TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI
TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

…📖…

BÁO CÁO HỌC PHẦN TRÍ TUỆ NHÂN TẠO

ĐỀ TÀI : ỨNG DỤNG CÁC THUẬT TOÁN TÌM KIẾM ĐỂ GIẢI BÀI TOÁN GHÉP TRANH (N-PUZZLE)

GVHD : **Ths. Mai Thanh Hồng**

Mã lớp : 20242IT6094003

Nhóm : 11

Hà Nội, Năm 2025

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI
TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

…

BÁO CÁO HỌC PHẦN TRÍ TUỆ NHÂN TẠO

ĐỀ TÀI : ỨNG DỤNG CÁC THUẬT TOÁN TÌM KIẾM ĐỂ GIẢI BÀI TOÁN GHÉP TRANH (N-PUZZLE)

GVHD : **Ths. Mai Thanh Hồng**

Mã lớp : 20242IT6094003

Nhóm : 11

Sinh viên thực hiện : Hồ Văn Cường - 2023605691
Vũ Huy Đạt - 2023604757
Nguyễn Bá Đức - 2023607032
Đỗ Huy Hoàng - 2023604555

Hà Nội, Năm 2025

MỤC LỤC

DANH MỤC HÌNH ẢNH

LỜI NÓI ĐẦU

N-puzzle không chỉ đơn thuần là một tựa game ghép số, tranh mang tính giải trí mà còn mang tính học thuật sâu sắc, là ví dụ điển hình trong lĩnh vực Trí tuệ nhân tạo về các bài toán tìm kiếm trạng thái. Bài toán ban đầu đặt ra một ma trận vuông có kích thước n*n với các ô số và ô ảnh ngẫu nhiên, nhiệm vụ của người chơi là di chuyển ô trống sao cho đưa được tất cả các ô về đúng trạng thái đích, ảnh về đúng ảnh ban đầu thì trò chơi coi như kết thúc.

Đề tài mà nhóng chúng em thực hiện không chỉ dừng lại là giải quyết bài toán một cách đơn thuần, chúng em muốn hướng tới mục tiêu đánh giá và so sánh hiệu quả của các thuật toán tìm kiếm trong một số tình huống cụ thể. Để đưa ra lời giải tối ưu nhất sao cho số bước di chuyển ô trống ít nhất đòi hỏi người chơi phải có kinh nghiệm cao để có thể xử lý. Trong bài giảng về thuật toán tìm kiếm thuộc môn " Trí tuệ nhân tạo " do GV.Mai Thanh Hồng giảng dạy đã cho nhóm em cách giải quyết bài toán đề ra ở trên với số bước ngắn nhất thông qua 2 thuật toán tìm kiếm đó là BFS (Best First Search) và A^{KT}.

Với đề tài "Ứng dụng các thuật toán tìm kiếm để giải bài toán ghép tranh (N-puzzle)" nhóm em đã xây dựng bài toán với các kích thước có thể thay đổi 3*3, 4*4, 5*5 tương ứng với các cấp độ của trò chơi.

Chúng em hy vọng rằng, đề tài này sẽ phần nào minh họa sinh động các khái niệm trừu tượng trong lĩnh vực trí tuệ nhân tạo. Trong quá trình thực hiện đề tài thì không thể tránh khỏi những sai sót, nhóm chúng em rất mong nhận được sự đánh giá và góp ý của cô để hoàn thiện cho đề tài này.

CHƯƠNG 1: GIỚI THIỆU BÀI TOÁN N-PUZZLE

1.1 Sơ lược về bài toán N-puzzle

Bài toán N-Puzzle, còn được biết đến với các tên gọi khác như "Gem Puzzle" hay "Sliding Puzzle", là một trong những bài toán kinh điển trong lĩnh vực trí tuệ nhân tạo (Artificial Intelligence – AI) và tìm kiếm. Bài toán này không chỉ mang tính giải trí mà còn chứa đựng những yếu tố học thuật sâu sắc, phù hợp cho việc nghiên cứu và ứng dụng các thuật toán giải quyết vấn đề, đặc biệt là trong các môn học về trí tuệ nhân tạo, tối ưu hóa, và lý thuyết đồ thị.

Vị trí của các số được sắp xếp ngẫu nhiên ban đầu và người chơi sẽ thực hiện việc dịch chuyển các ô số vào vị trí ô trống theo bốn hướng cơ bản: lên, xuống, trái, phải (không được đi chéo). Mục tiêu cuối cùng là đưa bảng về trạng thái đích – tức là sắp xếp các số theo thứ tự tăng dần từ trái sang phải, từ trên xuống dưới, với ô trống nằm ở góc dưới bên phải. Một ví dụ quen thuộc của bài toán này là 8-Puzzle, ứng với bảng 3x3 (9 ô), trong đó các số từ 1 đến 8 phải được sắp xếp đúng vị trí.

Bài toán N-Puzzle là một dạng cụ thể của bài toán tổ hợp, trong đó không phải mọi trạng thái khởi đầu đều có thể giải được. Có những trạng thái không thể dẫn về trạng thái đích, bất kể người chơi di chuyển thế nào. Do đó, một bước quan trọng trong việc giải bài toán này là kiểm tra tính khả giải của trạng thái ban đầu. Điều này được xác định dựa trên số lần nghịch thế trong chuỗi các số. Nếu chuỗi trạng thái không thoả mãn điều kiện khả giải thì việc tìm lời giải là vô nghĩa.

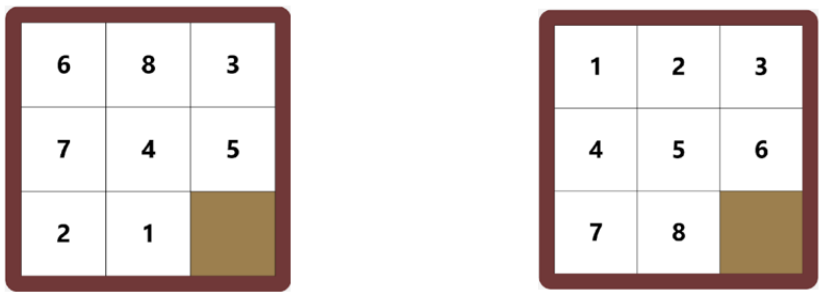
N-Puzzle là một ví dụ tiêu biểu cho bài toán tìm kiếm trong không gian trạng thái. Việc giải bài toán tương đương với việc tìm đường đi ngắn nhất từ trạng thái ban đầu đến trạng thái đích, trong một không gian có thể có hàng ngàn hoặc hàng triệu trạng thái trung gian. Các thuật toán thường được sử dụng để giải bài toán này

bao gồm: Breadth-First Search (BFS), Depth-First Search (DFS), đặc biệt là A*, kết hợp với các hàm heuristic để đánh giá chi phí ước lượng còn lại.

Ngoài tính ứng dụng cao trong giảng dạy, bài toán N-Puzzle còn có nhiều ứng dụng thực tế như trong robot học, thiết kế hệ thống định tuyến, xử lý ảnh, hay các hệ thống lập kế hoạch. Với quy mô bài toán tăng lên như 15-Puzzle (4x4) hay 24-Puzzle (5x5), độ khó và kích thước không gian trạng thái tăng lên nhanh chóng, làm cho bài toán trở thành một thử thách thực sự cho các thuật toán giải quyết vấn đề.

