# Project 1

3180103779

谢涛羽

## n\_puzzle

* 1. update\_cost function
     1. parameter explanation

1. curr\_state & dst\_state
2. method

Using specific method to determining heuristic function. It can be

* 'euclidean': 
* 'manhattan': 
* 'chebyshev': 
* 'hamming': 

1. Return curr\_state with cost h and g updated.
   * 1. code structure

A loop is used to get every number’s position, and calculate the number’s distance to the destination.

Adding up all the distance, we get the heuristic function of curr\_state.

As for g, the price which has been cost, we need to find the previous state of curr\_state.

Since g stands for the cost from initial state to curr\_state, so we update g via



because from pre\_state to curr\_state, we move once, add 1 for the cost.

Use the h calculated to update curr\_state.h

Use pre\_state’g+1 to update curr\_state.g

Calculate h respect to different method

Calculate the coordinate difference separately

Get coordinates

def update\_cost(curr\_state, dst\_state, method):

h = 0

for i in range(curr\_state.state.shape[0]):

for j in range(curr\_state.state.shape[1]):

curr\_row, curr\_col = curr\_state.num\_pos(curr\_state.state[i][j])

dst\_row, dst\_col = dst\_state.num\_pos(curr\_state.state[i][j])

x = abs(curr\_row - dst\_row)

y = abs(curr\_col - dst\_col)

if method == 'euclidean':

h = h + (x \*\* 2 + y \*\* 2) \*\* 0.5

elif method == 'manhattan':

h = h + x + y

elif method == 'chebyshev':

h = h + max(x, y)

elif method == 'hamming':

if x != 0 or y != 0:

h = h + 1

curr\_state.h = h

curr\_state.g = curr\_state.pre\_state.g + 1

return curr\_state

* 1. find\_front\_node
     1. parameter explanation

open\_list[]:

This is a list stored with unexplored nodes, but these nodes are expanded before. We are searching for the node with smallest price g+h.

* + 1. code structure

Return the node together with it’s index

Find the element with smallest g+h

Initialize the smallest cost with the first element

def find\_front\_node(open\_list):

index = 0

cost = open\_list[0].g + open\_list[0].h

for i in range(1, len(open\_list)):

cost\_i = open\_list[i].g + open\_list[i].h

if cost\_i < cost:

index = i

return index, open\_list[index]

* 1. get\_path
     1. parameter expalanation

curr\_state:

We intend to find the path from initial state to curr\_state.

Curr\_state.pre\_state and curr\_state.pre\_move recursively indicate the path.

pre\_move

curr\_state

pre\_state

pre\_pre\_state

pre\_pre\_move

……

* + 1. code structure

Reverse move\_list, a path from top to bottom, from the start to curr\_state, is obtained

From bottom to top,

add the move to move\_list

def get\_path(curr\_state):

move\_list = []

state = curr\_state

while state.pre\_move != None:

move\_list.append(state.pre\_move)

state = state.pre\_state

move\_list.reverse()

return move\_list

* 1. expand\_state
     1. parameter expalanation

curr\_state:

From curr\_state, we normally can get 4 next\_states, to move left, right, up, down (if the move is valid).

We should put the valid next\_states into open\_list because these are possible ways to the destination state.

* + 1. code structure

4 next\_states,

If valid, expand it.

def expand\_state(curr\_state):

childs = []

for i in range(4):

valid, next\_state = once\_move(curr\_state, i)

if valid:

childs.append(next\_state)

return childs

* 1. state\_in\_list
     1. parameter explanation

state, list

Judge whether state is in list or not.

If the list is open\_list, it means the state has been expand before.

If the list is close\_list, it means the state has been explored before.

* + 1. code structure

Simple use a loop to compare whether the state is in list or not.

def state\_in\_list(state, list):

in\_list = False

match\_state = None

for item in list:

if item == state:

in\_list = True

match\_state = item

break

return in\_list, match\_state

* 1. thorough process

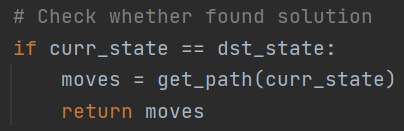
def astar\_search\_for\_puzzle\_problem(init\_state, dst\_state, method = 'manhattan'):

* 1. Get start\_state and dst\_state.

