**CHƯƠNG TRÌNH AI CHO MÁY CHƠI GAME 8-PUZZLE VỚI NGƯỜI**

**Phạm Nhật Bình1, Đỗ Đức Anh1**

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

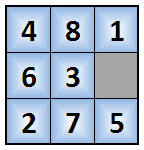
**TÓM TẮT**

Bài toán n-puzzle có lẽ rất quen thuộc với chúng ta cũng như những người mới bắt đầu tiếp cận với bộ môn trí tuệ nhân tạo. Nó được biết đến với nhiều phiên bản và tên gọi khác nhau như 8-puzzle, 15-puzzle, Gem puzzle, Boss puzzle, Game of Fifteen, Mystic Square,… Ở mức độ đơn giản có thể nói đến 8-puzzle. Bài toán gồm một bảng 3x3 với các ô số được đánh từ 1->8 và một ô trống. Ở trạng thái bắt đầu, các ô được sắp đặt ngẫu nhiên, và nhiệm vụ của người giải là tìm cách đưa chúng về đúng thứ tự kiểu lợp ngói.

Từ khóa: puzzle; trí tuệ nhân tạo ; kiểu lợp ngói; trạng thái ban đầu, trạng thái địch.

1. **GIỚI THIỆU**

Đồ án đặt ra một yêu cầu là thiết lập chương trình gane 8-puzzle gồm 2 đối tượng là người và máy chơi. 8-puzzle có hình dạng như sau:



***Hình 1.*** *Trạng thái của 8-puzzle*

Để đơn giản trong cách tiếp cận giải bài toán, người ta giả định chỉ có ô trống trong bảng là di chuyển đến những vị trí khác. Như vậy tại một trạng thái thì chỉ có tối đa 4 cách đi để chuyển sang trạng thái khác (trái, phải, lên, xuống). Người ta cũng nhận ra được rằng để có thể chuyển từ 1 trạng thái bất kì về trạng thái đích như trên thì trạng thái đầu đó phải theo một quy luật.

Các phần còn lại của bài báo cáo được tổ chức như sau. Phần 2 trình bày về phương pháp tìm hiểu và thực hiện, bao gồm các thuật toán sử dụng, mã giả và các ràng buộc cho các trường hợp đặc biệt.Phần 3 trình bày kết quả của quá trình thực hiện và một số ví dụ kết quả mà chương trình chạy được.Phần 4 tổng hợp và kết luận về đồ án này.

1. **PHƯƠNG PHÁP**

**2.1. Cách nghiên cứu và thực hiện**

Có n2-1 số mang các giá trị từ 1 tới n2-1 được sắp xếp vào một lưới các ô vuông kích thước n x n. Mỗi số đó được gọi là một quân cờ và lưới ô đó được gọi là bàn cờ. Có một vị trí của bàn cờ bỏ trống. Mỗi lần di chuyển quân, người chơi được phép chuyển một quân ở vị trí ô tiếp giáp cạnh với ô trống vào ô trống.

Từ một trạng thái ban đầu (sự sắp xếp ban đầu của các quân trên bàn cờ), thực hiện các nước đi hợp lệ để thu được trạng thái kết thúc (trạng thái đích cần đạt được).

Có những trạng thái của bảng số không thể chuyển về trạng thái đích, ta gọi là **cấu hình hợp lệ** và không hợp lệ. Tỉ lệ giữa chúng là ½, điều này có thể nhận ra dễ dàng từ phương pháp tính xem bài toán có thể đưa về trạng thái đích hay không.

Rất dễ thấy là mỗi trạng thái của bảng số là một hoán vị của m x m phần tử (với m là cạnh), như vậy **không gian trạng thái** của nó là (m x m)!, với 8-puzzle là 9! = 362880 (m = 3) và 15-puzzle là 16! = 20922789888000 (m =4). Bạn có thể khi m tăng lên 1 đơn vị thì không gian trạng thái tăng lên rất nhanh, điều này khiến cho việc giải quyết các phiên bản m>3 ít khi được áp dụng.