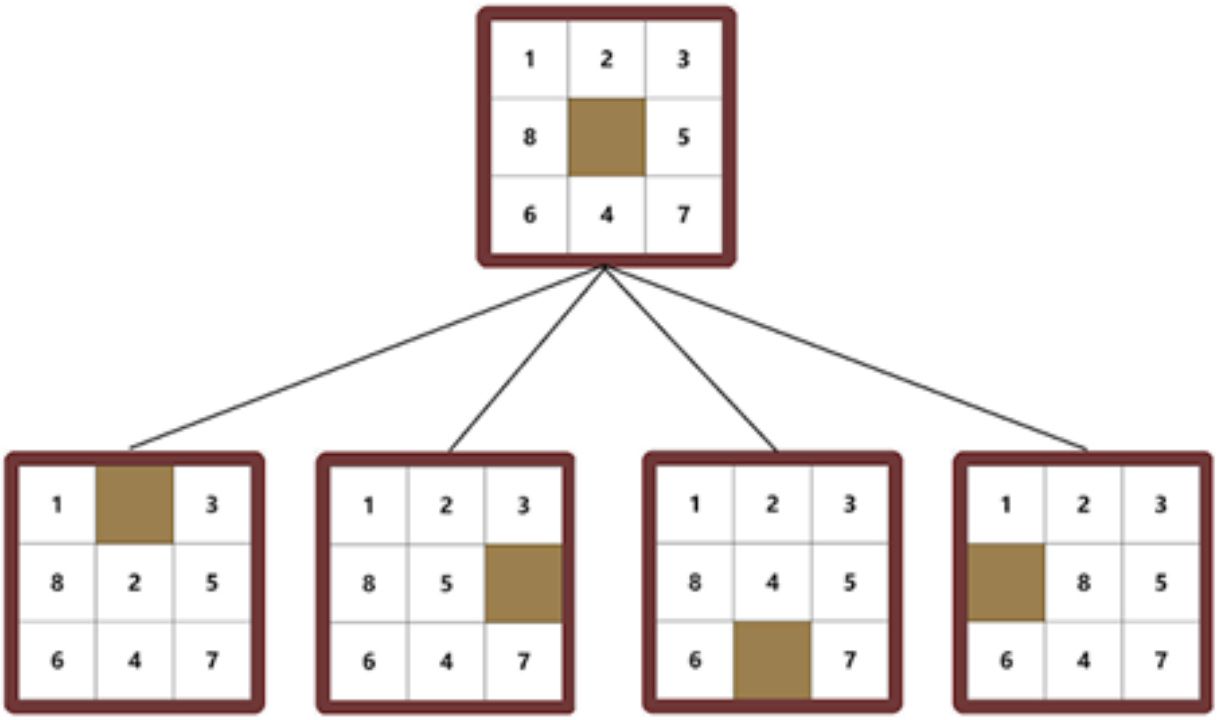
1.2 Minh họa bài toán bằng hình ảnh

Bên dưới là hình minh họa về bài toán 8-Puzzle với 1 bảng kích thước 3*3 và các ô số được đánh lần lượt từ 1 đến 8:



Hình 1. 1 : Minh họa về bài toán 8-Puzzle với bảng 3*3

Tại mỗi trạng thái người chơi có thể dịch chuyển từ 2 → 4 vị trí khác nhau:



Hình 1. 2 : Minh họa trạng thái người chơi dịch chuyển sang các vị trí khác

CHƯƠNG 2: GIẢI QUYẾT BÀI TOÁN BẰNG THUẬT TOÁN BFS

2.1. Giới thiệu thuật toán

Thuật toán BFS (Breadth-First Search) hay còn gọi là thuật toán tìm kiếm theo chiều rộng, là một kỹ thuật duyệt đồ thị quan trọng trong lĩnh vực khoa học máy tính. BFS xuất phát từ đỉnh gốc và duyệt qua tất cả các đỉnh lân cận trước khi đi sâu hơn vào các mức tiếp theo. Trong bối cảnh bài toán tìm đường hoặc giải câu đố (như xếp hình, trượt ô số,...), BFS đóng vai trò như một công cụ để tìm ra đường đi ngắn nhất từ điểm bắt đầu đến mục tiêu.

Không giống như DFS (Depth-First Search), BFS không đi sâu ngay vào một nhánh, mà ưu tiên mở rộng theo lớp, tức là các đỉnh cách gốc 1 bước sẽ được duyệt trước, sau đó là các đỉnh cách 2 bước. Cơ chế này giúp BFS đảm bảo rằng khi đến được đích, ta đã có ngay lời giải tối ưu về số bước đi.

BFS hoạt động dựa trên hai cấu trúc chính:

- Hàng đợi : Lưu các trạng thái đang chờ xử lý.
- Danh sách đánh dấu : Đảm bảo không xử lý trùng trạng thái.

Quy trình cơ bản như sau:

1. Khởi tạo hàng đợi với đỉnh xuất phát.
2. Trong khi hàng đợi chưa rỗng:

- Lấy ra một trạng thái từ đầu hàng.
- Nếu đó là trạng thái đích → dừng thuật toán.
- Nếu chưa → tạo các trạng thái kế tiếp, thêm vào hàng đợi nếu chưa từng xét.

Cơ chế lặp đơn giản nhưng hiệu quả này là nền tảng cho nhiều ứng dụng trong thực tế và bài toán học thuật.

2.2. Đặc điểm

2.2.1. Ưu điểm

- Dễ lập trình và triển khai: BFS có cách cài đặt rõ ràng, dễ hiểu, phù hợp với người mới bắt đầu học thuật toán. Việc sử dụng hàng đợi và danh sách kiểm tra trùng lặp là các kỹ thuật cơ bản nhưng hiệu quả.
- Đảm bảo tìm được lời giải nếu tồn tại: Với đồ thị hữu hạn và liên thông, BFS luôn tìm ra đường đi từ đỉnh gốc đến đích, nếu tồn tại. Điều này làm cho BFS trở thành một trong những giải pháp tin cậy cho các bài toán tìm đường hoặc tìm trạng thái.
- Tìm được đường đi ngắn nhất: BFS duyệt theo từng lớp, vì vậy đích khi được tìm thấy lần đầu tiên sẽ luôn nằm trên đường đi ngắn nhất từ gốc. Không cần kiểm tra các nhánh sâu hơn, không bị rơi vào bẫy vòng lặp như DFS.
- Có thể áp dụng cho nhiều loại bài toán: BFS phù hợp không chỉ với đồ thị mà còn với các không gian trạng thái tổng quát, như chuyển đổi chuỗi, trò chơi logic, bài toán xếp hình, di chuyển robot, giải mê cung, ...

2.2.2. Nhược điểm

- Tốn nhiều bộ nhớ: Do lưu toàn bộ các trạng thái cùng cấp độ và tất cả trạng thái con vào hàng đợi, BFS có thể tốn rất nhiều bộ nhớ nếu không gian trạng thái lớn.
- Tốc độ chậm trong không gian lớn: Khi số trạng thái tăng nhanh, thời gian duyệt cũng tăng tương ứng. Thuật toán có thể phải kiểm tra hàng nghìn trạng thái không cần thiết trước khi chạm được đích.

- Thiếu thông tin định hướng: Là một thuật toán mù, BFS không sử dụng thông tin nào về khoảng cách tới đích. Do đó, nó không tối ưu về hiệu suất so với các thuật toán như A*.
- Không phù hợp với không gian vô hạn hoặc quá lớn: Trong các bài toán mà không gian trạng thái là vô hạn, BFS có thể không dừng được hoặc chiếm quá nhiều tài nguyên.

2.2.3. Ứng dụng thực tế

Thuật toán BFS được sử dụng rộng rãi trong nhiều lĩnh vực, không chỉ trong lý thuyết đồ thị mà còn cả trong trí tuệ nhân tạo, game, và tối ưu hoá. Một số ví dụ cụ thể:

- Giải bài toán xếp hình, trò chơi logic: Ví dụ trò chơi 8 puzzle hoặc Rubik dạng đơn giản, mỗi trạng thái có thể được biểu diễn bằng một mảng các số nguyên. BFS sẽ duyệt qua các trạng thái sinh từ trạng thái ban đầu để tìm cách về đích.
- Tìm đường đi trong mê cung hoặc bản đồ: Các trò chơi 2D có thể sử dụng BFS để tìm đường đi tối ưu cho nhân vật, tránh chướng ngại vật.
- Tìm mức độ kết nối trong mạng xã hội: Dùng để xác định khoảng cách giữa hai người dùng, hoặc nhóm người gần nhau nhất.
- Truy vết đường đi ngắn nhất: Trong quá trình duyệt, ta lưu lại trạng thái cha của mỗi trạng thái. Khi tìm được trạng thái đích, việc lần ngược lại theo các cha sẽ cho ta đường đi đầy đủ từ gốc đến đích.

2.2.4. Mô phỏng minh hoạ

Ví dụ đơn giản: Tìm đường đi trong mê cung 2D từ điểm (0,0) đến điểm (n,m):

- Bắt đầu từ (0,0), BFS duyệt qua tất cả các ô hàng xóm (trên, dưới, trái, phải) nếu ô đó chưa bị đánh dấu và không phải tường.
- Lưu lại vị trí cha của mỗi ô để truy vết sau này.
- Khi đến được (n,m), kết thúc và in ra đường đi bằng cách lần ngược từ đích đến gốc.

