# GIẢI BÀI TOÁN 8 PUZZLE TRONG TRƯỜNG HỢP CÁC Ô BỊ CHE CHỈ CÓ MỘT Ô ĐỂ HIỂN THỊ GIÁ TRỊ

**Huỳnh Thanh Tâm­­1 18110194**

**Hồ Duy Tân1 18110195**

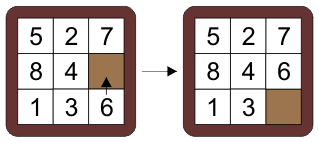
*1Trường Đại học Sư phạm Kỹ thuật TP.HCM*

## TÓM TẮT

Puzzle (còn gọi là Gem Puzzle, Ô vuông kỳ bí) là một câu đố trượt bao gồm một khung các ô vuông được đánh số theo thứ tự ngẫu nhiên với một ô bị thiếu. Puzzle có thể tồn tại nhiều kích cỡ khác nhau, phổ biến nhất là 8-puzzle với số ô là 3x3 hay 15-puzzle với kích cỡ 4x4. Mục tiêu của Puzzle là đặt các viên gạch theo thứ tự bằng các thực hiện các động tác trượt sử dụng khoảng trống cho đến khi các viên gạch được đặt theo đúng thứ tự. Trong bài viết và đồ án chúng em sử dụng 8-puzzle để thực hiện bài toán. Nhiều giải thuật có thể giải bài toán chính xác, tuy nhiên có nhiều trường hợp đặc biệt mà bài toán không thể tìm ra lời giải từ trạng thái đầu tới đích đến mong muốn, vì thế chúng em giới hạn đích đến trong 2 trường hợp để thuật giải luôn luôn tìm ra câu trả lời. Đầu vào Start của bài toán được sinh tự động và bị che khuất chỉ hiện giá trị ô chính giữa. Sau đó tiến hành khai phá các ô trống và dựa vào trạng thái Start đã được mở cuối cùng để quyết định đích đến, sau đó tiến hành giải bài toán. Để có thể tìm ra thuật giải nào tốt hơn (thời gian thực thi và sử dụng bộ nhớ), chúng em sử dụng 2 giải thuật Breadth-First Search và A\* để giải bài toán sau đó so sánh dựa trên thời gian thực thi của mỗi thuật toán.

**Từ khóa:** 8 puzzle; Breadth-First Search; A\*; giải thuật tìm kiếm

## ĐẶT VẤN ĐỀ

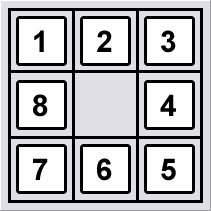


**Hình 1.** Bài toán 8 puzzle

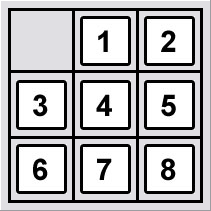
Puzzle 8 là một vấn đề cổ điển để mô hình hóa các thuật toán liên quan đến phương pháp phỏng đoán. Các phương pháp phỏng đoán thường được sử dụng cho vấn đề này bao gồm đếm số lượng gạch bị đặt sai vị trí và tìm tổng khoảng cách giữa mỗi khối và vị trí của nó trong cấu hình mục tiêu. Trong những năm gần đây, trí tuệ nhân tạo (AI) được ứng dụng ngày càng nhiều bằng việc áp dụng vào cuộc sống hằng ngày. Việc tìm hiểu bài toán 8 puzzle là một trong những ứng dụng của AI, từ trạng thái đầu tìm lời giải cho đến khi đạt được trạng thái đích đã cho trước. Vấn đề được đặt ra là tất cả các ô bị che chỉ thấy được ô tâm vì thế làm cách nào để giải bài toán nhanh nhất với điều kiện, ít tốn chi phí nhất mà vẫn chính xác. Có 1 thay đổi nhỏ ở phần đầu vào là khi đã có Start state, bài toán vẫn chưa được giải vì Start state đã bị che. Tiến hành mở Start state sau đó mới thực hiện bài toán. Sử dụng 2 thuật toán tìm kiếm là BFS và A\* và so sánh để xác định thuật toán nào là tối ưu, nhanh, tiết kiệm hơn.

