên thuật toán BFS.

Ý tưởng cơ bản của beam search là giữ một số lượng hữu hạn các lựa chọn tốt nhất (được gọi là "beam width" hoặc "beam size") thay vì duyệt qua tất cả các lựa chọn có thể. Beam search giữ lại các lựa chọn hàng đầu dựa trên một hàm đánh giá hay một tiêu chí chọn lọc (ví dụ: xác suất cao nhất, điểm số cao nhất, hoặc một hàm mục tiêu).

III. PHÂN TÍCH, THIẾT KẾ GIẢI PHÁP

1. Các hàm cài đặt phụ trợ

1.1 Hàm tạo ra các bước di chuyển của trạng thái

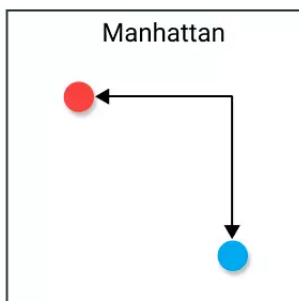
Hàm `possible_moves()` này ứng dụng trong hầu hết các thuật toán, trả về một mảng các bước đi có thể di chuyển, và xáo trộn ngẫu nhiên những bước đi đó, số lượng bước đi có thể tùy góc di chuyển mà thay đổi từ 2 \rightarrow 4 (lên, xuống, trái phải).

```
def possible_moves(current_node):
    moves = []
    empty_index = current_node.index(0)
    row, col = empty_index // COL, empty_index % COL
    for dr, dc in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < ROW and 0 <= new_col < COL:
            new_empty_index = new_row * COL + new_col
            new_node = list(current_node)
            new_node[empty_index], new_node[new_empty_index] = (
                new_node[new_empty_index],
                new_node[empty_index],)
            moves.append((new_node, new_node[empty_index]))
    random.shuffle(moves)
    return moves
```

1.2 Các hàm ước lượng chi phí Heuristic

Các hàm ước lượng chi phí này dùng để cài đặt cho các thuật toán tìm kiếm có thông tin và các tìm kiếm cục bộ.

1.2.1 Khoảng cách Manhattan



Là khoảng cách giữa hai điểm trên mặt phẳng theo dạng "đường đi chấm dứt", tức là chỉ di chuyển theo các hướng ngang và dọc, không cho phép di chuyển theo đường chéo.

```

def manhattan_distance(puzzle):
    distance = 0
    for i in range(ROW):
        for j in range(COL):
            if puzzle[i * COL + j] != 0:
                correct_row = (puzzle[i * COL + j]) // COL
                correct_col = (puzzle[i * COL + j]) % COL
                distance += abs(i - correct_row) + abs(j - correct_col)
    return distance

```

1.2.2 Khoảng cách Hamming

Là số lượng vị trí mà các phần tử tại đó của hai trạng thái khác nhau. Hàm này lặp qua lưới của bài toán và đối với mỗi ô khác không và không ở vị trí đúng (không ở chỉ số đúng theo trạng thái mục tiêu), nó tăng giá trị khoảng cách lên 1 đơn vị.

```

def hamming_distance(puzzle):
    distance = 0
    for i in range(ROW):
        for j in range(COL):
            if puzzle[i * COL + j] != 0 and puzzle[i * COL + j] != i * COL + j + 1:
                distance += 1
    return distance

```

1.2.3 Xung đột tuyến tính

Heuristic này mở rộng trên heuristic khoảng cách Manhattan bằng cách xem xét các xung đột bổ sung xảy ra khi hai ô trong cùng một hàng hoặc cột đều ở trong hàng hoặc cột đúng của chúng nhưng lại hoán đổi vị trí. Xung đột này xuất hiện vì việc di chuyển một ô vào vị trí đúng của nó có thể làm cản trở ô khác đến vị trí đúng của nó. Heuristic này tính toán các xung đột này để đưa ra ước tính chính xác hơn về khoảng cách giữa trạng thái hiện tại của bài toán và trạng thái mục tiêu.

```

def linear_conflict(puzzle):
    conflicts = 0
    for i in range(ROW):
        for j in range(COL):
            if puzzle[i * COL + j] != 0:
                correct_row = (puzzle[i * COL + j] - 1) // COL
                correct_col = (puzzle[i * COL + j] - 1) % COL
                if i == correct_row and j != correct_col:
                    conflicts += sum(1 for k in range(j + 1, COL))

```

```

        if puzzle[i * COL + k] != 0
        and (puzzle[i * COL + k] - 1) // COL == i
        and (puzzle[i * COL + k] - 1) % COL < correct_col)
    elif j == correct_col and i != correct_row:
        conflicts += sum(1 for k in range(i + 1, ROW)
            if puzzle[k * COL + j] != 0
            and (puzzle[k * COL + j] - 1) % COL == j
            and (puzzle[k * COL + j] - 1) // COL < correct_row)
    return conflicts * 2