2.3. Cài đặt thuật toán:

2.3.1 Quy ước

- Lưu trữ: Sử dụng hai danh sách DONG và MO hoạt động theo kiểu FIFO (hàng đợi).
    - DONG: Chứa các đỉnh đã xét
    - MO: chứa các đỉnh đang xét

2.3.2 Các bước cài đặt

- Bước 1:

  MO = ∅; MO = MO ∪ {T0}
- Bước 2:

```
while (MO != ∅ )
{       n = get(MO) // lấy đỉnh đầu trong danh sach MO
        if (n==TG) // nếu n là trạng thái kết thúc
                return TRUE // tìm kiếm thành công, dừng
        DONG = DONG ∪ {n} //đánh dấu n đã được xét
        for các đỉnh kề v của n
                if (v chưa đc xét) //v chưa ở trong DONG
                        MO = MO ∪ {v} //đưa v vào cuối DS MO
                                father(v)=n// lưu lại vết đường đi từ n đến v
}
```

2.3.3 Thiết kế thuật toán

- Hàm kiểm tra xem trạng thái hiện tại đã là trạng thái đích chưa

```
from collections import deque

# Hàm kiểm tra xem trạng thái hiện tại có phải trạng thái đích
chưa
def is_goal(state, goal):
    return state == goal
```

- Hàm tìm kiếm các trạng thái con từ trạng thái hiện tại

```
# Hàm tìm các trạng thái con từ trạng thái hiện tại
def get_neighbors(state):
    neighbors = []
    n = int(len(state) ** 0.5)  # Tính kích thước lưới (ví dụ
3x3 nếu 8-puzzle)
    zero_index = state.index(0)  # Tìm vị trí ô trống (ô số 0)
    x, y = divmod(zero_index, n)  # Tính hàng (x) và cột (y)

    # Các hướng di chuyển: lên, xuống, trái, phải
    moves = [(-1,0), (1,0), (0,-1), (0,1)]

    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < n and 0 <= new_y < n:
            new_zero_index = new_x * n + new_y
            new_state = list(state)
            # Đổi chỗ 0 với số kế bên
            new_state[zero_index], new_state[new_zero_index] =
new_state[new_zero_index], new_state[zero_index]
            neighbors.append(tuple(new_state))

    return neighbors
```

- Hàm chính của thuật toán BFS

```
# Hàm chính BFS
def bfs(start, goal):
    MO = deque()
```

```
    MO.append((start, [start]))  # Mỗi phần tử lưu (trạng thái
hiện tại, đường đi từ đầu tới đó)
    DONG = set()
    DONG.add(start)

    while MO:
        current_state, path = MO.popleft()

        if is_goal(current_state, goal):
            return path  # Trả về đường đi

        for neighbor in get_neighbors(current_state):
            if neighbor not in DONG:
                DONG.add(neighbor)
                MO.append((neighbor, path + [neighbor]))

    return None  # Không tìm thấy lời giải
```

- Ví dụ sử dụng cụ thể

```
# Ví dụ sử dụng
if __name__ == "__main__":
    start = (0, 4, 2,
             6, 5, 1,
             8, 7, 3)

    goal = (1, 2, 3,
            4, 5, 6,
            7, 8, 0)

    path = bfs(start, goal)

    if path:
        print(f"Số bước: {len(path) - 1}")
        for state in path:
            for i in range(0, len(state), 3):
                print(state[i:i+3])
            print("---------")
    else:
        print("Không tìm thấy lời giải.")
```

2.3.4 Ví dụ

Ví dụ với hình ma trận cỡ 3x3 cho đoạn code cài đặt thuật toán trên ta thu được kết quả như sau :

Hình 2. 1 : Minh họa kết quả thu được khi cài đặt thuật toán

- Tổng số bước để ra được kết quả : 22 bước
- Số node đã duyệt: 448128
- Tổng số node trên cây: 777963
- Thời gian tìm kiếm: 324ms

2.3.5 Đánh giá

Dễ cài đặt, phù hợp với các bài toán có không gian trạng thái nhỏ nhưng không tốt cho các bài toán yêu cầu hiệu suất cao và dung lượng lớn, không thông minh vì BFS duyệt mù từng lớp, không ưu tiên trạng thái gần đích. Điều này khiến nó tốn công hơn các thuật toán heuristic như A*.

So sánh với các thuật toán khác: Khi so sánh với thuật toán heuristic như A*, rõ ràng BFS kém hiệu quả hơn về mặt hiệu suất. A* sử dụng hàm đánh giá để định hướng việc mở rộng các node, giúp thuật toán đi đúng hướng và thường chỉ duyệt qua một phần nhỏ của không gian trạng thái. Nhờ đó, A* vừa đảm bảo tìm được lời giải tối ưu, vừa tiết kiệm thời gian và bộ nhớ hơn đáng kể.

CHƯƠNG 3: GIẢI QUYẾT BÀI TOÁN BẰNG THUẬT TOÁN A*

3.1. Giới thiệu thuật toán

3.1.1. Giới thiệu

Thuật toán A* được mô tả lần đầu vào năm 1968 bởi Peter Hart, Nils Nilsson, và Bertram Raphael. Trong bài báo của họ, thuật toán được gọi là thuật toán A; khi sử dụng thuật toán này với một đánh giá heuristic thích hợp sẽ thu được hoạt động tối ưu, do đó mà có tên A*.

Trong khoa học máy tính, A* là thuật toán tìm kiếm trong đồ thị. Thuật toán này tìm một đường đi từ một nút khởi đầu tới một nút đích cho trước. Thuật toán này sử dụng một "đánh giá heuristic" để xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó. Thuật toán này duyệt các nút theo thứ tự của đánh giá heuristic này. Do đó, thuật toán A* là một ví dụ của tìm kiếm theo lựa chọn tốt nhất (*best-first search*).

Xét bài toán tìm đường - bài toán mà A* thường được dùng để giải. A* xây dựng tăng dần tất cả các tuyến đường từ điểm xuất phát cho tới khi nó tìm thấy một đường đi chạm tới đích. Tuy nhiên, cũng như tất cả các thuật toán tìm kiếm có thông tin, nó chỉ xây dựng các tuyến đường có khả năng dẫn về phía đích.

Để biết những tuyến đường nào có khả năng sẽ dẫn tới đích, A* sử dụng một đánh giá heuristic về khoảng cách từ điểm bất kỳ cho trước tới đích. Trong trường hợp tìm đường đi, đánh giá này có thể là khoảng cách đường chim bay - một đánh giá xấp xỉ thường dùng cho khoảng cách của đường giao thông.

Điểm khác biệt của A* đối với tìm kiếm theo lựa chọn tốt nhất là nó còn tính đến khoảng cách đã đi qua. Điều đó làm cho A* đầy đủ và tối ưu, nghĩa là A* sẽ luôn luôn tìm thấy đường đi ngắn nhất nếu tồn tại một đường đi như vậy. A* không đảm bảo sẽ chạy nhanh hơn các thuật toán tìm kiếm đơn giản hơn. Trong một môi

trường dạng mê cung, cách duy nhất để đến đích có thể là trước hết phải đi về phía xa đích và cuối cùng mới quay lại. Trong trường hợp đó, việc thử các nút theo thứ tự gần đích hơn thì được thử trước có thể gây tốn thời gian.

3.1.2. Mô tả

A* lưu giữ một tập các lời giải chưa hoàn chỉnh, nghĩa là các đường đi qua đồ thị, bắt đầu từ nút xuất phát. Tập lời giải này được lưu trong một hàng đợi ưu tiên. Thứ tự ưu tiên gán cho một đường đi n được quyết định bởi hàm $f(n) = g(n) + h(n)$. Trong đó :

- $g(n)$ là chi phí của đường đi từ nút gốc đến nút hiện tại hiện tại n, nghĩa là số các bước đã đi từ nút gốc đến nút hiện tại.
- $h(n)$ là hàm đánh giá heuristic về chi phí nhỏ nhất để đến đích từ n (chi phí ước lượng từ nút hiện tại n đến đích) . Ví dụ, nếu "chi phí" được tính là khoảng cách đã đi qua, khoảng cách đường chim bay giữa hai điểm trên một bản đồ là một đánh giá heuristic cho khoảng cách còn phải đi tiếp.
- $f(n)$ là tổng thể ước lượng của đường đi qua nút hiện tại n đến đích và chi phí từ nút ban đầu đến nút hiện tại.

Hàm $f(n)$ có giá trị càng thấp thì độ ưu tiên của càng cao.

Một ước lượng Heuristic $h(n)$ được xem là chấp nhận được nếu đối với mọi nút n: $0 \leq h(n) \leq h^*(n)$. Trong đó: $\mathbf{h^*(n)}$ là chi phí thực tế từ nút n đến đích

3.2. Đặc điểm

Nếu không gian các trạng thái là hữu hạn và có giải pháp để tránh việc xét (lặp) lại các trạng thái, thì giải thuật A* là hoàn chỉnh – nhưng không đảm bảo là tối ưu.

Nếu không gian các trạng thái là hữu hạn và không có giải pháp để tránh việc xét (lặp) lại các trạng thái, thì giải thuật A* là không hoàn chỉnh.

Nếu không gian các trạng thái là vô hạn, thì giải thuật A* là không hoàn chỉnh.

3.3. Cài đặt thuật toán

3.3.1. Quy ước

- FRINGE : là hàng đợi ưu tiên chứa các node con sinh ra mà chưa được xét đến.
- CLOSED : tập các trạng thái đã xét, các trạng thái con được sinh ra sẽ được kiểm tra, nếu đã tồn tại trong CLOSED thì không thêm vào FRINGE nữa.
- RESULT: tập trạng thái từ trạng thái hiện tại cho đến đích.
- CHILD: tập node con của một node bất kỳ.