## PHƯƠNG PHÁP

### Xác định đích đến dựa trên trạng thái đầu



**Hình 2.** Goal A

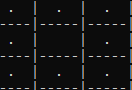
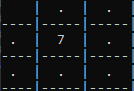
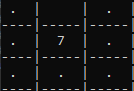


**Hình 3.** Goal B

Trò chơi có trạng thái đầu (Start state) và trạng thái đích (Goal state). Trạng thái đích mà nhóm sử dụng chỉ có 2 loại (A hoặc B). Trạng thái đầu được sinh ngẫu nhiên. Dựa trên trạng thái đầu, bắt đầu từ vị trí I = Start[0][0], gọi N là tổng các số có giá trị bé hơn I (khác 0) từ sau nó đến cuối ma trận, sau đó chuyển I đến ô tiếp theo, thực hiện N tương tự cho đến khi I là ô sát cuối (vì ô cuối cùng đằng sau nó không còn ô nào). Sau khi đã có tổng N, ta tiến hành xác định trạng thái đích cho bài toán bằng cách xét N là lẻ (N % 2 !=0) hay chẵn (N % 2 == 0). Nếu N lẻ thì trạng thái đích được xác định là Goal A (hình 2) và N chẵn thì là Goal B (hình 3). Việc xác định Goal sau khi có Start nhằm giảm thiểu tình trạng thuật toán không thể tìm ra lời giải cho những Goal có thứ tự các giá trị được sắp xếp ngẫu nhiên.

### Ẩn giá trị và khai phá các ô trống

Đầu vào Start của bài toán được sinh ngẫu nhiên từ 0 đến 8. Sau đó tiến hành che giá trị các ô chỉ để chừa ô chính giữa hiện giá trị. Điểm mấu chốt của việc mở ô là chỉ có thể di chuyển ô trống và mọi ô bị che chỉ được mở khi đi qua ô chính giữa. Chúng em chia quá trình mở ô thành 3 phần theo thứ tự: đưa ô trống vào giữa, mở các ô cạnh còn bị ẩn, mở các ô góc. Phần đầu tiên tiến hành tìm vị trí của ô trống trên ma trận Start đã tạo, nếu ô trống đã nằm ngay vị trí Start[1][1] rồi thì kết thúc phần 1, ngược lại nếu ô trống nằm trên 1 trong những ô cạnh thì tráo vị trí ô trống với ô chính giữa, ô cạnh lúc này mang giá trị của ô chính giữa lúc đầu, nếu ô trống nằm ở 1 trong các ô góc thì đổi vị trí của ô góc đó với ô cạnh gần nó, sau đó đổi vị trí ô trống với ô chính giữa. Phần này có số bước thực hiện tối đa là 2 (cho trường hợp số trống nằm ở 1 trong các ô góc), và số bước ít nhất là 0 (cho trường hợp số trống nằm ngay chính giữa). Mỗi lần tráo ô trống với 1 ô khác được tính là 1 bước thực hiện. Ở phần 2 có số bước thực hiện tối đa là 8 (trường hợp ô trống nằm chính giữa ở phần 1) và tối thiểu là 6 (2 trường hợp còn lại ở phần 1). Trường hợp ô trống nằm chính giữa thì 4 ô cạnh đều chưa được mở, việc mở 1 ô cạnh tốn 2 bước bao gồm tráo ô trống với ô đó rồi đưa ô trống vào giữa lại và ô trống đã được mở, vì thế tổng là 8 bước cho 4 ô. Trường hợp còn lại đã có 1 ô được mở sẵn do chuyển ô trống vào giữa từ phần 1 nên số ô cạnh chưa mở chỉ còn 3, tổng số bước là 6. Sau khi hoàn thành mở các ô cạnh, chúng em dùng hàm find\_Min() để tìm ô cạnh có giá trị nhỏ nhất (min). Đến với phần 3, đổi ô trống với ô min vừa tìm được; sau đó đổi vị trí ô trống với 1 trong 2 ô góc gần nó, rồi chuyển vị trí ô trống vào giữa lại mà không làm thay đổi vị trí ô góc vừa tráo, việc này tốn 2 bước thực hiện, sau đó mở ô trống đó nhưng không đưa ô trống vào giữa, mà thực hiện mở tiếp tục cho ô góc kia, tổng số bước cho phân đoạn này là 8. Lúc này ô trống đang ở ô min, thực hiện 2 bước đưa ô trống sang ô cạnh đối diện, và thực hiện tiếp tục phân đoạn trên, cuối cùng đưa ô trống vào lại chính giữa. Vậy tổng số bước cho phần này = 1 + 8 + 2 + 8 + 1 = 20 bước. Quá trình mở ô có tổng số bước tối đa là 28 cho trường hợp ô trống ban đầu ở chính giữa hoặc ở góc, tối thiểu là 27 cho trường hợp ô trống nằm ở cạnh.

