Artificial Intelligence: Project 1

3190102060　黄嘉欣

1. N-Puzzle Problem.

As the simple program pipeline provided in *puzzle_state.py*, I finished the A* search algorithm by setting the backward cost g(n) of a child node as the number of steps form the initial state to this child state, and when it comes the forward cost h(n), it is measured using the Hamming distance, which is the number of different elements in the chessboard between the child state and the target state. In addition, the Manhattan distance and Euclidean Distance are applied too to compare the differences between these methods (For more details, please refer to the method *update_cost()* at line 551 of the file *puzzle_state.py*). Followings are the results of this problem.

a) Hamming distance

Lets comment out the output of halfway states in method *print_moves()* to see a brief outcome, which is shown as follows, meaning the algorithm works well.

```
Hamming distance
node expanded: 994

time consumed: 0.970165s

Initial state
----------------------
1       5       14      13
2       -1      4       7
11      10      6       15
8       9       12      3
----------------------

We get final state:
----------------------
1       4       5       14
2       6       13      15
11      7       -1      10
8       9       12      3
----------------------

Our dst state:
----------------------
1       4       5       14
2       6       13      15
11      7       -1      10
8       9       12      3
----------------------

Get to dst state. Success !!!
```

The time consumed when moving step number is **50** is 0.9702s and the

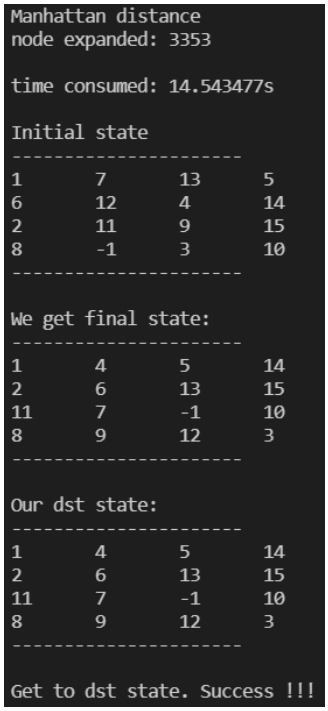
number of expanded nodes is 994. Take the randomness into consideration, I repeated running the program for 5 times setting the moving step number as 50 to check the time, which can be summarized as below:

times	1	2	3	4	5
Time/s	40.29581	0.57559	0.32813	0.05286	23.27601
Expanded nodes	5963	648	487	96	4847

The average time consumed is 12.9057s and the number of expanded nodes is 2408.2. However, as I can feel when running this program, it is a little bit time-wasting to use the Hamming distance as the heuristic function. We will later compare it with Manhattan distance and Euclidean distance.

b) Manhattan distance

Similar to we have done in part a), we tested the time that Manhattan distance used when setting the moving step number as **100**, whose result is as follows:



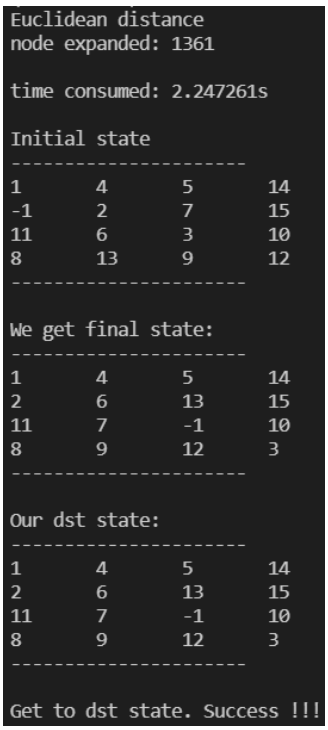
Using the way known as controlling variates, we set the input move number as **50**. The overall test results can be summarized as the table shows:

times	1	2	3	4	5
Time/s	0.01995	1.74625	0.08680	1.65364	0.05593
Expanded nodes	28	1078	128	883	67

The average time consumed is 0.7125s and the number of expanded nodes is 436.8. Obviously, when using Manhattan distance, it is less time-wasting because it uses relevant less time and expands less nods than Hamming distance as the input steps are both 50.