3.3.2. Cài đặt

- Bước 1

    RESULT = {}

    startNode.f = startNode.h = đánh giá heuristic từ startNode đến goalNode

    startNode.g = 0

    totalNodes = approvedNodes = 0

    FRINGE = {startNode}

- Bước 2

    While(FRINGE != {})

        Nếu quá thời gian tìm kiếm hoặc có yêu cầu dừng

            FRINGE.clear()

            CHILD.clear()

            CLOSED.clear()

        Tìm node có đánh giá f(n) nhỏ nhất trong FRINGE và gán cho currentNode

        Nếu currentNode là goalNode

            totalNodes = approvedNodes + FRINGE.size()

Thêm kết quả và RESULT

Tính thời gian tìm kiếm

FRINGE.clear()

CHILD.clear()

CLOSED.clear()

Dừng thuật toán

Thiết lập các node con của currentNode vào CHILD

Loại những node có trạng thái đã xét ra khỏi CHILD

Đưa các node con vào FRINGE

Đặt child của các node con là currentNode

Tính chi phí từ đầu đến node con = currentNode + 1

Tính hàm ước lượng của các node con đến goalNode

Tính hàm f = g + h

CHILD.clear()

approvedNodes++

3.3.3. Thiết kế thuật toán

- Các thư viện được sử dụng trong đoạn code và hàm Heuristic 1(Tổng số ô sai vị trí) :

```
import heapq
import math
import time

# Heuristic 1: Tổng số ô sai vị trí
def heuristic_1(state, goal):
    return sum(1 for i in range(len(state)) if state[i] != 0 and state[i] != goal[i])
```

- Hàm Heuristic 2(Tổng khoảng cách để đưa các ô về đúng vị trí) :

```python
# Heuristic 2: Tổng khoảng cách để đưa các ô về đúng vị trí
def heuristic_2(state, goal):
    n = int(len(state) ** 0.5)
    distance = 0
    for i in range(len(state)):
        if state[i] == 0:
            continue
        goal_index = goal.index(state[i])
        x1, y1 = divmod(i, n)
        x2, y2 = divmod(goal_index, n)
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance
```

- Hàm Heuristic 3(Tổng khoảng cách Euler của các ô với vị trí đích) :

```python
# Heuristic 3: Tổng khoảng cách Euler của các ô với vị trí đích
def heuristic_3(state, goal):
    n = int(len(state) ** 0.5)
    distance = 0
    for i in range(len(state)):
        if state[i] == 0:
            continue
        goal_index = goal.index(state[i])
        x1, y1 = divmod(i, n)
        x2, y2 = divmod(goal_index, n)
        distance += math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
    return distance
```

- Hàm Heuristic 4(Tổng số ô sai hàng và số ô sai cột) :

```python
# Heuristic 4: Tổng số ô sai hàng và số ô sai cột
def heuristic_4(state, goal):
    n = int(len(state) ** 0.5)
    h = 0
    for i in range(len(state)):
        if state[i] == 0:
            continue
        goal_index = goal.index(state[i])
        x1, y1 = divmod(i, n)
        x2, y2 = divmod(goal_index, n)
        if x1 != x2:
            h += 1
        if y1 != y2:
            h += 1
    return h
```

- Hàm heuristic 5(Tổng khoảng cách để đưa các ô về đúng vị trí + số ô xung đột tuyến tính) :

```python
# Heuristic 5: Tổng khoảng cách để đưa các ô về đúng vị trí + số ô xung đột tuyến tính
def heuristic_5(state, goal):
    n = int(len(state) ** 0.5)
    manhattan = heuristic_2(state, goal)
    linear_conflict = 0

    # Row conflicts
    for row in range(n):
        row_start = row * n
        for i in range(n):
            for j in range(i + 1, n):
                a = state[row_start + i]
                b = state[row_start + j]
                if a != 0 and b != 0:
                    a_goal = goal.index(a)
                    b_goal = goal.index(b)
                    if a_goal // n == row and b_goal // n == row:
                        if a_goal > b_goal:
                            linear_conflict += 1

    # Column conflicts
    for col in range(n):
        for i in range(n):
            for j in range(i + 1, n):
                a = state[i * n + col]
                b = state[j * n + col]
                if a != 0 and b != 0:
                    a_goal = goal.index(a)
                    b_goal = goal.index(b)
                    if a_goal % n == col and b_goal % n == col:
                        if a_goal > b_goal:
                            linear_conflict += 1

    return manhattan + 2 * linear_conflict
```

- Hàm Heuristic 6(Heuristic 5 + số ô không thể về đích) :

```python
# Heuristic 6: heuristic 5 + số ô không thể về đích
def heuristic_6(state, goal):
    n = int(len(state) ** 0.5)
    h = heuristic_5(state, goal)

    blocked_tiles = 0
    for i in range(len(state)):
        if state[i] == 0 or state[i] == goal[i]:
            continue
        x, y = divmod(i, n)
        surrounded = True
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < n:
                ni = nx * n + ny
                if state[ni] != goal[ni]:
                    surrounded = False
                    break
        if surrounded:
            blocked_tiles += 1

    return h + blocked_tiles
```

- Hàm lấy các heuristic ra để sử dụng theo type :

```python
# Lấy hàm heuristic theo type
def get_heuristic(state, goal, type):
    if type == 1:
        return heuristic_1(state, goal)
    elif type == 2:
        return heuristic_2(state, goal)
    elif type == 3:
        return heuristic_3(state, goal)
    elif type == 4:
        return heuristic_4(state, goal)
    elif type == 5:
        return heuristic_5(state, goal)
    elif type == 6:
        return heuristic_6(state, goal)
    else:
        raise ValueError("Invalid heuristic type")
```

- Hàm sinh trạng thái lân cận :

```python
# Sinh trạng thái lân cận
def get_neighbors(state):
    n = int(len(state) ** 0.5)
    zero = state.index(0)
    x, y = divmod(zero, n)
    neighbors = []

    directions = [(-1,0),(1,0),(0,-1),(0,1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < n:
            nz = nx * n + ny
            new_state = list(state)
            new_state[zero], new_state[nz] = new_state[nz], new_state[zero]
            neighbors.append(tuple(new_state))

    return neighbors
```

- Hàm kiểm tra trạng thái có giải được không :

```python
# Kiểm tra trạng thái có giải được không
def is_solvable(puzzle):
    n = int(len(puzzle) ** 0.5)
    inv_count = 0
    arr = [x for x in puzzle if x != 0]

    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] > arr[j]:
                inv_count += 1

    if n % 2 == 1:
        return inv_count % 2 == 0
    else:
        row_blank = puzzle.index(0) // n
        return (inv_count + row_blank) % 2 == 1
```

- Hàm chính giải thuật A* :

```python
# Giải thuật A*
def a_star(start, goal, heuristic_type=1):
    if not is_solvable(start):
        print("Trạng thái KHÔNG thể giải được!")
        return None

    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {start: 0}
    visited = set()

    while open_set:
        _, current = heapq.heappop(open_set)
        visited.add(current)

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        for neighbor in get_neighbors(current):
            tentative_g = g_score[current] + 1
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f = tentative_g + get_heuristic(neighbor, goal, heuristic_type)
                heapq.heappush(open_set, (f, neighbor))

    return None
```

- Hàm in trạng thái :

```python
# In trạng thái
def print_state(state):
    n = int(len(state) ** 0.5)
    for i in range(0, len(state), n):
        print(state[i:i+n])
    print("---")
```

- Hàm main để test trạng thái 3x3 :

```python
# MAIN
if __name__ == "__main__":
    start = (0, 4, 2,
             6, 5, 1,
             8, 7, 3)

    goal = (1, 2, 3,
            4, 5, 6,
            7, 8, 0)

    for h in range(1, 7):
        print(f"\n===== Heuristic {h} =====")
        t0 = time.time()
        path = a_star(start, goal, heuristic_type=h)
        t1 = time.time()

        if path:
            print(f"Tìm thấy lời giải trong {len(path)-1} bước, thời gian: {t1 - t0:.4f} giây")
            for step in path:
                print_state(step)
        else:
            print("Không tìm thấy lời giải.")
```

3.4. Cơ sở của các hàm heuristic

3.4.1. Hàm heuristic 1 – Tổng số ô sai vị trí

- Ý tưởng : Đây là một heuristic rất đơn giản và trực quan, đếm tổng số hiện đang không nằm ở vị trí đúng so với trạng thái mục tiêu. Mỗi ô sai vị trí sẽ cần ít nhất một bước di chuyển để về đúng chỗ, nên tổng số ô sai vị trí là một ước lượng thô về số bước cần thiết để hoàn thành trò chơi.
- Cài đặt :

```python
import heapq
import math
import time

# Heuristic 1: Tổng số ô sai vị trí
def heuristic_1(state, goal):
    return sum(1 for i in range(len(state)) if state[i] != 0 and state[i] != goal[i])
```