**Hình 4.** Ô trống ở giữa **Hình 5.** Ô trống ở góc **Hình 6.** Ô trống ở cạnh

### Thuật toán tìm kiếm

* Thuật toán A\*

Procedure ASTAR( root, Goal)

Q,V := queue

Enqueue(root,Q)

While( Q is not null )

temp:= dequeue(Q)

if temp is Goal then

return temp

else

for all Actions in temp.legal\_Action() do

child:=temp.Action(i)

if(child is not in V)

child.path\_Cost += heuristic(child,Goal)

FIFO\_Insert(child,queue)

enqueue(temp,V)

Đây có thể được xem như là một mở rộng của thuật toán Dijkstra với các bước cài đặt tương đồng nhưng có thêm thông tin lượng giá (heuristic) ở mỗi bước tìm kiếm. 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](https://vi.wikipedia.org/w/index.php?title=H%C3%A0ng_%C4%91%E1%BB%A3i_%C6%B0u_ti%C3%AAn&action=edit&redlink=1) (*priority queue*). Thứ tự ưu tiên gán cho một đường đi x được quyết định bởi hàm: f(x) = h(x) + g(x) (1). Trong đó: g(x) là chi phí của đường đi cho đến thời điểm hiện tại, tức là tổng trọng số của tất cả các cạnh đã đi qua, h(x) là hàm heuristic đánh giá chi phí thấp nhất đi từ điểm hiện tại đến đích. Dựa vào path\_Cost được tính bằng hàm heuristic, thuật toán lưu các đỉnh sẽ xét trong 1 tập ưu tiên theo thứ tự giá trị path\_Cost từ bé đến lớn, tức là ở mỗi bước sinh đỉnh con, thuật toán sẽ tính toán chi phí dựa trên hàm heuristic để gán cho đỉnh con, sau đó cho đỉnh con vào hàng đợi ưu tiên giá trị path\_Cost bé hơn. Thuật toán kết thúc khi hàng đợi chờ xét không còn phần tử nào hoặc đã tìm ra lời giải. Thuật toán giải quyết được vấn đề xét các đỉnh dư thừa, qua đó tăng tốc độ giải quyết của bài toán.

* Thuật toán Breadth-First Search (BFS)

Procedure BFS( root, Goal)

Q,V := queue

Enqueue(root,Q)

While( Q is not null )

temp:= dequeue(Q)

if temp is Goal then

return temp

else

for all Actions in temp.legal\_Action() do

child:=temp.Action(i)

if(child is not in V)

enqueue(child,queue)

enqueue(temp,V)

Thuật toán BFS bắt đầu từ đỉnh gốc và lần lượt nhìn các đỉnh kề với đỉnh gốc. Sau đó, với mỗi đỉnh trong số đó, thuật toán lại lần lượt sinh đỉnh con dựa trên vị trí ô trống để đỉnh con có Action hợp lệ và tránh lặp lại những đỉnh đã xét trước đó. Đỉnh con sau khi được tạo ra sẽ được đưa vào hàng đợi chờ xét. Do có tính chất là thuật toán vét cạn nên nếu có lời giải thì chắc chắn sẽ tìm ra, thuật toán phải dò đến từng đỉnh được sinh. Số đỉnh có độ sâu n sẽ nhiều hơn số đỉnh có độ sâu n-1 nên việc thuật toán phải xét đến những đỉnh dư thừa là không thể tránh khỏi.