The default method

* 1. Defind empty open\_list and close\_list.
  2. Add start\_state to open\_list

LOOP: while open\_list not empty

* 1. Get the element with smallest (g+h), delete it from open\_list.
  2. Add the element into close\_list, which means it’s been explored.

else

* 1. Expan the element, get valid childs.

LOOP: traverse valid childs

* If a child in close\_list, continue.
* Update the child’s cost
* If a child in open\_list, compare their cost, use lower one.

But this situation hardly happens

If the one in list is with higher cost

Use lower cost

if in\_list:

if match\_state.g + match\_state.h > child\_state.g + child\_state.h:

match\_state.g = child\_state.g

match\_state.h = child\_state.h

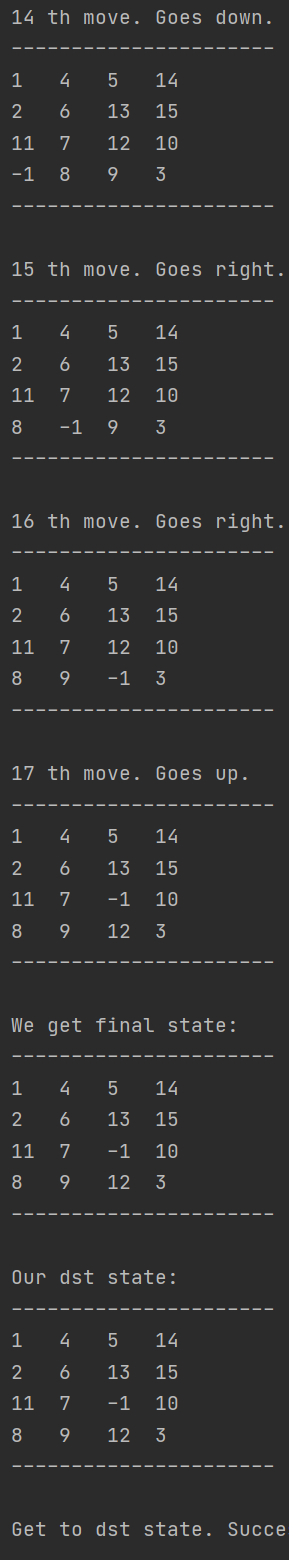
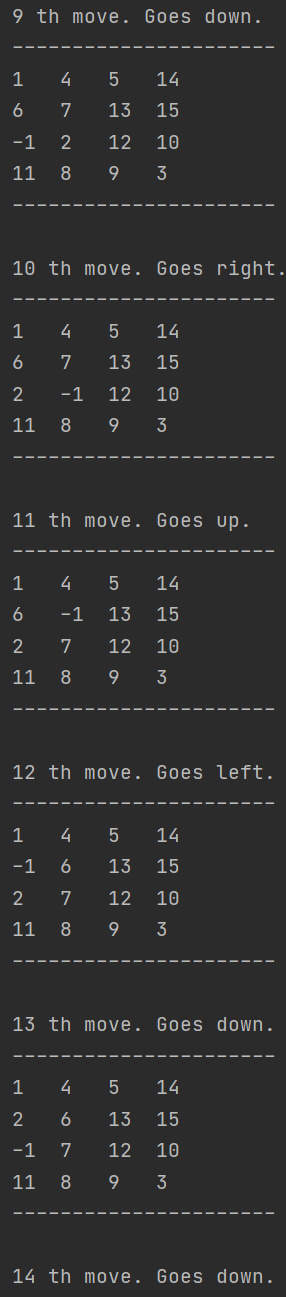
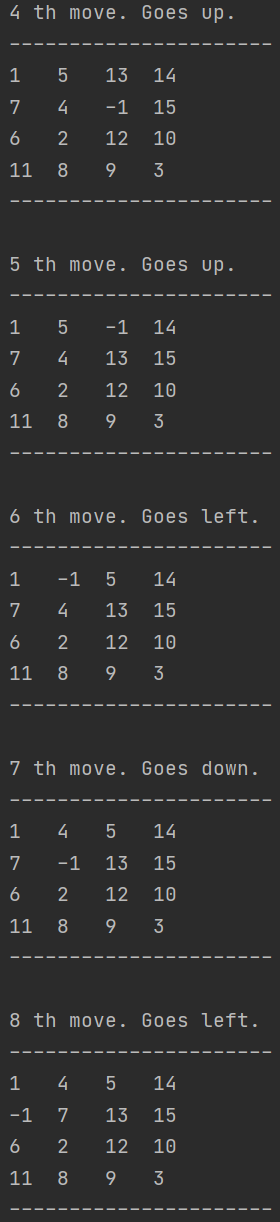
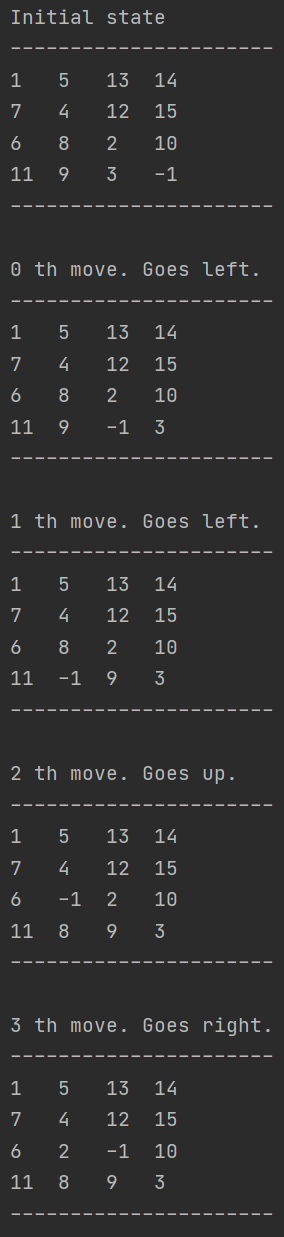
match\_state.pre\_state = child\_state.pre\_state