```

1.2.4 Độ lệch giữa các ô

Hàm này trả về số lượng ô vuông không ở đúng vị trí so với trạng thái đích.

```

def misplaced_tiles(puzzle):
    count = 0
    for i in range(0, len(puzzle)):
        if puzzle[i] != i:
            count += 1
    return count

```

2. Các giải thuật tìm kiếm

2.1 Toàn cục

2.1.1 Không có thông tin

2.1.1.1 BFS (Breadth-First Search)

```

def bfs_solve(puzzle):
    global total_nodes
    total_nodes = 0
    visited = set()
    start_node = tuple(puzzle)
    queue = deque([(start_node, [])])
    while queue and not stop:
        current_node, path = queue.popleft()
        for item in possible_moves(current_node):
            node, pos_move = item
            if tuple(node) not in visited:
                visited.add(tuple(node))
                total_nodes += 1

```

```

new_path = path + [pos_move]
if is_solved(list(node)):
    return new_path
queue.append((node, new_path))
return None

```

Ưu điểm:

- BFS đảm bảo tìm kiếm tất cả các giải pháp có thể và tìm kiếm ra giải pháp . Điều này làm cho nó rất hữu ích khi bạn cần một giải pháp chính xác cho bài toán.
- BFS hoạt động dựa trên nguyên tắc "first come, first served." Nó tạo ra một cây tìm kiếm trạng thái, và mỗi trạng thái được thăm theo thứ tự theo chiều rộng trước.

=> Dễ hiểu và triển khai: Thuật toán BFS tương đối dễ hiểu và triển khai. Chỉ cần một hàng đợi (queue) để lưu trữ các trạng thái chờ xem xét và một bộ dữ liệu để lưu trạng thái đã thăm qua.

Nhược điểm:

BFS đòi hỏi một lượng lớn bộ nhớ, đặc biệt đối với các bài toán N-Puzzle lớn (mxn!). Điều này có thể làm cho nó không hiệu quả với các bài toán có không gian trạng thái lớn hoặc không gian trạng thái không được hạn chế.

2.1.1.2 DFS (Depth-First Search)

```

def dfs_solve(puzzle):
    global total_nodes, stop
    total_nodes = 0
    start_node = tuple(puzzle)
    stack = [(start_node, [])]
    visited = set()
    while stack and not stop:
        current_node, path = stack.pop()
        for item in possible_moves(current_node):
            node, pos_move = item
            if tuple(node) not in visited:
                total_nodes += 1
                visited.add(tuple(node))
                new_path = path + [pos_move]
                if is_solved(list(node)):
                    return new_path
                stack.append((node, new_path))
    return None

```

Ưu điểm:

Dễ triển khai: DFS dễ triển khai bằng cách sử dụng một ngăn xếp đơn giản để lưu trữ trạng thái.

Nhược điểm:

Không đảm bảo tìm kiếm giải pháp tối ưu: DFS không đảm bảo tìm kiếm giải pháp tối ưu. Điều này có nghĩa là nó có thể tìm thấy một giải pháp nhanh chóng, nhưng giải pháp đó có thể không phải là tối ưu, tức là không có số bước ít nhất để đạt được trạng thái mục tiêu.

Dễ rơi vào vòng lặp vô hạn: Nếu không kiểm soát cẩn thận, DFS có thể rơi vào vòng lặp vô hạn khi tìm kiếm trạng thái. Điều này có thể được khắc phục bằng cách theo dõi các trạng thái đã thăm qua và không quay lại chúng.

2.1.1.3 DLS (Depth-Limited Search)

```
def dls_solve(puzzle, depth_limit):
    global total_nodes
    start_node = tuple(puzzle)
    stack = [(start_node, [], 0)]
    visited = set()
    while stack and not stop:
        current_node, path, depth = stack.pop()
        if is_solved(list(current_node)):
            return path
        if depth == depth_limit:
            continue
        for item in possible_moves(current_node):
            node, pos_move = item
            if tuple(node) not in visited:
                total_nodes += 1
                visited.add(tuple(node))
                new_path = path + [pos_move]
                stack.append((node, new_path, depth + 1))
    return None
```

Ưu điểm:

Đảm bảo tìm kiếm toàn bộ không gian ở độ sâu hạn chế: DLS đảm bảo rằng tìm kiếm sẽ không di chuyển quá xa, chỉ tìm kiếm trạng thái trong một khoảng độ sâu nhất định. Điều này giúp tránh việc tốn thời gian và tài nguyên vào việc tìm kiếm toàn bộ không gian trạng thái, như trong DFS.

Nhược điểm:

Khả năng không tìm kiếm giải pháp tối ưu: DLS có thể không tìm thấy giải pháp tối ưu nếu giải pháp nằm ở một độ sâu vượt quá giới hạn độ sâu đã đặt.

Cần xác định giới hạn độ sâu hợp lý: Việc xác định giới hạn độ sâu phải là một thách thức và phụ thuộc vào kiểu bài toán và không gian trạng thái. Nếu giới hạn độ sâu được đặt quá thấp, bạn có thể bỏ lỡ giải pháp; nếu quá cao, thuật toán có thể tốn nhiều thời gian và không gian bộ nhớ.