Thuật toán A\* là một trường hợp của Best First Search nên ta sẽ đi tìm hiếu trực quan Best First Search trước khi bắt tay vào A\* để có thể có một cái nhìn chung về những thuật toán tìm kiếm tối ưu này.

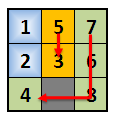
Thuật toán tìm kiểm tối ưu – Best First Search

ưu tiên tối ưu là sự kết hợp cả hai thuật toán tìm kiếm theo chiều sâu và thuật toán tìm kiếm theo chiều rộng, cho phép người dùng đi một con đường duy nhất tại một thời điểm, nhưng đồng thời vẫn xem xét các hướng đi khác. Nếu nhận thấy rằng hướng đang đi không có triển vọng bằng những con đường còn lại thì ta sẽ chuyền sang đi theo một trong những đường đó.

A\* là một giải thuật tìm kiếm thường được sử dụng trong các vấn đề liên quan đến đồ thị và tìm đường đi. Nó được chọn không chỉ vì tính hiệu quả mà còn vì rất dễ dàng để hiểu và cài đặt

Để áp dụng thuật toán A\* giải bài toán này, bạn cần một hàm heuristic h để ước lượng giá trị của mỗi trạng thái của bảng số. Có một số cách có thể đã biết tới như tính dựa vào khoảng cách sai lệch của các ô số với vị trí đúng, hoặc đơn giản là đếm xem có bao nhiêu ô sai vị trí,…

Ở đây tôi chọn theo cách thứ nhất, tức là tính tổng số ô sai lệch của các ô số so với vị trí đúng của nó. Đây là cách tính thường được sử dụng và nó có tên gọi là Manhattan



***Hình 2.*** *Tính khoảng cách Manhattan*

**2.2. Mã giả của các thuật toán và phương pháp được sử dụng**

**BFS(Node bắt đầu, Node mục tiêu)**

//Khởi tạo node list lưu trữ các node

Node\_List\* listPoint = newNodeList();

//Khởi tạo node list kiểm tra các node đã xét

Node\_List\* checkExist = newNodeList();

//Thêm node bắt đầu vào listPoint

addNodeList(start, listPoint);

//Khởi tạo node gốc

Node\* root;

do

{

//lấy node đầu tiên trong listPoint ra và gán vào node gốc

root = popNodeList(listPoint);

//Duyệt các hành động của node gốc

for (int i = 0; i < 4; i++)

{

//Kiểm tra hành động hợp lệ không và có đi ngược lại hành động vừa thực hiện không

if (isLegalMove(root, i) && isOposite(root->action, i) == false)

{

//Tạo node con của node gốc mang giá trị hành động đang duyệt

Node\* temp = childNode(root, i);

//Kiểm tra có trùng với node mục tiêu không

if (isEqual(temp->state, goal->state))

{

//Truyền đường dẫn từ node đang xét đến node bắt đầu để hình thành đường đi

listMove(start, temp, goal);

//Kết thúc hàm tìm kiếm

return true;

}

//Thêm node đã xét vào listPoint

addNodeList(temp, listPoint);

}

}

//Thêm node gốc vào checkExist để kiểm lưu lại node đã xét

addNodeList(root, checkExist);

} while (root->depth < 25);

//Lặp đến khi độ sâu bằng 25

return false;

**AStar( Node bắt đầu, Node mục tiêu)**

{

//Khởi tạo node list lưu trữ các node

Node\_List\* listPoint = newNodeList();

//Khởi tạo node list kiểm tra các node đã xét

Node\_List\* checkExist = newNodeList();

//Tính chi phí theo hàm Heuristic

start->cost = Heuristic(start->state, goal->state);

//Thêm node bắt đầu vào listPoint

addNodeList(start, listPoint);