match\_state.pre\_move = child\_state.pre\_move

continue

* If not in open\_list, add the expanded child into open\_list.
  1. The results

The running time varies. Sometimes it is fast while others it’s slow.

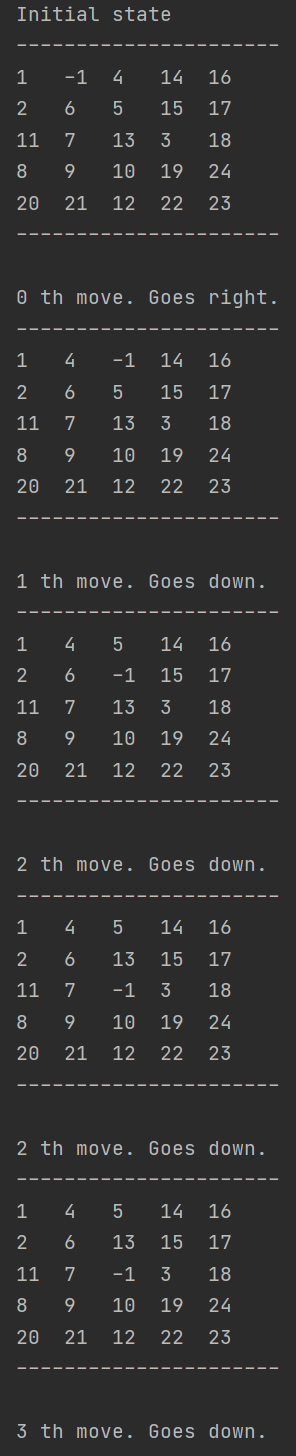
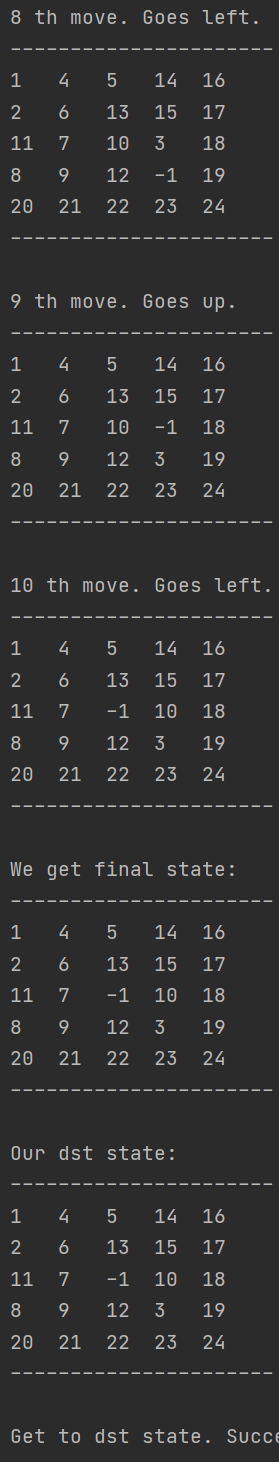
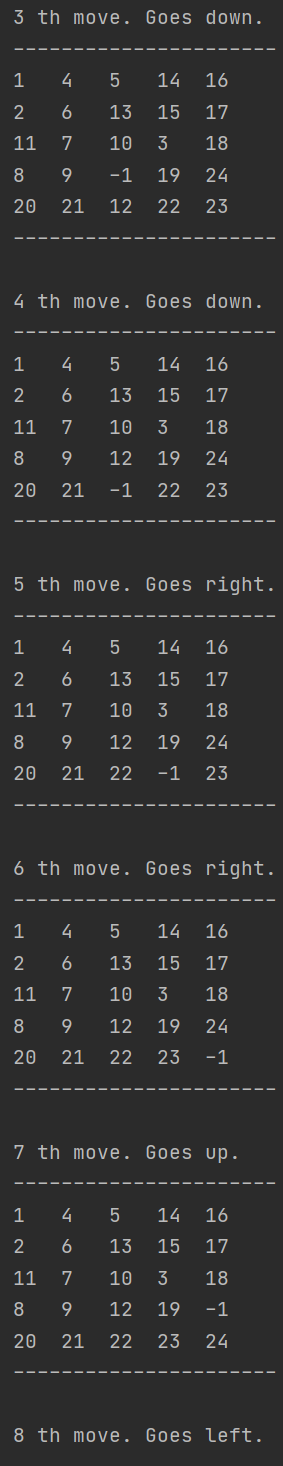
Using Manhattan distance:

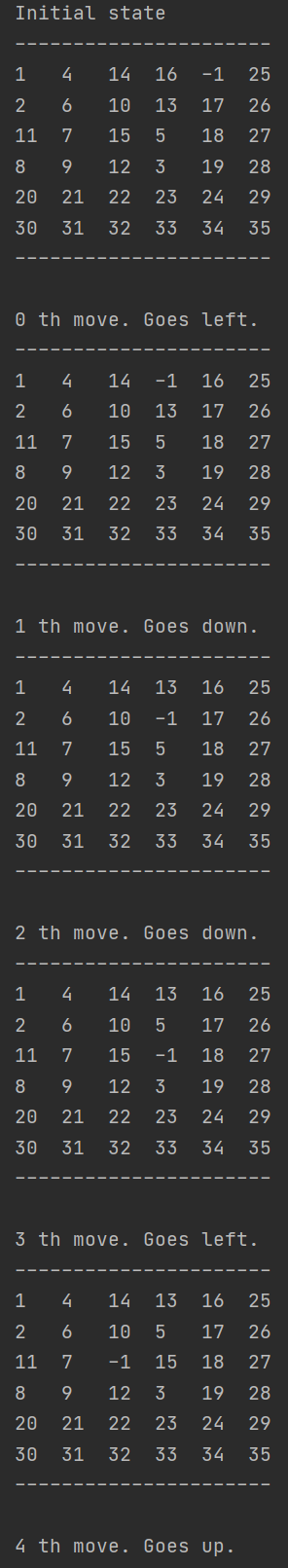
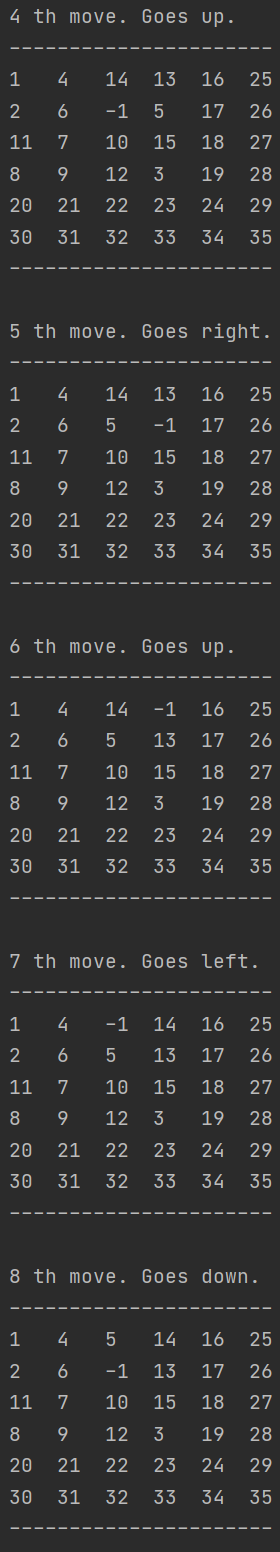
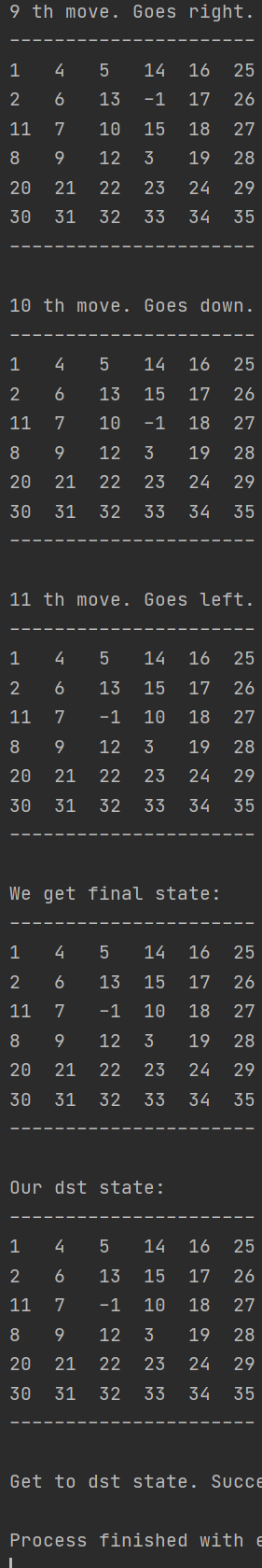
Change square size, this takes longer running time, so I change