2.1.1.4 IDDFS (Iterative Deepening Depth-First Search)

```
def iddfs_solve(puzzle):
    global depth_limit
    depth_limit = 1
    result = dls_solve(puzzle, depth_limit)
    while not result and not stop:
        depth_limit += 1
        result = dls_solve(puzzle, depth_limit)
    return result
```

Ưu điểm:

Bảo đảm tìm kiếm toàn bộ không gian trạng thái: IDDFS đảm bảo tìm kiếm toàn bộ không gian trạng thái, giống như DFS. Một khi đã hoàn thành tại một độ sâu, nó sẽ chuyển sang độ sâu tiếp theo. Điều này đảm bảo rằng nó không bỏ sót bất kỳ trạng thái nào.

Không tốn nhiều bộ nhớ: IDDFS thường không tốn nhiều bộ nhớ bằng cách thực hiện tìm kiếm DFS theo sâu tại mức hạn chế. Nó chỉ cần lưu trữ trạng thái tại độ sâu hiện tại trong ngăn xếp.

Nhược điểm:

Khả năng không tìm kiếm giải pháp tối ưu: IDDFS có thể không tìm thấy giải pháp tối ưu.

2.1.1.5 UCS (Uniform Cost Search)

```
def ucs_solve(puzzle):
    global total_nodes
    total_nodes = 0
    priority_queue = queue.PriorityQueue()
    visited = set()
    start_node = tuple(puzzle)
    priority_queue.put((0, start_node, []))
    while not priority_queue.empty() and not stop:
```



```

cost, current_node, path = priority_queue.get()
for item in possible_moves(current_node):
    node, pos_move = item
    if tuple(node) not in visited:
        total_nodes += 1
        new_path = path + [pos_move]
        visited.add(tuple(node))
        new_cost = cost + 1
        if is_solved(list(node)):
            return new_path
        priority_queue.put((new_cost, tuple(node), new_path))
return None

```

Ưu điểm:

Đảm bảo tối ưu: UCS đảm bảo tìm kiếm đường đi có chi phí tổng cộng thấp nhất đến trạng thái mục tiêu.

Đối với bài toán N-Puzzle thì UCS không quá tối ưu về tìm kiếm vì trọng số luôn bằng 1 cho mỗi ô di chuyển

Nhược điểm:

UCS đòi hỏi nhiều bộ nhớ và thời gian tính toán.

2.1.1.1.6 BS (Bidirectional Search)

```

def bidirectional_solve(puzzle):
    global total_nodes
    total_nodes = 0
    forward_open = deque([(tuple(puzzle), [])])
    backward_open = deque([(tuple(goal), [])])
    forward_visited = set()
    backward_visited = {}

    while forward_open and backward_open and not stop:
        forward_state, forward_path = forward_open.popleft()
        backward_state, backward_path = backward_open.popleft()
        if tuple(forward_state) in backward_visited:
            return forward_path + list(backward_visited[tuple(forward_state)][::-1])
        for item in possible_moves(forward_state):
            node, pos_move = item
            if tuple(node) not in forward_visited:
                total_nodes += 1

```

```

        forward_visited.add(tuple(node))
        new_path = forward_path + [pos_move]
        forward_open.append((node, new_path))
    for item in possible_moves(backward_state):
        node, pos_move = item
        if tuple(node) not in backward_visited:
            total_nodes += 1
            new_path = backward_path + [pos_move]
            backward_visited[tuple(node)] = new_path
            backward_open.append((node, new_path))
    return None

```

Ưu điểm:

Tốc độ tìm kiếm rất nhanh.

Nhược điểm:

Khả năng sử dụng bộ nhớ lớn: BS đòi hỏi lưu trữ thông tin về cả hai hướng của tìm kiếm.

2.1.1 Có thông tin

2.1.1.1 Greedy Search

```

def greedy_solve(puzzle):
    global total_nodes
    total_nodes = 0
    priority_queue = queue.PriorityQueue()
    visited = set()
    start_node = tuple(puzzle)
    priority_queue.put((0, start_node, []))
    while not priority_queue.empty() and not stop:
        _, current_node, path = priority_queue.get()
        for item in possible_moves(current_node):
            node, pos_move = item
            if tuple(node) not in visited:
                visited.add(tuple(node))
                total_nodes += 1
                new_path = path + [pos_move]
                new_cost = comparator(node)
                if is_solved(list(node)):
                    return new_path

```

```
priority_queue.put((new_cost, tuple(node), new_path))
return None
```

Ưu điểm:

Tìm kiếm nhanh chóng và có thể tìm ra một giải pháp gần đúng mà không yêu cầu nhiều thời gian tính toán. Dễ triển khai: Thuật toán tham lam thường dễ triển khai hơn so với các thuật toán tìm kiếm khác, như A* hoặc tìm kiếm đệ quy.

Nhược điểm:

Phụ thuộc vào hàm ước tính khoảng cách: Nếu ước tính khoảng cách không tốt, nó có thể dẫn đến kết quả không tốt nữa.

