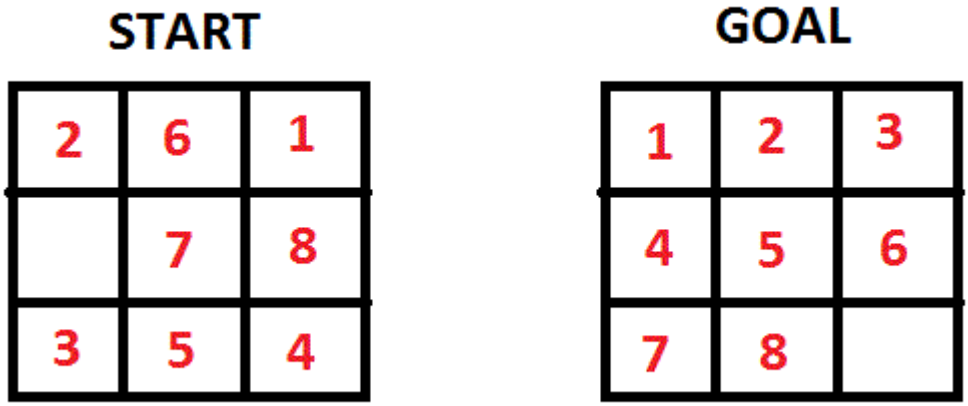
1. Bài toán đặt ra, mục tiêu

- Bài toán 8 puzzle dựa trên game 8 puzzle. Cho 9 ô có dạng 3x3, với các ô số đã bị đảo lộn, và có một lỗ trống, để di chuyển cho các ô, yêu cầu đặt ra là sắp xếp các ô sao cho quay về trạng thái đích ban đầu.



- Vấn đề đặt ra là làm sao có thể đưa các vị trí đã bị đảo lộn sang vị trí đích.
- Để giải được vấn đề trên ta cần xây dựng thuật giải có khả năng tìm được các đường có thể đi đến điểm đích.
- Thuật giải sẽ được trình bày dưới đây.

2. Ý tưởng và thuật giải bằng mã giả

a. Ý tưởng cơ bản

- Bài toán trên được giải theo phương pháp A*, kết hợp với yếu tố Heuristic.

- Đầu tiên ta sẽ sinh ra các trường hợp có thể di chuyển được từ trạng thái ban đầu. Sau đó ta sẽ ước lượng với từng trường hợp sinh ra trường hợp nào có số ước lượng thấp nhất (có khả năng đến đích trước), ta sẽ chọn trường hợp đó để đi trước. Với trường hợp mới đã chọn ta lại tiếp tục sinh ra các trường hợp khác để ước lượng. Cứ như vậy cho đến khi trường hợp được sinh ra giống hoàn toàn với mẫu goal.

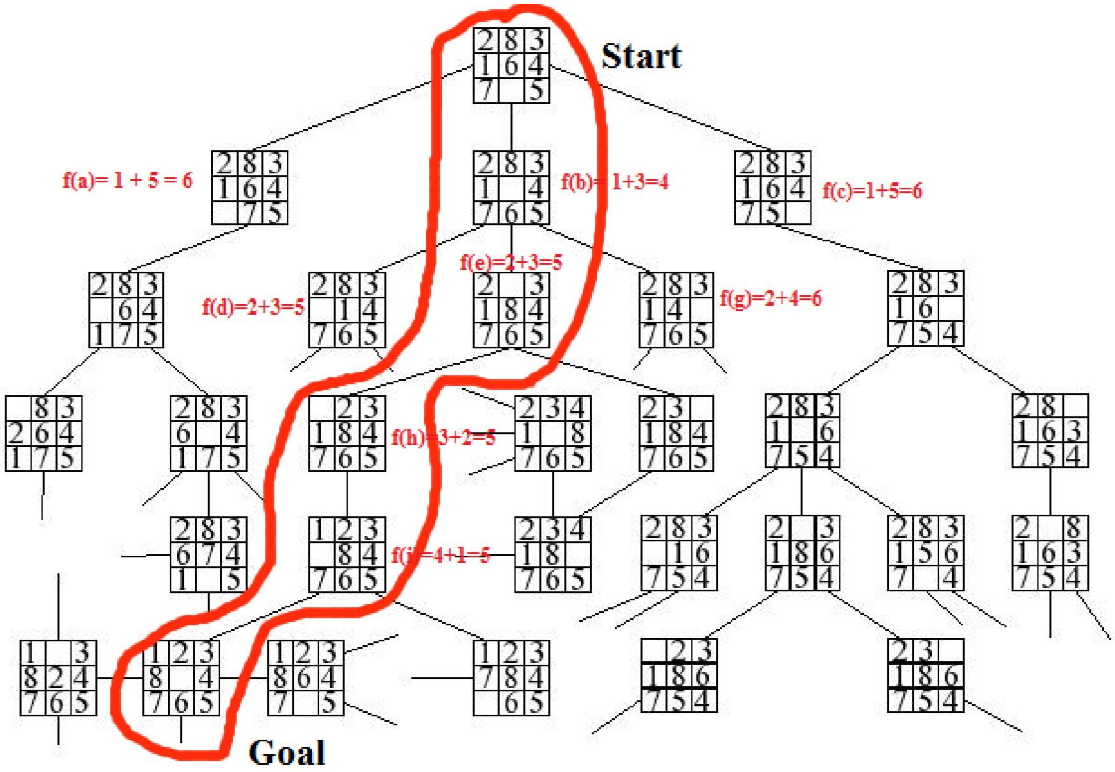
- Ý tưởng bài toán sử dụng phương pháp A*, bằng cách dùng khả năng ước lượng thông qua công thức f(n) = g(n) + h(n).