move\_list = generate\_moves(30)

for simplicity

square size = 5:



squre\_size = 6

## tic\_tac\_toe

* 1. Utility Function
     1. Parameter explanations

1. current\_state: An array presents the current chess board.

-1 is the computer’s chess.

-1 is the human’s chess.

-0 means the position is blank

1. evaluation: the utility of the current state to be returned.
   * 1. Game situations

Based on the game rule:

* A line(row or column or diagonal) of three same chesses leads to the end.

【S1】Situations like these mean game over.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | X | X |  | X | … | … |  | X | … | … |
| … | … | … |  | X | … | … |  | … | X | … |
| … | … | … |  | X | … | … |  | … | … | X |

\*due to space limitation, only three typical situations are listed

* Two same chesses in a line(the third position is blank). These situations are close to win.

【S2】Situations like these have high probability to finish.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | X |  |  | X | … | … |  | X | … | … |
| … | … | … |  | X | … | … |  | … | X | … |
| … | … | … |  |  | … | … |  | … | … |  |

\*due to space limitation, only three typical situations are listed

* One chess in a line with other two positions blank. These situations, though not likely to win, show some advantages of the chess player.

【S3】Situations like these have chance to win(though the possibility is small)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X |  |  |  |  | … | … |  | … | X | … |
| … | … | … |  | … | X | … |  | … |  | … |
| … | … | … |  | … | … |  |  | … |  | … |

\*due to space limitation, only three typical situations are listed

* Other situations’ utility is considered as zero.
  + 1. Utility calculation

On the board, computer uses 1, human uses -1. We calculate the sum of each row, column and diagonal.

|  |  |
| --- | --- |
| evaluation = 0 | The utility of current state to be returned. |
| max\_return = 100 | If 【S1】exists, max\_return is used to calculate evaluation. |
| two\_chess = 3 | If 【S2】exists, two\_chess is used to calculate evaluation. |
| one\_chess = 1 | If 【S3】exists, one\_chess is used to calculate evaluation |

1. Sum = 3

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 |  | 1 | … | … |  | 1 | … | … |  |
| … | … | … |  | 1 | … | … |  | … | 1 | … | …… |
| … | … | … |  | 1 | … | … |  | … | … | 1 |  |

From this state, we can tell max\_palayer is the winner, so the evaluation must be very big. So max\_return is used here.

max\_return is a big number, which is big enough to influence the utility.