Không đảm bảo tối ưu: Thuật toán tham lam thường không đảm bảo tìm kiếm giải pháp tối ưu.

2.1.1.2 A* Search

```
def A_solve(puzzle):
    global total_nodes
    total_nodes = 0
    priority_queue = [(comparator(puzzle), 0, tuple(puzzle), [])]
    visited = set()
    while priority_queue and not stop:
        _, g_value, current_node, path = heapq.heappop(priority_queue)
        for item in possible_moves(current_node):
            node, pos_move = item
            if tuple(node) not in visited:
                visited.add(tuple(node))
                total_nodes += 1
                new_path = path + [pos_move]
                new_cost = g_value + 1 + comparator(node)
                if is_solved(list(node)):
                    return new_path
                heapq.heappush(priority_queue,
                               (new_cost, g_value + 1, tuple(node), _path,))
    return None
```

Ưu điểm:

Tìm kiếm tối ưu: A* đảm bảo tìm kiếm giải pháp tối ưu cho bài toán N-Puzzle.

A* có khả năng xử lý các không gian trạng thái lớn hơn, do đó, nó có thể được sử dụng trong các bài toán N-Puzzle có kích thước lớn.

Nhược điểm:

Phụ thuộc vào hàm ước tính khoảng cách: Thuật toán A* cũng dựa vào ước tính khoảng cách tương tự Greedy.

2.1.1.3 IDA* Search

```
def IDA_solve(puzzle):
    global total_nodes
    total_nodes = 0
    threshold = comparator(puzzle)
    def dls(puzzle, threshold):
        global total_nodes
        start_node = tuple(puzzle)
        stack = [(start_node, [], 0)]
        visited = set()
        min_cost = float("inf")
        while stack and not stop:
            current_node, path, g_value = stack.pop()
            f_value = g_value + comparator(current_node)
            if f_value > threshold:
                min_cost = min(min_cost, f_value)
                continue
            if is_solved(list(current_node)):
                return path, float("inf")
            for item in possible_moves(current_node):
                node, pos_move = item
                if tuple(node) not in visited:
                    total_nodes += 1
                    visited.add(tuple(node))
                    new_path = path + [pos_move]
                    stack.append((node, new_path, g_value + 1))
            return None, min_cost
    while not stop:
        result, new_threshold = dls(puzzle, threshold)
        if result is not None:
            return result
        if new_threshold == float("inf"):
            return None
        threshold = new_threshold
```

Ưu điểm:

IDA* có thể tìm kiếm theo chiều sâu với chiều sâu giới hạn trong mỗi lần lặp, giúp giảm bớt yêu cầu về bộ nhớ so với thuật toán A* truyền thống.

Nhược điểm:

IDA* không đảm bảo tìm ra giải pháp tối ưu theo chiều rộng vì nó có thể dừng lại khi đã tìm thấy giải pháp ở một chiều sâu nào đó, thậm chí khi có khả năng tìm ra giải pháp tốt hơn ở chiều sâu khác.

2.2 Cục bộ

2.2.1 Hill Climbing

```
def hc_solve(puzzle):
    global total_nodes
    total_nodes = 0
    start_node = tuple(puzzle)
    queue = deque([(start_node, [])])
    while queue and not stop:
        current_node, path = queue.popleft()
        node, pos_move = possible_moves(current_node)[0]
        cost = comparator(node)
        for i in range(1, len(possible_moves(current_node))):
            if cost >= comparator(node):
                cost = comparator(node)
                node, pos_move = possible_moves(current_node)[i]
                total_nodes += 1
            else:
                return None
        new_path = path + [pos_move]
        if is_solved(list(node)):
            return new_path
        queue.append((node, new_path))
    return None

def hc_loop(puzzle):
    global shuffling_count
    shuffling_count = 0
    path = hc_solve(puzzle)
    while not path and not stop:
        random_shuffle(puzzle)
```

```
shuffling_count += 1
path = hc_solve(puzzle)
return path
```

Ưu điểm:

So với một số thuật toán tìm kiếm khác, Hill Climbing yêu cầu ít bộ nhớ hơn vì nó chỉ giữ lại trạng thái hiện tại và trạng thái kế tiếp.

Nhược điểm:

Kết quả của Hill Climbing có thể phụ thuộc nhiều vào điểm khởi đầu. Nếu bắt đầu từ một điểm không tốt, thuật toán có thể rơi vào một giải pháp không tối ưu.

Hiện tại ở bài toán N-puzzle việc giải quyết với trạng thái ban đầu ngẫu nhiên dường như là không thể. Do đó đang sử dụng giải pháp thay thế là tìm trạng thái ngẫu nhiên cho tới khi Hill Climbing ra luôn đáp án.