//Khởi tạo node gốc

Node\* root;

//Khởi tạo và gán số bước đi bằng 0

int step = 0;

do

{

//lấy node đầu tiên trong listPoint ra và gán vào node gốc

root = popNodeList(listPoint);

//Duyệt các hành động của node gốc

for (int i = 0; i < 4; i++)

{

//Kiểm tra hành động hợp lệ không và có đi ngược lại hành động vừa thực hiện không

if (isLegalMove(root, i) && isOposite(root->action, i) == false)

{

//Tạo node con của node gốc mang giá trị hành động đang duyệt

Node\* temp = childNode(root, i);

//Kiểm tra node con này đã duyệt chưa

if (isExistInNodeList(temp, checkExist) == false)

{

if (isEqual(temp->state, goal->state))

{ //Truyền đường dẫn từ node đang xét đến node bắt đầu để hình thành đường đi

listMove(start, temp, goal);

//Kết thúc hàm tìm kiếm

return true;

}

//Tính chi phí của node con theo hàm Heuristic

temp->cost = step + Heuristic(temp->state, goal->state);

//Thêm node con vào listPoint theo thứ tự tăng dần chi phí của node

insertNodeList(temp, listPoint);

}

//Nếu đã duyệt thì xóa node con

else delete temp;

}

}

//Thêm node gốc vào checkExist để kiểm lưu lại node đã xét

addNodeList(root, checkExist);

//Tăng bước đi

step++;

} while (true);

return false;

}

**Heuristic( Mảng các số của node bắt đầu, Mảng các số của node mục tiêu)**

{

//Khởi tạo các mảng thể hiện tọa độ của các điểm

int xS[9] = { 0 };

int yS[9] = { 0 };

int xG[9] = { 0 };

int yG[9] = { 0 };

//Khởi tạo và gán điểm bằng 0

int point = 0;

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

{

//Lưu vị trí của từng điểm vào các bảng tọa độ

xS[S[i][j]] = i; yS[S[i][j]] = j;

xG[G[i][j]] = i; yG[G[i][j]] = j;

}

for (int i = 0; i < 9; i++)

{

//Tính số bước đi cần thực hiện để đưa các điểm về đúng vị trí

int Xpoint = (xS[i] > xG[i]) ? (xS[i] - xG[i]) : ( xG[i] -xS[i] );

int Ypoint = (yS[i] > yG[i]) ? (yS[i] - yG[i]) : ( yG[i] -yS[i] );

//Điểm bằng tổng số bước đi tìm được

point += Xpoint + Ypoint;

}

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

{

//Tính số ô khác nhau

if (S[i][j] != G[i][j])

{

//Điểm cộng thêm số ô cờ khác nhau

point++;

}

}

return point;

}

**addNodeList(Node bắt đầu, Node mục tiêu)**

{

//Tăng chiều dài của danh sách

NodeList->length++;

//Kiểm tra nodelist đã có node đầu chưa

if (NodeList->head == NULL)

{

//Gán node đầu của nodelist là node

NodeList->head = node;

}

//Kiểm tra nodelist đã có node đuôi chưa

else if (NodeList->tail == NULL)

{

//Gán node đuôi của nodelist là node và tạo liên kết với node đầu

NodeList->tail = node;

NodeList->tail->undoNode =

NodeList->head;

NodeList->head->nextNode =

NodeList->tail;

}

//Nodelist đã có node đầu và đuôi

else

{

//Gán node đuôi của nodelist là node và tạo liên kết với node đuôi

NodeList->tail->nextNode = node;

node->undoNode = NodeList->tail;

NodeList->tail = node;

}

}

**insertNodeList(Node bắt đầu, Node mục tiêu)**

{

//Tăng chiều dài của danh sách

NodeList->length++;

//Kiểm tra nodelist đã có node đầu chưa

if (NodeList->head == NULL)

{

//Gán node đầu của nodelist là node

NodeList->head = node;