1. Sum = -3

The same as sum = 3, but this time, min\_player wins, so -max\_return is used, which is small enough to influence the utility.

|  |  |
| --- | --- |
| sum = 3 | Evaluation += max\_return |
| Sum = -3 | Evaluation -= max\_return |

1. Sum = 2, -2, 1, -1

|  |  |
| --- | --- |
| Sum = 2 | Evaluation += two\_chess |
| Sum = -2 | Evaluation -= two\_chess |
| Sum = 1(product=0) | Evaluation += one\_chess |
| Sum = -1(product=0) | Evaluation -= one\_chess |

Product = 0 means the situation is 1+0+0 rather than 1-1+1.

under other situations, we don’t change the evaluation.

Here in sum = 3 and sum = -3, we need to emphasize that the state must be final state that one player wins.

If not, things like below will happen.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 |  |  | 1 | 1 | 1 |
|  |  |  | Take an action |  |  |  |
| -1 | -1 | -1 | 🡪 | -1 | -1 | -1 |

Before action, the utility is extremely low due to row3, after action, the utility is close to 0 due to row1. Obviously, it’s unreasonable.

So the when the caller calls *utility(state)* function, it must judge whether it is final state or not. If is, return the utility value, stop searching any further, no matter the depth is 0 or not.

(This is an attention need to be paid while designing min\_value and max\_value function, the caller of utility function)

Help calculate each row, column and diagonal’s evaluation

* + 1. Code structure

def utility(current\_state):

def utility\_in\_detail(a, b, c):

max\_return = 100

two\_same\_chess = 3

one\_chess = 1

s = a + b + c

if s == 3:

return 0.5\*max\_return

elif s == -3:

return -1\*max\_return

elif s == 2:

return two\_same\_chess

elif s == -2:

return -1 \* two\_same\_chess

elif s == 1:

if not a \* b \* c:

return one\_chess

elif s == -1:

if not a \* b \* c:

return -1 \* one\_chess

return 0

The weight of 【S2】 is 3, and the weight of 【S1】is 1. This is reasonable since 【S2】has much more probability to win.

A big number

def utility(current\_state, flag):

def utility\_in\_detail(a, b, c):

(…on the left…)

evaluation = 0

for i in range(3):

evaluation += utility\_in\_detail(rowi)

evaluation += utility\_in\_detail (coli)

evaluation += utility\_in\_detail (diagonal\_1)

evaluation += utility\_in\_detail (diagonal\_2)

return evaluation

【S1】case

【S2】case

【S3】case

Product should be 0

Add all evaluations and return

* 1. Actions\_result & get\_available\_actions

def get\_available\_actions(current\_state):

(some assert)

actions = []

for i in range(current\_state.shape[0]):

for j in range(current\_state.shape[1]):

if current\_state[i][j] == 0:

actions.append((i, j))

return actions

def action\_result(current\_state, action, player):

(some assert)

current\_state[action[0]][action[1]] = player return current\_state

Positions that are blank are available

Add player’s chess to the board.

* 1. Minimax Algorithm

def MinimaxSearch(current\_state):

(assert, game\_state copy)

actions = get\_available\_actions(game\_state)

values = []

depth = 3

for action in actions:

values.append(min\_value(action\_result(game\_state.copy(), action, 1), depth))

max\_ind = int(np.argmax(values))

row, col = actions[max\_ind][0], actions[max\_ind][1]

return row, col

Get available actions

Call min\_value,



def min\_value(current\_state, depth):

game = GameJudge()

game.game\_state = current\_state

check = game.check\_game\_status()

if check == 1 or check == -1:

return utility(current\_state)

if depth == 0:

return utility(current\_state, -1)

else:

actions = get\_available\_actions(current\_state)

if not actions:

return utility(current\_state, -1)

values = []

for action in actions:

values.append(max\_value(action\_result(current\_state.copy(), action, -1), depth-1))

min\_index = int(np.argmin(values))

return values[min\_index]

With the help of *Class GameJudge*, judge whether the current state is finished with a specific winner or not.