2.2.2 Simulated Annealing

```
def sa_solve(puzzle):
    def acceptance_probability(cost, new_cost, temperature):
        if new_cost < cost:
            return 1.0
        return math.exp((cost - new_cost) / temperature)
    global total_nodes
    total_nodes = 0
    temperature = 1.0
    cooling_rate = 0.99
    current_state = tuple(puzzle)
    path = []
    while temperature > 0.01 and not stop:
        neighbors = possible_moves(current_state)
        next_state, move = random.choice(neighbors)
        current_cost = comparator(current_state)
        new_cost = comparator(next_state)
        if (acceptance_probability(current_cost, new_cost, temperature)
            > random.random()):
            total_nodes += 1
            path.append(move)
            current_state = next_state
        if is_solved(list(current_state)):
            return path
    temperature *= cooling_rate
```

```

return None
def sa_loop(puzzle):
    global shuffling_count
    shuffling_count = 0
    path = sa_solve(puzzle)
    while not path and not stop:
        random_shuffle(puzzle)
        shuffling_count += 1
        path = sa_solve(puzzle)
    return path

```

Ưu điểm:

Có khả năng tránh được cực tiểu cục bộ do có tỉ lệ xác suất chấp nhận chuyển sang trạng thái xấu hơn.

Nhược điểm:

Kết quả phụ thuộc nhiều vào giá trị các tham số định nghĩa ban đầu (ví dụ như xác suất chấp nhận trạng thái xấu, tốc độ làm nguội). Cần tinh chỉnh các giá trị này.

Dễ bị mắc kẹt ở các cực tiểu cục bộ nếu tốc độ làm nguội quá nhanh hoặc có quá ít biến động.

2.2.3 Beam Search

```

def beam_solve(puzzle):
    global total_nodes
    total_nodes = 0
    visited = set()
    start_node = tuple(puzzle)
    queue1 = deque([(start_node, [])])
    while queue1 and not stop:
        current_node, path = queue1.popleft()
        k = random.randint(2, len(possible_moves(current_node)))
        top_k_elements = []
        priority_queue = queue.PriorityQueue()
        for item in possible_moves(current_node):
            node, pos_move = item
            priority_queue.put((comparator(node), node, pos_move))
        for _ in range(k):
            if not priority_queue.empty():
                top_k_elements.append(priority_queue.get())
        for item in top_k_elements:

```

```
_, node, pos_move = item
if tuple(node) not in visited:
    visited.add(tuple(node))
    total_nodes += 1
    new_path = path + [pos_move]
    if is_solved(list(node)):
        return new_path
    queue1.append((node, new_path))
return None
```

Ưu điểm:

Beam Search giữ lại một số lượng giới hạn các ứng viên tốt nhất, giúp tránh mất thông tin quan trọng và tăng khả năng tìm ra giải pháp tối ưu.

Nhược điểm:

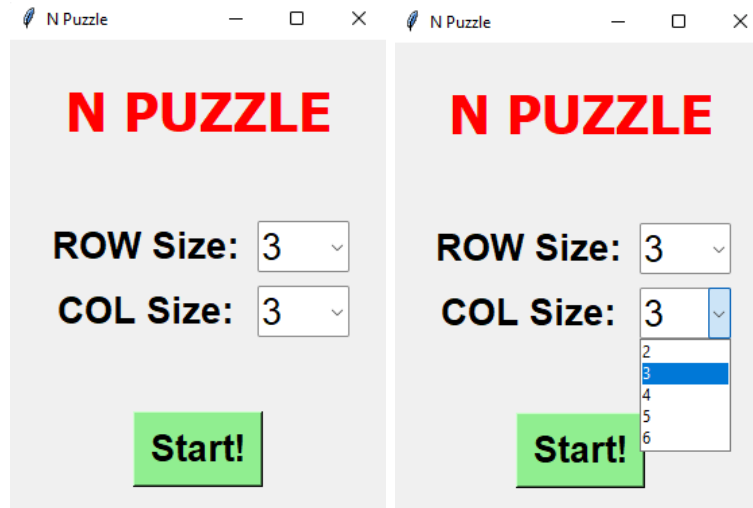
Beam Search có thể bị kẹt lại ở giải pháp cục bộ nếu không giữ lại đủ nhiều độ rộng để khám phá các lựa chọn tốt khác.