- Chứng minh : Xét các ô bất kỳ, nếu các ô đó sai vị trí thì ta có tổng số ô sai vị trí luôn nhỏ hơn hoặc bằng giá trị h*(giá trị thực tế để đưa ô đó về đúng vị

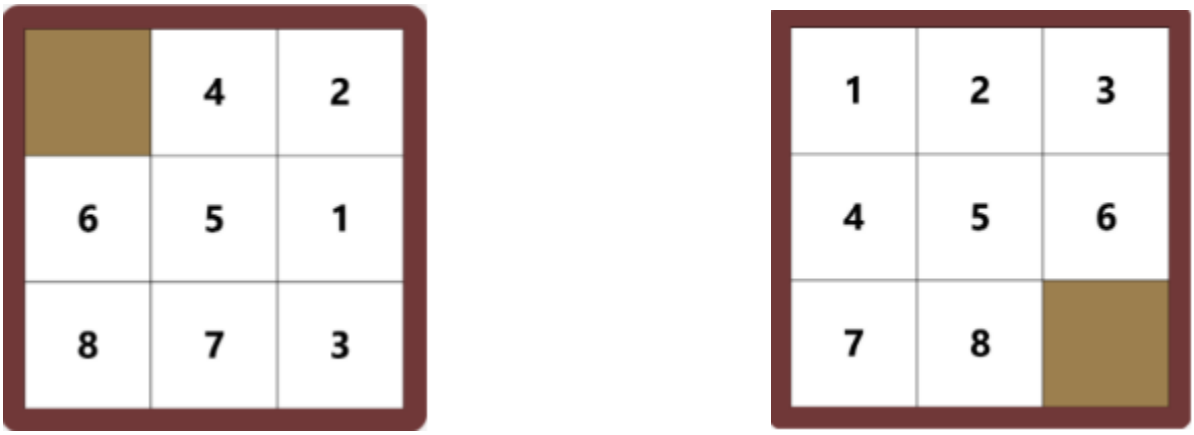
trí) vì vậy hàm h1 sẽ luôn nhỏ hơn h* nên đây là một heuristic chấp nhận được.

- Ví dụ :

Xét trạng thái như hình bên dễ dàng thấy được h1 = 7

Chi phí thực tế h* = 22

Thời gian thực thi : 0.1615 giây



Hình 3. 1 : Kết quả hàm Heuristic 1

3.4.2. Hàm heuristic 2 – Tổng khoảng cách để đưa các ô về đúng vị trí

- Ý tưởng : Tính tổng khoảng cách Manhattan của tất cả các ô đến vị trí đích của chúng, nghĩa là tính tổng số bước đi dọc theo hàng và cột mà mỗi ô cần di chuyển để về đúng vị trí. Khoảng cách Manhattan là tổng giá trị tuyệt đối hiệu số hàng và cột của vị trí hiện tại và vị trí đích.
- Cài đặt :

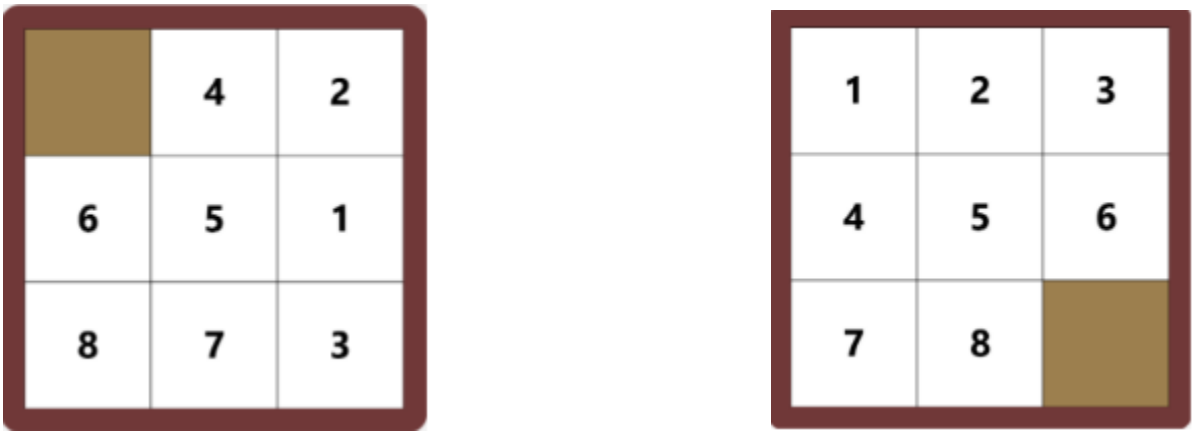
```python
# Heuristic 2: Tổng khoảng cách để đưa các ô về đúng vị trí
def heuristic_2(state, goal):
    n = int(len(state) ** 0.5)
    distance = 0
    for i in range(len(state)):
        if state[i] == 0:
            continue
        goal_index = goal.index(state[i])
        x1, y1 = divmod(i, n)
        x2, y2 = divmod(goal_index, n)
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance
```

- Chứng minh : Xét một ô vuông bất kì đang nằm sai vị trí ta tính được h2 tại đó bằng khoảng cách ngắn nhất để đưa ô đó về đúng vị trí. Trong thực tế chi phí để đưa 1 ô về đúng vị trí sẽ không thể luôn là khoảng cách ngắn nhất, vì vây h2 luôn nhỏ hơn hoặc bằng h*. h2 là một heuristic chấp nhận được.
- Ví dụ :

Xét trạng thái như hình bên dễ dàng thấy được h2 = 12

Chi phí thực tế h* = 22

Thời gian thực thi : 0.0350 giây



Hình 3. 2 : Kết quả hàm Heuristic 2

3.4.3. Hàm heuristic 3 – Tổng khoảng cách Euler của các ô với vị trí đích

- Ý tưởng : Tính tổng khoảng cách Euclid (khoảng cách thẳng giữa hai điểm trên mặt phẳng) của các ô đến vị trí đích. Đây là khoảng cách đường chim bay.
- Cài đặt :

```python
# Heuristic 3: Tổng khoảng cách Euler của các ô với vị trí đích
def heuristic_3(state, goal):
    n = int(len(state) ** 0.5)
    distance = 0
    for i in range(len(state)):
        if state[i] == 0:
            continue
        goal_index = goal.index(state[i])
        x1, y1 = divmod(i, n)
        x2, y2 = divmod(goal_index, n)
        distance += math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
    return distance
```

- Chứng minh : Xét một ô vuông bất kì đang nằm sai vị trí ta tính được h3 tại đó bằng đường chéo nối nó với đích, sau đó ta chỉ lấy phần nguyên chính vì vậy khoảng cách này luôn luôn nhỏ hơn hoặc bằng khoảng cách ngắn nhất để đưa ô đó về đúng vị trí. Từ chứng minh heuristic 2 có thể suy ra h3 cũng là một heuristic chấp nhận được.
- Ví dụ :

Xét trạng thái như hình bên dễ dàng thấy được h3 = 10

Chi phí thực tế h* = 22

Thời gian thực thi : 0.0574 giây

Hình 3. 3 : Kết quả hàm Heuristic 3

3.4.4. Hàm heuristic 4 – Tổng số ô sai hàng và số ô sai cột

- Ý tưởng : Thay vì tính khoảng cách Manhattan chính xác, heuristic này chỉ kiểm tra xem ô hiện tại có nằm đúng hàng (row) và cột (column) hay không. Mỗi ô sai hàng cộng 1 điểm, mỗi ô sai cột cộng thêm 1 điểm. Tổng cộng là số lần vị trí của ô lệch hàng và cột so với đích.
- Cài đặt :

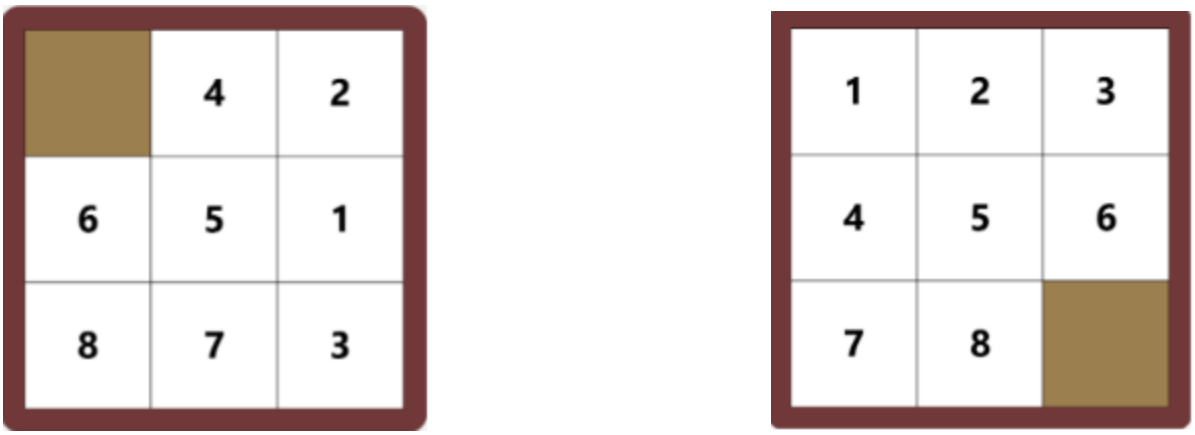
```
# Heuristic 4: Tổng số ô sai hàng và số ô sai cột
def heuristic_4(state, goal):
    n = int(len(state) ** 0.5)
    h = 0
    for i in range(len(state)):
        if state[i] == 0:
            continue
        goal_index = goal.index(state[i])
        x1, y1 = divmod(i, n)
        x2, y2 = divmod(goal_index, n)
        if x1 != x2:
            h += 1
        if y1 != y2:
            h += 1
    return h
```