- Với g(n) là số các bước đã đi được, h(n) là khoảng cách ước lượng từ vị trí hiện tại đến vị trí đích. f(n) là số đánh giá trường hợp có khả thi không.

- Điều quan trọng nhất để giải bài toán này đó chính là phải tìm yếu tố Heuristic để dự đoán khoảng cách của vi trí hiện tại còn cách vị trí đích bao xa đó là hàm h(n).

- Hàm h(n), đếm số lượng hiện tại các ô số sai vị trí so với các ô số đích.

- Ta sẽ minh họa ví dụ cụ thể trong trường hợp dưới đây.



- Ta xét dòng 1, f(a) = g(a)+h(a)=1+5=6, g(a)=1 vì đã đi qua được 1 bước di chuyển, h(a)=5, do có 5 ô 2, 8, 7, 1, 6 sai so với vị trí ô đích. Lần lượt tính như vậy đối với các trường hợp f(b), f(c). Ta sẽ chọn ra f nào có giá trị thấp nhất đó là f(b).
- Ta tiếp tục đi từ f(b) lại sinh ra các trường hợp có thể di chuyển, rồi tính f() của từng trường hợp.
- Quá trình lặp cho đến khi gặp giá trị đích goal thì ngưng.

b. Thuật giải mã giả.

- Dưới đây là thuật giải bằng mã giả.(trích: http://web.cecs.pdx.edu/~mm/AIFall2011/ProblemSolvingAsSearch.pdf)

```
create the open list of nodes, initially containing only our starting node

create the closed list of nodes, initially empty

while (we have not reached our goal) {
    consider the best node in the open list (the node with the lowest f value)

    if (this node is the goal) { then we're done }
    else {
            move the current node to the closed list and consider all of its successors

            for (each successor) {
                    if (this suceessor is in the closed list and our current g value is lower) {
                            update the successor with the new, lower, g value
                            change the sucessor's parent to our current node }

                    else if (this successor is in the open list and our current g value is lower) {
                            update the suceessor with the new, lower, g value
                            change the sucessor's parent to our current node }

                        else this sucessor is not in either the open or closed list {
                            add the successor to the open list and set its g value } } } }
```

3. Hiện thực hóa

a. Cấu trúc dữ liệu:

- Ta sẽ tạo lớp class tên matrixObj, bao gồm các thành phần chính:
  - Matrix: khối ma trận lưu các trạng thái của các ô số
  - Parent: lưu lại trạng thái trước khi sinh ra trạng thái hiện tại.
  - Depth: lưu số bước đã di chuyển được để đến trạng thái hiện tại.
  - hVal: ước lượng khoảng cách giữa trạng thái hiện tại với trạng thái đích.
- Ngoài ra, còn có một số hàm xử lý: di chuyển, sinh ra danh sách các trạng thái mới (generatePattern), ước lượng khoảng cách thông qua yếu tố heuristic.

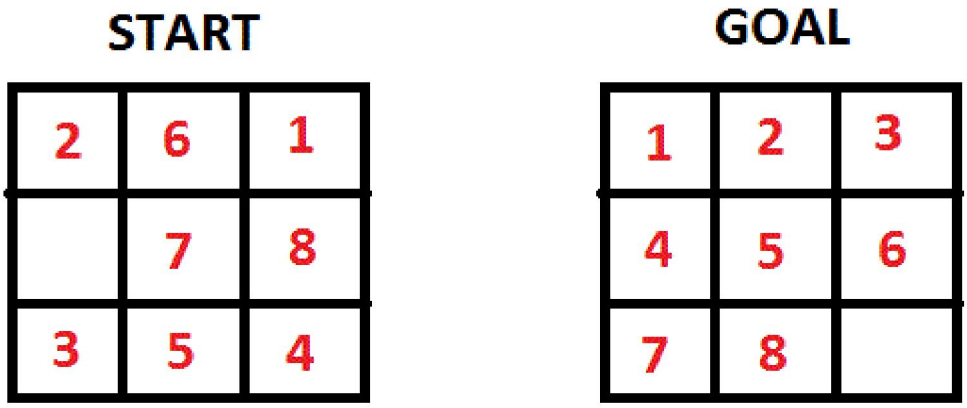
b. Hiện thực hóa thuật giải:

- Trước khi đi sâu vào thuật toán, ta sẽ quay lại tìm hiểu yếu tố đó là làm cách nào để xác định được khoảng cách ngắn nhất của các đối tượng dựa vào yếu tố Heuristic.
- Theo như thuật toán A*, việc chọn lựa các đường đi phù hợp dựa theo công thức f(n)= g(n) + h(n). Trong đó h(n) tính theo phương pháp **Manhattan Distance**, vì nếu theo cách tính ở phần 2.a, đó là đếm các ô sô sai lệch so với ô đích sẽ không chính xác và gây ra hiện tượng lặp nhiều lần trong quá trình giải bài toán**.**
- h(n) được tính theo công thức sau:

$$h(s) = \sum_{i=1}^{8} (|x_i(s) - \overline{x}_i| + |y_i(s) - \overline{y}_i|).$$

- Trong đó:
  - Với mỗi h(s), tức với mỗi trạng thái của các ô số.
  - xi(s), yi(s) là vị trí của 1 ô số trong bảng ma trận hiện tại.
  - xi, yi là vị trí của ô số đó trong bảng ma trận goal.

- Ta sẽ xét ví dụ sau:



- Xét mẫu Start, số 2 có vị trí (0,0), số 2 ở Goal có vị trí (0,1). S = 0, S = S + (|0-0| + |1-0|) = 1.
- Xét với số 6 ở vị trí Start là (0,1), số 6 ở vị trí Goal là (1,2), S = S + (|1-0| + |2-1|) = S + 2 = 1 + 2 = 3.

- Lần lượt như thế ta sẽ xét hết tất cả các ô còn lại. Cuối cùng h(s) = S.
- Dưới đây là thuật giải được viết bằng ngôn ngữ Python

```
1  p: pattern need solved
2  pGoal: the goal pattern
3
4  Open = [p]    // Open is a list (or array)
5  Close = []
6
7  While Open is not empty then loop {
8    current = Open.getFirstElement  // pop the first element in List
9
10   if isMatched(current, pGoal) // If current is matched to pGoal
11     return current
12
13   List = current.generatePattern() // generate new ways
14
15   Loop subP in List: // loop in List {
16     hVal = subP.mismatchNumber(pGoal) // calculate
17     fVal = current.depth + 1 + hVal
18
19     indexOpen = index(subP, Open)
20     indexClose = index(subP, Close)
21
22     if indexOpen == -1 and indexClose == -1 {
23        # new member
24        subP.parent = current
25        subP.depth = current.depth + 1
26        subP.hVal = hVal
27        Open.append(subP)
28     }
```

subP đại diện cho từng phần tử trong List

Dựa trên hàm heuristic để tính độ sai lệch (khoảng cách) giữa subP với pGoal

current.depth chính là chiều sâu (số bước đi từ đầu) đến trạng thái hiện thời

Tìm vị trí của subP trong tập Open và Close

Trường hợp không tìm thấy subP trong Open và Close

Thêm subP vào Open

```
29     elif indexOpen > -1 {
30         # subP similar parent's neighbour
31         neighbour = Open[indexOpen]
32         if fVal < neighbour.hVal + neib.depth {
33            neighbour.parent = current
34            neighbour.depth = current.depth + 1
35            neighbour.hVal = hVal
36         }
37     }
38     elif indexClose > -1: {
39         neighbour = Close[indexClose]
40         if fVal < neighbour.hVal + neib.depth {
41            subP.parent = current
42            subP.depth = current.depth + 1
43            subP.hVal = hVal
44            Close.remove(indexOpen)
45            Open.append(subP)
46         }
47     }
48   }
49   Close.append(current)
50   Open = sorted(Open, key=lambda obj: obj.hVal + obj.depth)
51 }
52
```

Ngược lại nếu đã có đối tượng trong tập Open giống với subP thì thực hiện

So sánh giá trị ước lượng giữa neighbour (đối tượng đã có trong Open) với đối tượng hiện tại

Nếu fVal nhỏ hơn, thì ta sẽ cập nhật cho đối tượng neighbour, và thay parent của neighbour là current

Nếu đã có đối tượng trong tập Close giống với subP thì thực hiện

Lấy đối tượng trong tập Close dựa trên indexClose

Nếu fVal nhỏ hơn giá trị ước lượng của neighbour, chứng to đã tìm ra được con đường ngắn hơn để tới được neihgtbour hiện tại