IV. THỰC NGHIỆM, ĐÁNH GIÁ, PHÂN TÍCH KẾT QUẢ

1. Giao diện

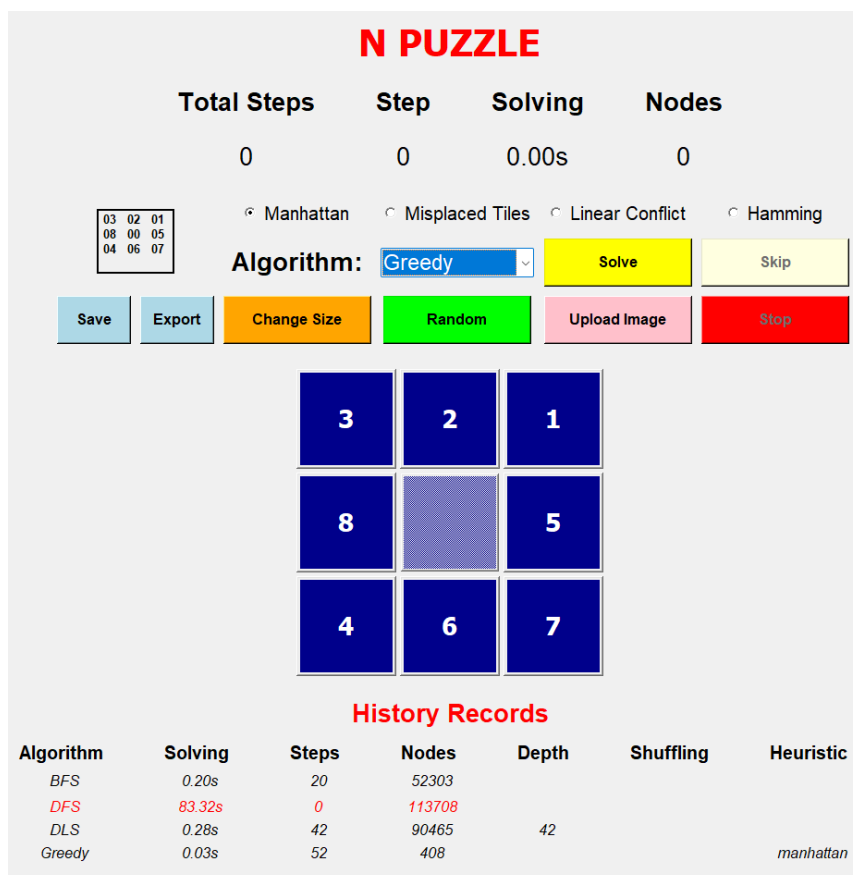
1.1 Tùy chỉnh kích thước của trò chơi

Ta có thể thay đổi kích thước số hàng và số cột (giới hạn từ 2 đến 6) của Puzzle để tạo ra nhiều kích thước khác nhau.



1.2 Hướng dẫn sử dụng

Giao diện của trò chơi ứng dụng.

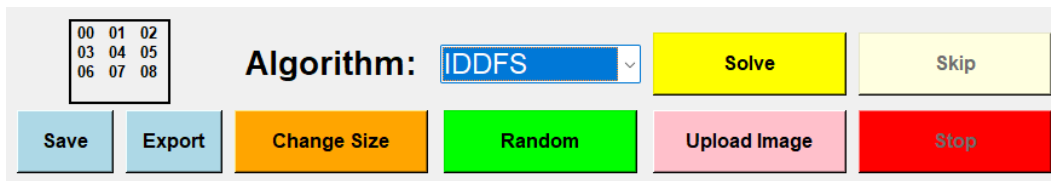


1.2.1 Thông số biểu diễn

Thông số để theo dõi gồm số bước, thời gian giải, số trạng thái đã duyệt qua. Ngoài ra còn có độ sâu và số lần xáo trộn ngẫu nhiên tùy theo từng thuật toán được lựa chọn.

Total Steps	Step	Solving	Nodes	Depth
0	0	0.00s	0	1

1.2.2 Các nút điều khiển



- **Save và Export:** Lưu và xuất trạng thái, dùng để lưu một trạng thái ngẫu nhiên và xuất trạng thái đó để theo dõi và so sánh được nhiều thuật toán trên cùng một trạng thái.
- **Change size:** Thay đổi kích thước N, khi nhấn vào sẽ quay lại giao diện chọn kích thước ban đầu.
- **Random:** Xáo trộn ngẫu nhiên những ô số.
- **Upload Image:** Chèn hình ảnh để hiện lên các ô số.
- **Algorithm Combobox:** Lựa chọn thuật toán AI cần thực hiện.
- **Solve:** Giải bằng thuật toán AI.
- **Skip:** Bỏ qua trạng thái đang di chuyển các ô số của thuật toán nếu số bước di chuyển quá lớn (sau khi đã tìm ra lời giải).
- **Stop:** Dừng hẳn thuật toán đang giải (trong lúc đang tìm lời giải).

1.2.3 Các ô số

Người dùng có thể chọn vào các ô số để di chuyển, giải quyết bài toán. Ngoài ra, có thể sử dụng tính năng chèn hình thay đổi giao diện của các ô số.



1.2.3 Lịch sử các bản ghi đã giải

Bản ghi lưu lại những thông số biểu diễn trước đó để dễ theo dõi và đánh giá. Những thuật toán nào đã giải được sẽ hiện màu đen, ngược lại sẽ hiện màu đỏ.

History Records						
Algorithm	Solving	Steps	Nodes	Depth	Shuffling	Heuristic
BFS	0.20s	20	52303			
DFS	83.32s	0	113708			
DLS	0.28s	42	90465	42		
Greedy	0.03s	52	408			manhattan

2. Đánh giá, phân tích kết quả

2.1 Ghi chú

2.1.1 Lựa chọn ước lượng độ dài Manhattan

Đối với các hàm Heuristic, nhận thấy rằng khoảng cách Manhattan là tối ưu nhất (duyệt ít trạng thái nhất) nên chúng tôi quyết định lựa chọn ước lượng này cho mọi thuật toán tìm kiếm có thông tin và tìm kiếm cục bộ.