c) Euclidean distance

When we set moving steps as **50,** the outcome of the Euclidean distance is:



The results of 5 repeating tests are summarized as follows:

times	1	2	3	4	5
Time/s	0.34361	3.07045	0.09275	2.53921	0.76798
Expanded nodes	329	1201	82	1240	570

So using Euclidean distance, the algorithm consumes overage 1.3628s and expands 684.4 nodes. Compared with the other 2 options, Euclidean distance seems better than Hamming distance but worse than Manhattan distance for its number of expanded nodes is at a relevant middle level.

All in all, from the statics above, I think the Manhattan distance is a relevant better option for the heuristic function h(n), which could obtain the solution with a more efficient performance. On the other hand, we can also give

weighs to the backward cost g(n) and forward cost h(n)(not to big), which may help us get a more favorable performance. For example, when I change the cost function as f(n) = g(n) + 1.2*h(n) in part b), the results are:

times	1	2	3	4	5
Time/s	0.09477	0.10771	0.02993	0.17302	0.06184
Expanded nodes	98	121	32	169	89

In spite of the randomness, the performance seems better than the traditional cost function form f(n) = g(n) + h(n) to some extent.

d) Explanations and further explorations

As is shown above, the heuristic function of Manhattan distance consumes least time and expands fewest nodes. According to the theoretical knowledge, the cost function f(n) = g(n) + h(n), where the smaller h(n) is, the more nodes will the algorithm expands and the slower it will be. So as we have already known, the Hamming distance tends to be the smallest one in the 3 options and Manhattan distance is the biggest one. Although there is somehow randomness, the experiment's result is reasonable.

To test the heuristic function of Manhattan distance in harder scenarios, I increased the input steps and the square_size respectively, and the experimental results are as follows:

i) larger input steps

As I increased the input size from 50 to 150, the test results are:

Input steps	50	100	125	150
Time/s	0.48667	6.13134	19.94971	111.23577
Expanded nodes	457	2339	4528	10778

It is shown that the larger the input step is, the more nodes are expanded and time is consumed, which has a trend of exponential growth.

ii) larger square_size

When I tested the algorithm with square_size ranging from 3 to 6 while input steps stays being 50, the results are:

square_size	3	4	5	6
Time/s	0.01399	0.44880	1.65383	3.16044
Expanded nodes	32	431	996	1434

Similarly, as the square_size gets bigger, the difficulty level of this problem gets higher and the algorithm needs to expand more nodes and thus, consuming more time.

2. Tic-Tac-Toe Problem.

According to the simple pipeline, I completed the minimax search algorithm by setting the utility function Eval(state) = 10*O2 + O1 - (10*X2 + X1), where Xn means the number of rows/columns/diagonals/anti-diagonals that only contains n Xs and no Os for non-terminal states and On has a similar meaning. When it comes to the terminal states, the function Eval(state) will be assigned as 100 for the cases computer wins or -100 for the cases human wins. If it is a draw, there will be no additional processing for the score.

The reason why I assigned weights like that is that different states deserve different scores. It is more likely for the computer to win if the value of O2 is bigger and the same as the X2. So I assigned them a wight of 10, which is relevantly higher than the weight of O1 and X1. In addition, for the efficiency of the algorithm, linear evaluation function is applied. Nevertheless, for the terminal states, it must be distinguishable for the algorithm to tell the winning match from those draws and non-terminal states, so a strong reward/penalty is needed, which was set by me as 100/-100.

As for the methods *min_value()* and *max_value()*, they are implemented based on the assumption that the computer is the MAX player, who attempts to maximize the utility while the human, going first, does not. When it is the computer's turn, it will recursively get all possible states for the next several steps and calculate each state's score, choosing the action with the largest value. For more details, please refer to the methods at line 85 and 103 of the file *tic_tac_toe.py*.

The followings are some game screenshots between me and the computer, which are at most draws and I never won ☹, as the description file says.