- Chứng minh : Xét một ô nằm sai cả hàng và cột ta sẽ có h4 tại đó bằng 2 mà với một ô sai cả hàng và cột chi phí để đưa ô đó về đúng vị trí luôn lớn hơn hoặc bằng 2 tương tự với ô sai hàng hoặc cột. Vì vậy h4 là một heuristic chấp nhận được.
- Ví dụ :

Xét trạng thái như hình bên dễ dàng thấy được h4 = 9

Chi phí thực tế h* = 22

Thời gian thực thi : 0.0509 giây



Hình 3. 4 : Kết quả hàm Heuristic 4

3.4.5. Hàm heuristic 5 – Tổng khoảng cách để đưa các ô về đúng vị trí + số ô xung đột tuyến tính

- Ý tưởng : Kết hợp heuristic Manhattan với khái niệm “linear conflict” (xung đột tuyến tính). Linear conflict xảy ra khi hai ô cùng nằm trên một hàng hoặc một cột, nhưng vị trí mục tiêu của chúng xung đột với thứ tự hiện tại (ví dụ ô A phải đứng trước ô B trong trạng thái mục tiêu nhưng hiện tại lại ngược lại). Khi đó ít nhất một ô phải di chuyển vượt qua ô kia, tạo ra xung đột làm tăng chi phí.
- Cài đặt :

```python
# Heuristic 5: Tổng khoảng cách để đưa các ô về đúng vị trí + số ô xung đột tuyến tính
def heuristic_5(state, goal):
    n = int(len(state) ** 0.5)
    manhattan = heuristic_2(state, goal)
    linear_conflict = 0

    # Row conflicts
    for row in range(n):
        row_start = row * n
        for i in range(n):
            for j in range(i + 1, n):
                a = state[row_start + i]
                b = state[row_start + j]
                if a != 0 and b != 0:
                    a_goal = goal.index(a)
                    b_goal = goal.index(b)
                    if a_goal // n == row and b_goal // n == row:
                        if a_goal > b_goal:
                            linear_conflict += 1

    # Column conflicts
    for col in range(n):
        for i in range(n):
            for j in range(i + 1, n):
                a = state[i * n + col]
                b = state[j * n + col]
                if a != 0 and b != 0:
                    a_goal = goal.index(a)
                    b_goal = goal.index(b)
                    if a_goal % n == col and b_goal % n == col:
                        if a_goal > b_goal:
                            linear_conflict += 1

    return manhattan + 2 * linear_conflict
```

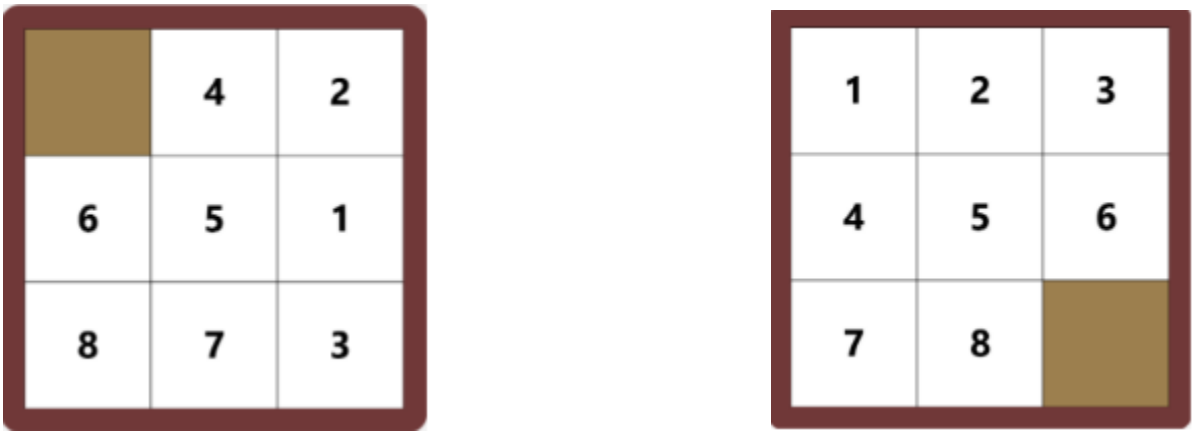
- Chứng minh : Với khoảng cách ngắn nhất để đưa một ô về đúng vị trí đã chứng minh ở h2 ta cộng thêm số ô xung đột tuyến tính. Hai ô được gọi là xung đột tuyến tính(Linear conflict) nếu 2 ô đó nằm đúng hàng hoặc cột nhưng ô này bị chặn bởi ô kia (giống như ô 7 và 8 ở ví dụ bên dưới). Để đưa cả 2 ô như vậy về đúng vị trí cần chuyển một ô sang hàng hoặc cột khác rồi mới đưa được 1 ô về đúng vị trí, sau đó đưa ô còn lại về đúng hàng và đúng vị trí. Việc làm trên cần 4 bước để hoàn thành vì vậy chi phí sẽ luôn lớn hơn hoặc bằng 4. Vì vậy h5 là một heuristic chấp nhận được.

- Ví dụ :

Xét trạng thái như hình bên dễ dàng thấy được h5 = 16

Chi phí thực tế h* = 22

Thời gian thực thi : 0.0245 giây



Hình 3. 5 : Kết quả hàm Heuristic 5

3.4.6. Hàm heuristic 6 – heuristic 5 + số ô không thể về đích

- Ý tưởng : Dựa trên heuristic 5, thêm vào phần đếm "blocked tiles" – những ô bị "chặn" bởi các ô khác sai vị trí xung quanh, tức là không thể di chuyển trực tiếp hoặc dễ dàng về vị trí đích. Cụ thể, nếu một ô sai vị trí mà xung quanh nó đều là các ô cũng đang sai vị trí (theo trạng thái mục tiêu), thì được xem là bị chặn và tăng thêm chi phí heuristic.
- Cài đặt :

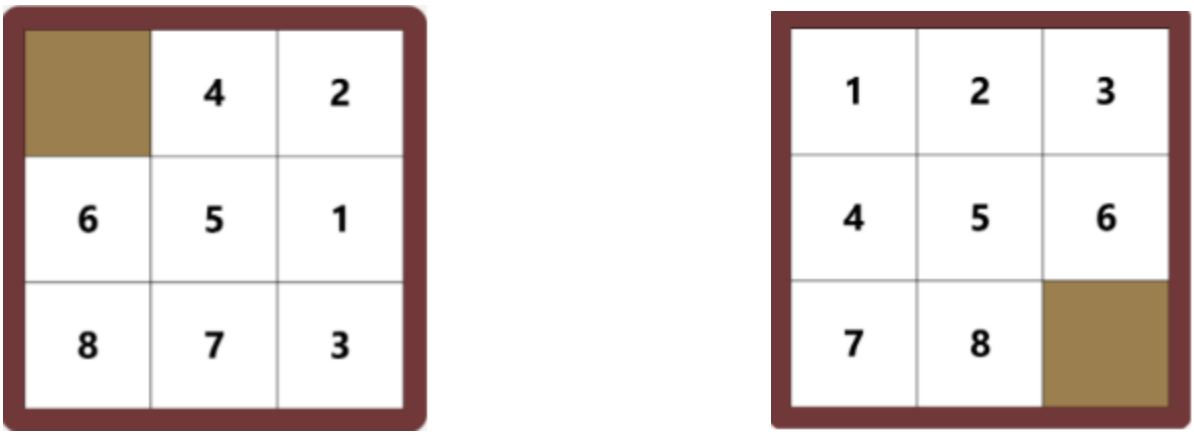
```python
# Heuristic 6: heuristic 5 + số ô không thể về đích
def heuristic_6(state, goal):
    n = int(len(state) ** 0.5)
    h = heuristic_5(state, goal)

    blocked_tiles = 0
    for i in range(len(state)):
        if state[i] == 0 or state[i] == goal[i]:
            continue
        x, y = divmod(i, n)
        surrounded = True
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < n:
                ni = nx * n + ny
                if state[ni] != goal[ni]:
                    surrounded = False
                    break
        if surrounded:
            blocked_tiles += 1

    return h + blocked_tiles
```