Algorithm	Solving	Steps	Nodes	Depth	Shuffling	Heuristic
Greedy	0.05s	86	740			manhattan
Greedy	0.30s	62	1136			misplaced tiles
Greedy	0.22s	70	796			linear conflict
Greedy	4.94s	144	46485			hamming

2.1.2 Hill Climbing và Simulated Annealing

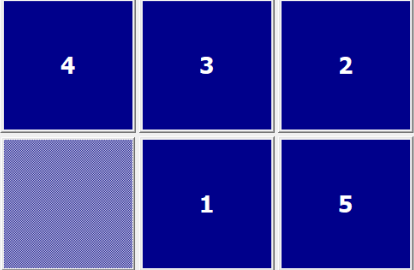
Đối với thuật toán Hill Climbing và Simulated Annealing: thuật toán cho phép xáo trộn ngẫu nhiên lại trạng thái ban đầu hoàn hảo nhất cho việc tìm kiếm lời giải nên không dùng để so sánh.

History Records						
Algorithm	Solving	Steps	Nodes	Depth	Shuffling	Heuristic
Hill Climbing	0.10s	1	1		10	manhattan
Hill Climbing	0.12s	8	11		10	manhattan
Sim-Annealing	0.33s	25	25		8	manhattan
Sim-Annealing	0.55s	13	13		14	manhattan
Sim-Annealing	0.08s	6	6		2	manhattan

Ngoài ra, chúng tôi đánh giá rằng Simulated Annealing có xu hướng duyệt nhiều trạng thái hơn và số lần xáo trộn ngẫu nhiên để tìm lời giải cũng sẽ ít hơn so với Hill Climbing vì nó có xác suất chấp nhận chuyển sang trạng thái xấu hơn kỹ thuật giảm nhiệt độ.

2.1.3 Đánh giá theo kích thước của bài toán

Đối với các kích thước nhỏ như 2x2, 2x3, 3x2: Tất cả các thuật toán đều giải quyết được bài toán với thời gian chạy là 0.00s với cùng 1 trạng thái bắt đầu. Vì vậy, chưa có cơ sở để phân tích đánh giá.



History Records

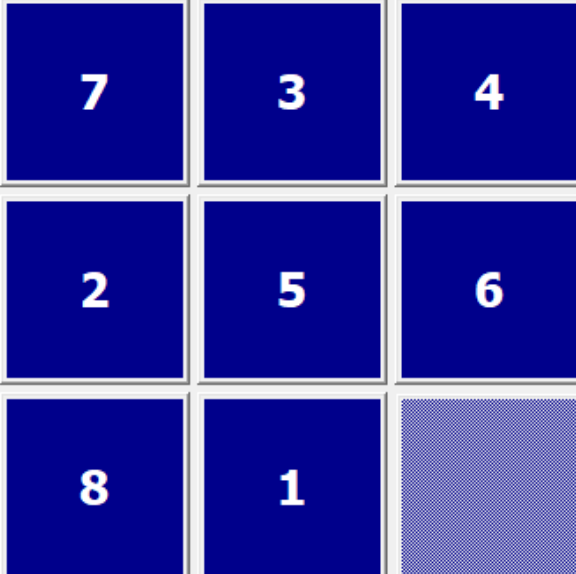
Algorithm	Solving	Steps	Nodes	Depth	Shuffling	Heuristic
DFS	0.00s	5	9			
IDDFS	0.00s	5	46	5		
Greedy	0.00s	5	9			manhattan
A* Search	0.00s	5	9			manhattan
Bidirectional	0.00s	5	22			

Còn đối với các kích thước lớn hơn 3x3, các thuật toán biểu diễn được thông số thời gian và số trạng thái duyệt được lập trình theo kỹ thuật chia luồng. Tuy nhiên, giải thuật tìm kiếm rất lâu và chỉ có một số thuật toán giải được nên cũng không có cơ sở đánh giá được nhiều.

Vì vậy, chúng tôi sẽ tập trung đánh giá độ hiệu quả thuật toán trên kích thước 3x3 vì nó là kích thước cơ bản dễ theo dõi và độ chính xác cao nhất.

2.2 So sánh, đánh giá dựa trên kích thước 3x3

Trạng thái ngẫu nhiên được so sánh:



Các bản ghi được lưu lại.

History Records						
Algorithm	Solving	Steps	Nodes	Depth	Shuffling	Heuristic
BFS	0.29s	22	73299			
DFS	5.28s	17196	31077			
DLS	0.19s	30	62693	30		
IDDFS	2.27s	36	747153	37		
UCS	0.68s	22	78742			

Algorithm	Solving	Steps	Nodes	Depth	Shuffling	Heuristic
Greedy	0.04s	124	637			manhattan
A* Search	0.02s	22	473			manhattan
IDA* Search	0.05s	24	763			manhattan
Bidirectional	0.02s	22	2296			
Beam Search	1.32s	22	10004			manhattan

Ta thấy các thuật toán đã đưa ra được lời giải tối ưu với số bước đi ngắn nhất là BFS, UCS, A*, Beam Search, và Bi-Search. Ở đây có Beam Search là tìm kiếm cục bộ nhưng vẫn tìm ra được lời giải tối ưu do nó có cơ chế duyệt qua các trạng thái tiềm năng và lựa chọn một số lượng hữu hạn các trạng thái tốt nhất để mở rộng tiếp tục tìm kiếm.