## KẾT QUẢ

Sau một vài kết quả chạy thử, có thể kết luận rằng: Thuật toán A\* chạy tốt hơn thuật toán BFS cả về thời gian thực thi và sử dụng bộ nhớ. Ví dụ với Start = { 6,0,2,1,7,3,4,8,5 } và Goal = { 0,1,2,3,4,5,6,7,8 } có 17 bước giải thì thuật toán BFS mất 2,92s để tìm, trong khi đó thuật toán A\* chỉ mất 0,18s để tìm ra. Hoặc với ví dụ đơn giản hơn Start = { 1,4,2,3,0,5,6,7,8 } cũng là Goal trên có 2 bước giải thì A\* giải trong 0,011s còn BFS là 0,014s. Kinh điển hơn là bài toán Start = { 1,2,3,4,5,6,7,8,0 } và Goal như trên, với 22 bước giải thì BFS tìm ra trong 240,87s, tức xấp xỉ 4 phút, còn A\* chỉ cần 0,28s. Cũng có nghĩa trong trường hợp này A\* chạy nhanh hơn BFS đến 860.25 lần. Lấy Start = { 6,7,2,0,1,3,4,8,5 } và Goal = { 0,1,2,3,4,5,6,7,8 } chúng em có bảng sau:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Số bước** | 0 đến 15 | 16 | 17 | 18 | 19 | 20 | ... |
| **Thời gian A\* (s)** | < 0.3 | ~0.24 | ~0.29 | ~0.28 | ~0.25 | ~0.21 | ... |
| **Thời gian BFS (s)** | < 1 | ~1.8 | ~3.1 | ~15.8 | ~37.5 | ~53.45 | ... |

**Bảng 1.** So sánh thời gian thực thi giữa 2 thuật toán BFS và A\*

Sau thí nghiệm trên chúng em rút ra được 1 số kết luận về thời gian thực thi như sau: Thuật toán A\* có thời gian thực thi thường không tới 1s, trong khi đó, thuật toán Breadth-First Search có thời gian thực thi tăng dần theo cấp số nhân số bước giải và công bội này trong giá trị khoảng từ 1 đến 3.

## KẾT LUẬN

Sau khi thực hiện đồ án, chúng em kết luận rằng thuật toán A\* cho ra kết quả chính xác, nhanh chóng, ít tốn kém bộ nhớ hơn BFS, giải quyết được vấn đề đặt ra ở đầu bài. Thuật toán BFS sử dụng kỹ thuật loan nên số đỉnh phải xét trước khi đến được đích tăng theo chiều sâu của đỉnh. Trong trường hợp xấu nhất thì đỉnh đích chính là đỉnh cuối cùng được xét và chiều sâu lớn hơn 20 thì thời gian thực thi đã có thể lên đến 1 hoặc 2 giờ đồng hồ. Thuật toán A\*, sử dụng hàm heuristic đánh giá chi phí thấp nhất từ đỉnh đang xét đến đích, nên loại bỏ được rất nhiều những đỉnh dư thừa. Vì thế thời gian giải của A\* nhanh hơn BFS rất nhiều khi BFS phải vét cạn, tức là xét toàn bộ trường hợp. Trong phần khai phá trạng thái Start, chương trình sẽ chạy nhiều bước nhất nếu ô trống ban đầu được sinh ra ở góc hoặc ở chính giữa ma trận, ngược lại sẽ chạy ít bước nhất khi ô trốn được sinh ra ở cạnh. Và sau khi kết thúc quá trình mở ô, ô trống luôn trở lại ô chính giữa. Việc ứng dụng thuật toán có hàm tự đánh giá heuristic như A\* giúp cho việc giải trò chơi cổ điển này trở nên nhanh chóng và vẫn giữ được tính chính xác.

# TÀI LIỆU THAM KHẢO

[1]: Trung Tín, “Bài toán 8 puzzle”, 2014.

Available: <https://eritvn.wordpress.com/2014/10/13/bai-toan-8-puzzle-hay-con-goi-la-bai-toan-ta-canh/>

[2]: Kênh Youtube Jinyue Han: Solving 8 puzzle with A\* search

[3]: YinYang,”[Algorithm – Phân tích và giải bài toán n-puzzle](https://yinyangit.wordpress.com/2010/12/16/algorithm-phan-tich-va-gi%e1%ba%a3i-bai-toan-n-puzzle/)”, 2010

Available:<https://yinyangit.wordpress.com/2010/12/16/algorithm-phan-tich-va-gi%e1%ba%a3i-bai-toan-n-puzzle/>

[4]: [Ajinkya Sonawane](https://blog.goodaudience.com/@sonawaneajinks?source=post_page-----7b509c331288----------------------), “Solving 8-Puzzle using A\* algorithm”, 2018.

Available: <https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288>.

[5]: Wikipedia,”Tìm kiếm theo chiều rộng”, 2020

Available: <https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_r%E1%BB%99ng>