- Chứng minh : Từ heuristic 5 bên trên, nhóm chúng em cộng thêm số ô không thể về đích. Các ô không thể về đích là các ô mà những ô xung quanh nó đều đúng vị trí đích, chính vì vậy để đưa ô này về trạng thái đích cần di chuyển ít nhất 2 ô khác ra khỏi vị trí đích nên chi phí để làm việc trên chắc chắn lớn hơn 1. Đây là một heuristic chấp nhận được.
- Ví dụ :

Xét trạng thái như hình bên dễ dàng thấy được h6 = 16

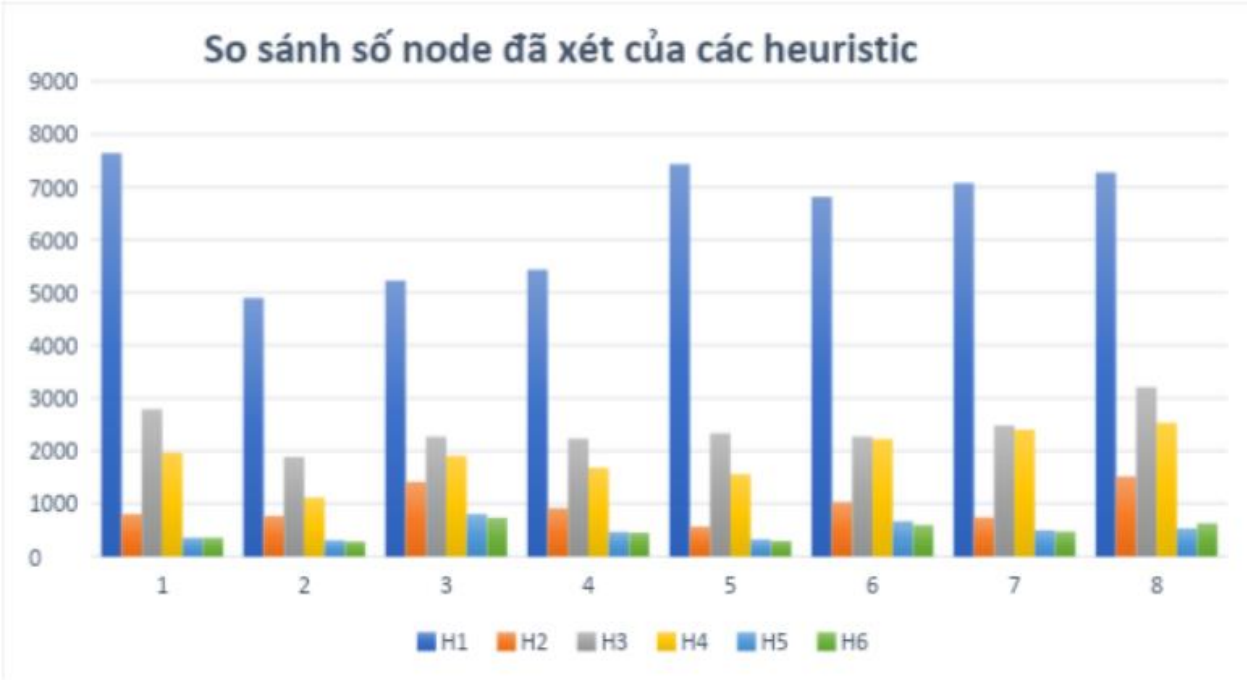
Chi phí thực tế h* = 22

Thời gian thực thi : 0.0281 giây

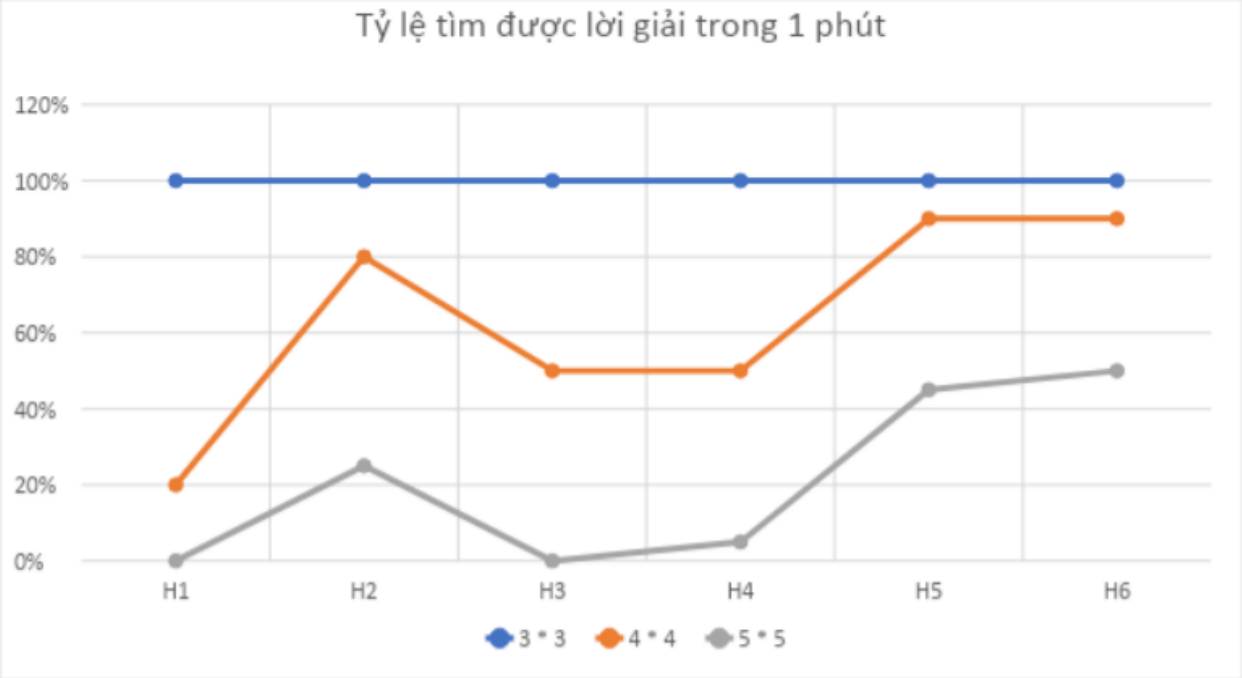


Hình 3. 6 : Kết quả hàm Heuristic 6

3.4.7. So sánh các heuristic



Hình 3. 7 : Hình ảnh so sánh số node đã xét của các heuristic



Hình 3. 8 : Hình ảnh minh họa tỷ lệ tìm được lời giải

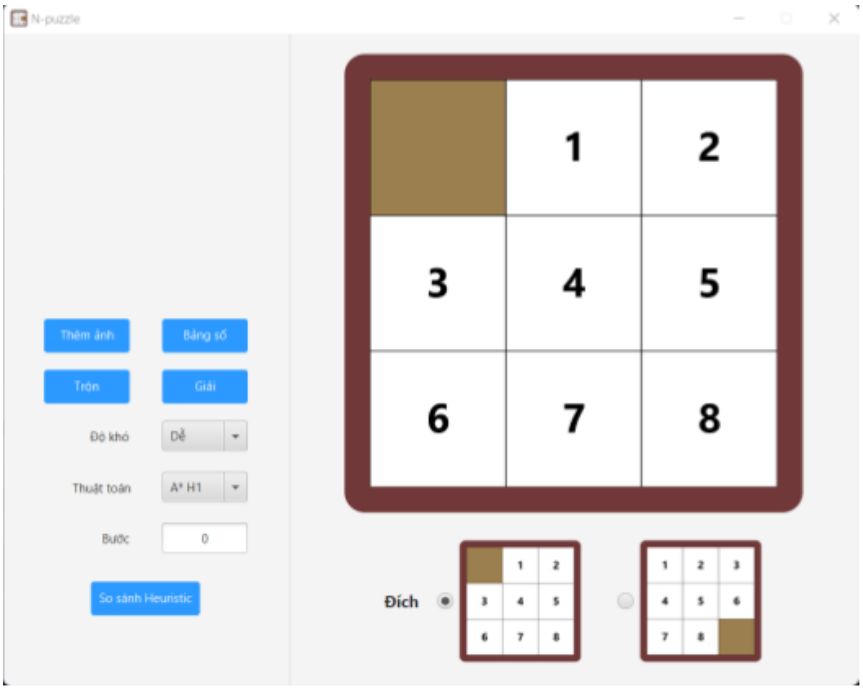
Từ những heuristic kể trên với nhiều bộ dữ liệu khác nhau để thu được kết quả. Sau khi thống kê lại bọn em thu được kết quả như sau:

H1 < H3 < H4 < H2 < H5 < H6

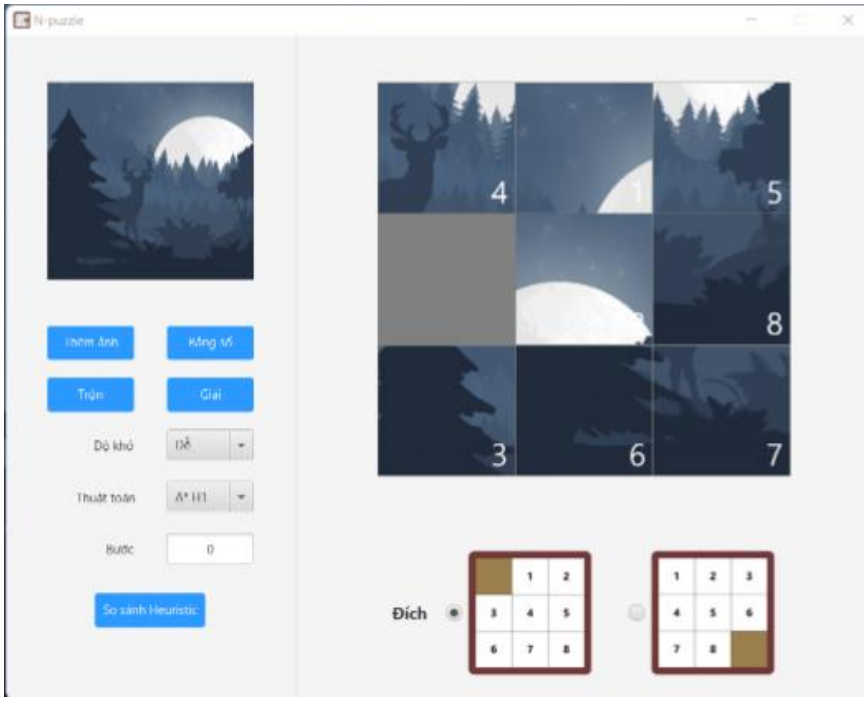
CHƯƠNG 4: TRÒ CHƠI

4.1 Giao diện:

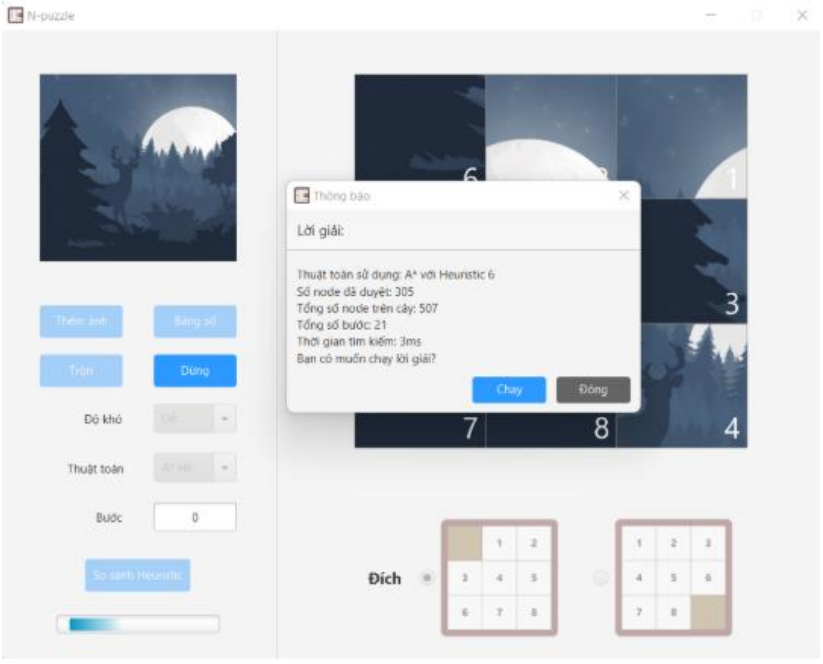
- Giao diện khi khởi chạy trò chơi



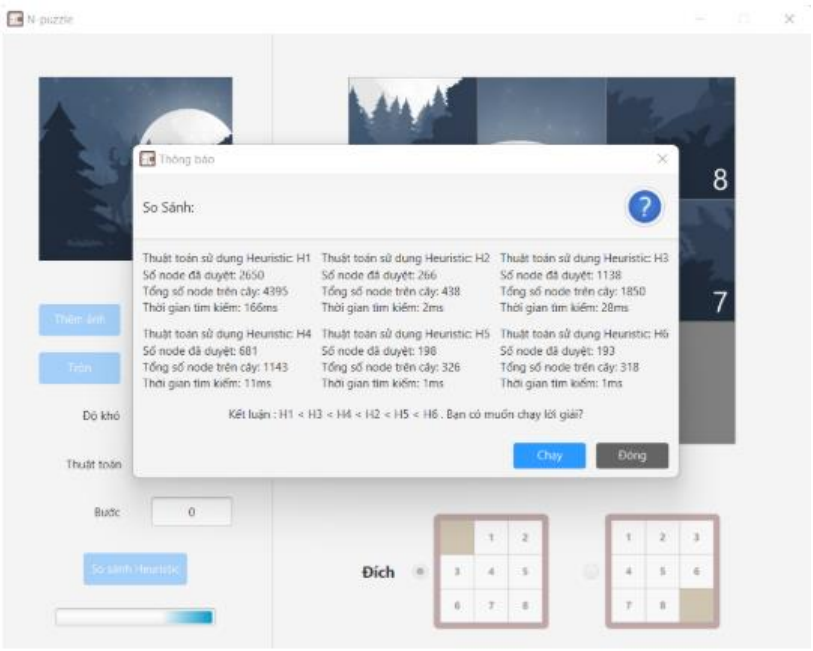
- Giao diện khi thêm ảnh



- Giao diện khi tìm được lời giải



- Giao diện so sánh heuristic


N-puzzle
Thông báo
So Sánh:
Thuật toán sử dụng Heuristic: H1
Số node đã duyệt: 2650
Tổng số node trên cây: 4395
Thời gian tìm kiếm: 166ms
Thuật toán sử dụng Heuristic: H2
Số node đã duyệt: 266
Tổng số node trên cây: 438
Thời gian tìm kiếm: 2ms
Thuật toán sử dụng Heuristic: H3
Số node đã duyệt: 1138
Tổng số node trên cây: 1850
Thời gian tìm kiếm: 28ms
Thuật toán sử dụng Heuristic: H4
Số node đã duyệt: 681
Tổng số node trên cây: 1143
Thời gian tìm kiếm: 11ms
Thuật toán sử dụng Heuristic: H5
Số node đã duyệt: 198
Tổng số node trên cây: 326
Thời gian tìm kiếm: 1ms
Thuật toán sử dụng Heuristic: H6
Số node đã duyệt: 193
Tổng số node trên cây: 318
Thời gian tìm kiếm: 1ms
Kết luận : H1 < H3 < H4 < H2 < H5 < H6 . Bạn có muốn chạy lời giải?
Chạy
Đóng
Thêm ảnh
Trộn
Độ khó
Thuật toán
Bước
0
So sánh Heuristic
Đích
8
7

KẾT LUẬN

Việc xây dựng và giải quyết bài toán "Ứng dụng các thuật toán tìm kiếm để giải bài toán ghép tranh (N-Puzzle)" được chúng em thực hiện dựa trên những kiến thức được học trong môn "Trí tuệ nhân tạo" do cô –TS. Mai Thanh Hồng giảng dạy. Trong quá trình học tập cũng như nghiên cứu và phát triển đề tài, nhóm chúng em đã học được rất nhiều về cách cài đặt và sử dụng các hàm heuristic để xử lý thuật toán A*, cũng qua đó giúp bọn em hiểu rõ về thuật toán A* hơn – một thuật toán tìm kiếm thông minh để có thể sử dụng trong việc học tập cũng như là công việc sau này.

Việc hoàn thành bài tập lớn này không chỉ là sự nỗ lực và cố gắng của cả nhóm mà còn là sự giảng dạy tận tình của cô – TS. Mai Thanh Hồng, chúng em cảm ơn cô vì những kiến thức mà cô đã cung cấp về Trí tuệ nhân tạo để chúng em có thể hoàn thành bài tập của mình.

TÀI LIỆU THAM KHẢO

[1]. Nguyễn Phương Nga (ch.b), Trần Hùng Cường, *Giáo trình trí tuệ nhân tạo*, Nhà xuất bản Thống kê, Hà Nội, 2021.

[2]. Giải thuật tìm kiếm A* - https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_t%C3%ACm_ki%E1%BA%BFm_A*

[3]. Solving the 8-Puzzle using A* Heuristic Search.

[4]. Implementation and Analysis of Iterative MapReduce Based Heuristic Algorithm for Solving N-Puzzle.