If is, check=1(max\_player wins) or check=-1(min\_player wins), this is a terminal state, return utility no matter depth is 0 or not.

Recursive exit

No actions available

Depth decrease

Switch player



def max\_value(current\_state, depth):

game = GameJudge()

game.game\_state = current\_state

check = game.check\_game\_status()

if check == 1 or check == -1:

return utility(current\_state)

if depth == 0:

return utility(current\_state, 1)

else:

actions = get\_available\_actions(current\_state)

if not actions:

return utility(current\_state, 1)

values = []

for action in actions:

values.append(min\_value(action\_result(current\_state.copy(), action, 1), depth - 1))

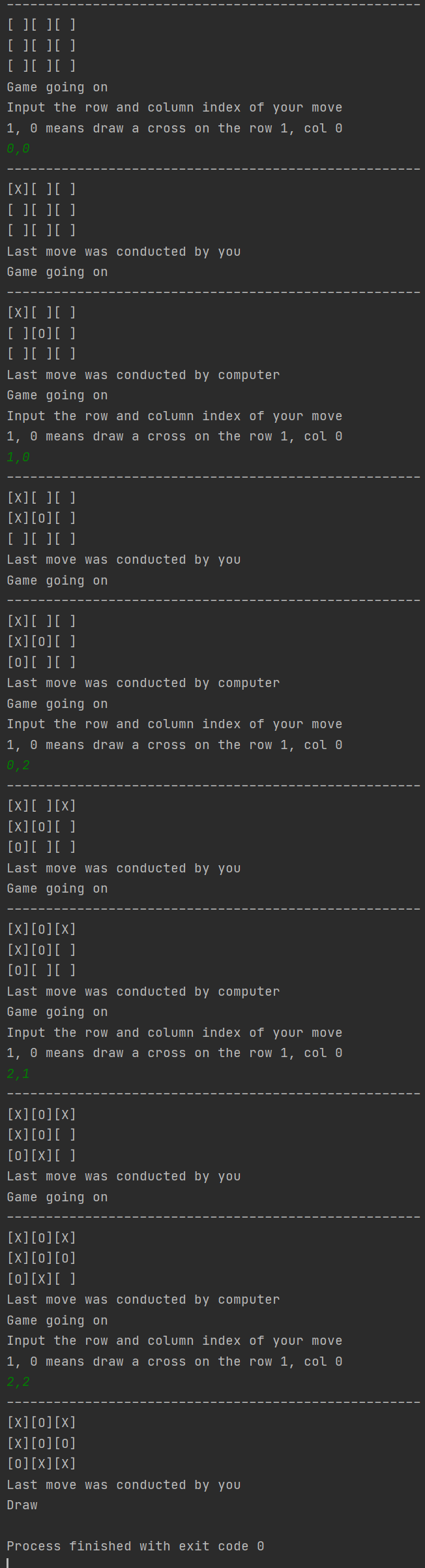
max\_index = int(np.argmax(values))

return values[max\_index]