Đối với các thuật toán tìm kiếm mù như BFS và Bi-Search, vì Bi-Search có cơ chế duyệt hàng đợi từ 2 đầu trạng thái đích và trạng thái ngẫu nhiên ban đầu nên số trạng thái đã duyệt cũng như thời gian duyệt cũng sẽ thấp hơn BFS khi BFS chỉ duyệt hàng đợi từ trạng thái bắt đầu. BFS và Bi-Search đều có thể tìm ra lời giải tối ưu nhưng nó tùy thuộc vào phạm vi bài toán, nếu quá lớn sẽ không thể nào tìm ra. Ngoài ra UCS với mức chi phí mỗi lần di chuyển là 1 cũng được tính như là tìm kiếm mù, như vậy nó cũng sẽ có cơ chế giống BFS nhưng thời gian giải chậm hơn cho nó phải so sánh chi phí mỗi lần di chuyển.

Đối với các thuật toán duyệt theo chiều sâu như DFS, DLS, IDDFS. Ta có thể thấy được độ hiệu quả chênh lệch rõ ràng nhất. DFS là phiên bản thô sơ, duyệt theo hình thức vét cạn từ đầu vào sâu xuống tìm lời giải nên thời gian và số bước di chuyển sẽ là cao nhất. Trong khi đó DLS được giới hạn tại độ sâu 30 và tới độ sâu đó nó sẽ quay lại duyệt hết tất cả đường còn lại trong ngăn xếp nên có số trạng thái duyệt nhiều hơn nhưng thời gian lại ngắn hơn. Nếu ta đặt độ sâu là tối ưu là 22 và duyệt lại nhiều lần thì thuật toán này cũng sẽ có khả năng đưa ra kết quả tối ưu vì cơ chế lựa chọn đường đi là ngẫu nhiên. Còn lại, dù là bản cải tiến của DLS, tuy nhiên sau mỗi độ sâu tăng dần nó phải duyệt lại nhiều lần nên số trạng thái duyệt là rất lớn và vì theo cơ chế tìm đường ngẫu nhiên không có thông tin nên nó cũng sẽ đưa ra số bước duyệt không cố định.

Đối với các thuật toán tìm kiếm có thông tin như Greedy, A*, và IDA*. Ta có thể thấy Greedy cũng có khả năng tìm ra lời giải với số trạng thái duyệt sẽ ít hơn rất

nhiều so với các thuật toán tìm kiếm mù. A^* ở đây có thể là thuật toán cải tiến hoàn thiện nhất vì nó vừa duyệt với số trạng thái ít nhất trong thời gian ngắn, tuy nhiên với nhiều trường hợp của nhiều bài toán khác nhau, Greedy có thể đưa ra lời giải nhanh hơn nhưng số bước không tối ưu còn A^* sẽ giải chậm hơn nhưng số bước tối ưu. Còn IDA* là thuật toán được kết hợp từ cơ chế đào sâu và tìm kiếm có thông tin, số bước duyệt của nó cũng sẽ thấp hơn so với các thuật toán duyệt sâu không có thông tin, tuy nhiên lời giải không tối ưu nhưng ở mức có thể chấp nhận được.

V. KẾT LUẬN

Bài toán N-Puzzle, một game kinh điển, và việc ứng dụng thuật toán AI vào game là một trải nghiệm tuyệt vời đem đến cho chúng tôi kiến thức, kinh nghiệm khi phát triển các thuật toán AI, củng cố kiến thức cho mỗi người, đồng thời học được thêm cách phát triển giao diện, cách xây dựng thuật toán dựa trên ngôn ngữ lập trình Python.

Hướng phát triển tương lai: Giao diện có thể phát triển hơn nữa, đồng thời xây dựng game theo kỹ thuật hướng đối tượng sẽ giúp cải thiện kỹ năng lập trình nâng cao hơn. Ngoài ra, nếu có cơ hội, chúng tôi sẽ mong muốn tìm hiểu được nhiều thuật toán tìm kiếm hơn để áp dụng vào bài toán giải được với nhiều kích thước lớn hơn.

TÀI LIỆU THAM KHẢO

<https://viblo.asia/p/data-structure-algorithm-graph-algorithms-breadth-first-search-bfs-gwd43kMM4X9>

<https://viblo.asia/p/data-structure-algorithm-graph-algorithms-depth-first-search-dfs-qPoL7zyXJvk>

<https://iq.opengenus.org/depth-limited-search/>

<https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/>

<https://www.educative.io/answers/what-is-uniform-cost-search>

<https://www.geeksforgeeks.org/bidirectional-search/>

<https://www.geeksforgeeks.org/greedy-algorithms/>