}

//Kiểm tra nodelist đã có node đuôi chưa

else if (NodeList->tail == NULL)

{

//So sánh giá trị của node với node đầu

if (NodeList->head->cost > node->cost)

{

//Gán node đuôi của nodelist là node đầu và node đầu của nodelist là node

NodeList->tail = NodeList->head;

NodeList->head = node;

}

//Gán node đuôi của nodelist là node và tạo liên kết với node đầu

else NodeList->tail = node;

NodeList->head->nextNode = NodeList->tail;

NodeList->tail->undoNode = NodeList->head;

}

//So sánh giá trị của node với node đầu

else if (NodeList->head->cost >= node->cost)

{

//Gán node đầu của nodelist là node vào tạo liên kết

node->nextNode = NodeList->head;

NodeList->head->undoNode = node;

NodeList->head = node;

}

//So sánh giá trị của node với node đuôi

else if (NodeList->tail->cost <= node->cost)

{

//Gán node đuôi của nodelist là node và tạo liên kết

node->undoNode = NodeList->tail;

NodeList->tail->nextNode = node;

NodeList->tail = node;

}

else

{

//Thêm node vào nodelist ở vị trí đầu tiên có giá trị lớn hơn node

Node\* root = NodeList->head;

while (root->nextNode != NULL && root->cost < node->cost)

root = root->nextNode;

node->nextNode = root;

node->undoNode = root->undoNode;

root->undoNode = node;

}

}

1. **KẾT QUẢ**

Tìm kiếm đường dẫn để đưa trạng thái bắt đầu đến trạng thái kết quả cho sẵn. Thời gian để tìm ra được đường đi của thuật toán A\* ngắn hơn khá nhiều so với thuật toán BFS.

Ta thấy thuật toán A\* sẽ tối ưu trong nhiều trường hợp sẽ tối ưu hơn thuật toán BFS. Về bản chất của thuật toán, ta cũng có thể dễ dàng nhận ra sự khác biệt này. BFS là thuật toán tìm kiếm mù đồng nghĩa với việc sẽ duyệt tất cả các trạng thái. Điều này sẽ làm cho thuật toán tốn bộ nhớ và thời gian. Trong khi đó A\* là thuật toán có hàm đánh giá Heuristic nên chỉ duyệt những trạng thái được đánh giá là tối ưu mà thôi. Cho nên sự chênh lệch thời gian ở bảng thống kê trên là minh chứng cho sự tiện ích mà hàm A\* mang lại.

|  |  |  |  |
| --- | --- | --- | --- |
| Trạng thái | BFS | | A\* |
| S: 012345678  G: 123456780 | Số bước | 22 | 22 |
| Thời gian | 0,374s | **0,002s** |
| S: 628351074  G: 051632874 | Số bước | **18** | 20 |
| Thời gian | 0,044s | **0,001s** |

1. **KẾT LUẬN**

Trong đồ án đã áp dụng các kiến thức đã học như A\* search algorithm và Breadth-First search và tư duy của môn trí tuệ nhân tạo để giải thuật toán. Qua đó hiểu được bản chất của 2 thuật toán này để áp dụng vào vấn đề. Dễ dàng thấy được điểm nổi bật cuat thuật toán A\* với việc tích hợp hàm Heuristic giúp nó tối ưu hơn BFS. Qua đồ án này giúp ôn lại các kiến thức đã được học ở môn học, biết được cách phát triển trí tuệ cho máy để có thể chơi được với người.

**TÀI LIỆU THAM KHẢO**

[1] <https://yinyangit.wordpress.com/2010/12/16/algorithm-phan-tich-va-gi%E1%BA%A3i-bai-toan-n-puzzle/> A\* algorithm

[2] <https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_r%E1%BB%99ng> BFS algorithhm

[3] Tất cả các code và bài giảng của môn Trí Tuệ Nhân tạo do GV.Huỳnh Xuân Phụng giảng dạy