Xóa bỏ đối tượng hiện tại, để cập nhật lại vào Open

Sau khi đã xét xong, ta sẽ cho vào Close để đánh dấu là không xét nữa

Sắp xếp tăng dần theo khoảng cách ước lượng, để dễ lấy ra đối tượng có khoảng cách ngắn nhất đầu tiên

- **Giải thích**: Thuật toán trên cũng dựa trên mã giả, ban đầu ta cũng sẽ sinh ra các khả năng có thể đi, sau đó tạo vòng lặp kiểm tra các khả năng vưa sinh ra, nếu chưa có mẫu nào nằm trong tập Open và Close, thì thêm vào (dòng 22-27). Ngược lại, nếu có mẫu đã nằm trong Open từ trước nhưng có độ khả năng tới đích ngắn hơn thì ta sẽ cập nhật lại (dòng 29-35). Xét tiếp trường hợp nếu mẫu đó đã có nằm trong tập Close, và mẫu đó có khả năng tới đích ngắn hơn so với mẫu đã xét thì, ta sẽ lưu lại trường hợp đó vào tập Open (xem như có con đường ngắn hơn để tới cùng 1 mẫu), dòng 38-45.

- Sau khi giải ra được bài toán, ta sẽ viết hàm để dò vết các bước đã đi qua thông qua biến parent đã được lưu từ trước.

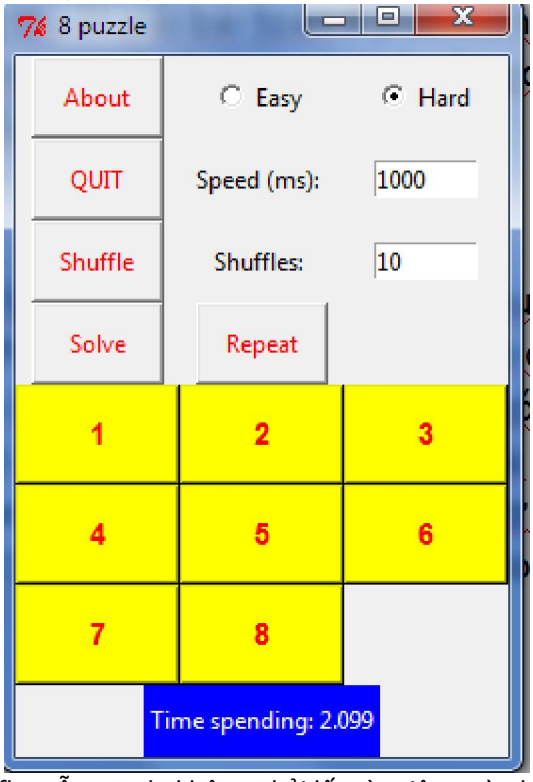
- Trên đây là ý tưởng chính để xây dựng thuật giải bài toán 8 puzzle, ngoài ra còn có một số hàm hỗ trợ cho bài toán như sinh ra các đường có khả năng đi (generatePattern), so sánh các mẫu đối tượng, …

c. Giao diện đồ họa:

- Ta sẽ viết ra hai phần riêng biệt để xử lý, một phần đó là xử lý về thuật giải đã trinh bày ở phần trên được viết trong eightPuzzle.py, phần còn lại là giao diện interface.py. Ngoài ra còn một số hàm khác hỗ trợ biên dịch đóng gói sang file exe.

- Ý tưởng xây dựng giao diện đồ họa, đó là hiển thị các nút button, và sẽ đọc ma trận từ dữ liệu đã được xử lý từ eightPuzzle.py, sau đó vẽ lên các đối tượng.

- Giao diện có thêm các chức năng như: Shuffle để tạo các mẫu random, đo thời gian, Repeat lặp lại các bước đi, tốc độ …



- Tuy nhiên, việc shuffle mẫu puzzle không phải lấy tùy tiện, mà phải từ mẫu Goal có sẵn (như hình trên), sau đó ta sẽ random các bước di chuyển để tạo thành các mẫu khác nhau. Vòng lặp các nhiều sẽ tạo ra các mẫu càng phức tạp.

- Để chạy thử chương trình trên, vào file dist/interface.exe

4. Phần cần cải tiến thêm

- Yếu tố vẫn chưa khắc phục được trong việc giải bài toán trên đó là việc tiêu hao thời gian vẫn còn khá lớn 30s-1.5 min, đối với bài toán có số lần shuffle cao (khoảng 100 lần).

Lưu ý về source code:

- Source code được viết bằng ngôn ngữ Python bao gôm 2 phần: eightPuzzle.py, interface.py

- Phần mềm compile source code là python 2.7, hoặc có thể chạy trực tiếp ở thư mục dist/interface.exe

5. Phần tham khảo:

https://heuristicswiki.wikispaces.com/Manhattan+Distance
http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html