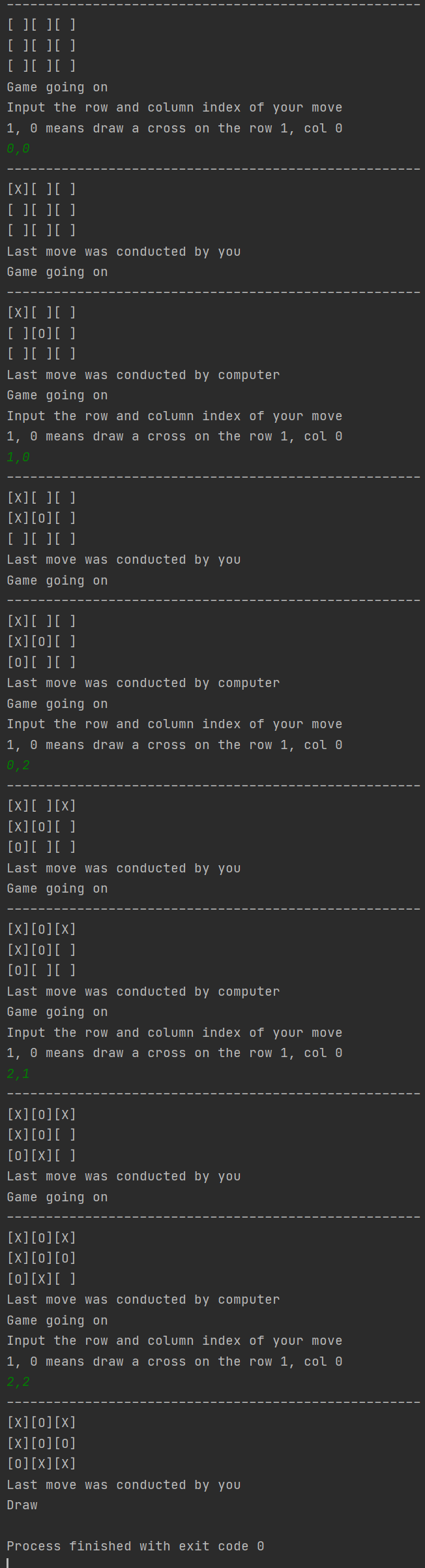
Almost same as min\_value

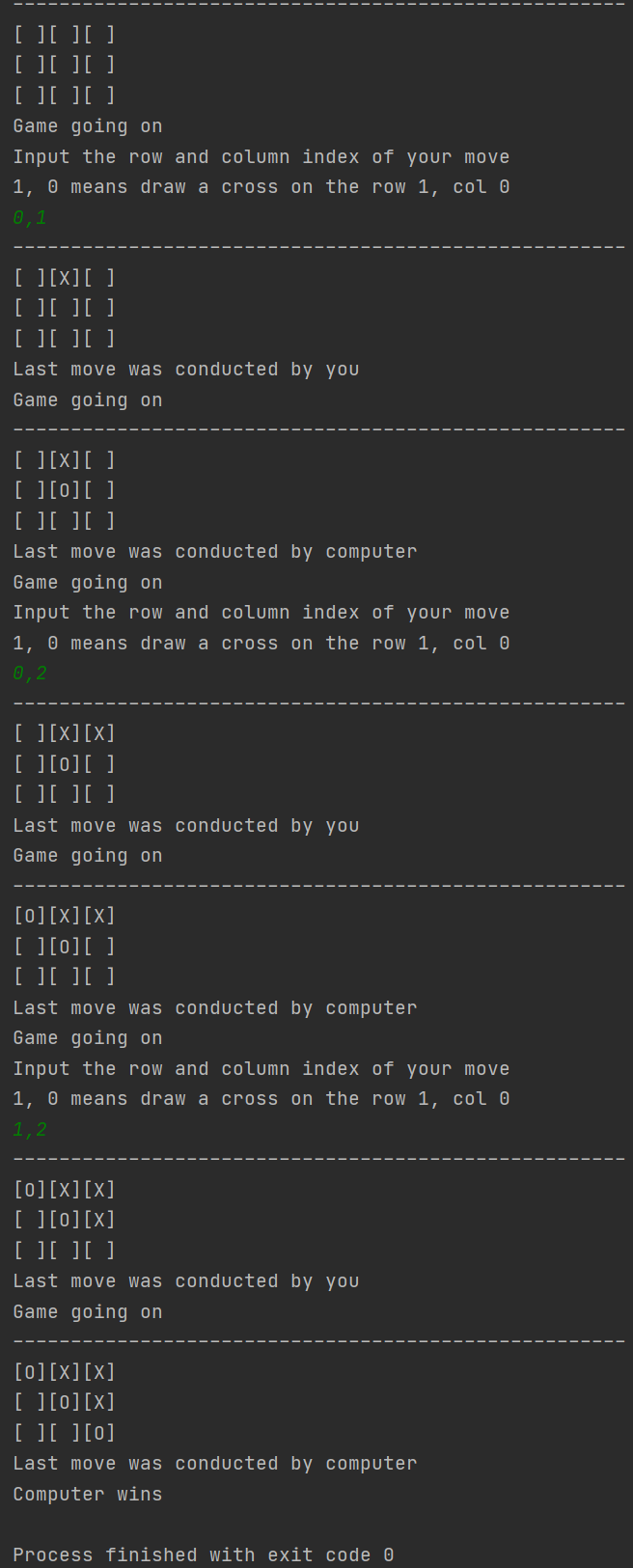


* 1. The results



Case 1: normal case:





Case 2:

deliberate to lose