Tic Tac Toe (GUI - Python)

S.Matoike

目次

第 1 章

Python による三目並べ

三目並べは「二人零和有限確定完全情報ゲーム」と呼ばれるゲームに分類されます．簡単に言うと「勝敗は偶然に左右されず，最善手を打てばどちらかが勝つまたは引き分けになるゲーム」のことです．他にもリバーシ，チェッカー，チェス，将棋に囲碁などもこのゲームに分類されます．

初めに三目並べというゲームについて定義をしておきます．

三目並べ: 3 × 3 の格子上に二人のプレイヤーが「O」「X」を配置し、自分のマークを先に 3 つ並べた方が勝ちです．

1.1　GUI 版三目並べ

新しいプロジェクトを開始することにしましょう

まずは、コンピュータが乱数で石を置いてくる単純な方法を実現します

1.1.1　Params.py

今回は、Params.py というファイルに、Params クラスと State クラスという、2 つのクラスを定義しておきます

ソースコード 1.1　Params.py

```
1  from enum import auto
2
3  class Params:
4      HUMAN_ID = auto()
5      MACHINE_ID = auto()
6      EMPTY_ID = auto()
7
8  class State:
9      GAME = auto()
10     MACHINE_WIN = auto()
11     HUMAN_WIN = auto()
12     DRAW = auto()
```

Params クラスには、盤面に置く石の識別情報、HUMAN_ID と MACHINE_ID、及び空きスロットを識別する EMPTY_ID の 3 つの識別子を持たせます

State クラスには、盤面の状態＝ゲームの進行状況を示す定数、即ち、GAME はゲーム進行中の状態、

HUMAN_WIN は人が勝った状態、MACHINE_WIN はコンピュータが勝った状態、DRAW は引き分けの状態を表す定数として保持させます

いずれのクラスも、auto() という関数を使って自動的に重複しない値を持たせることができています

auto() によって自動的に区別された値が割り振られるため、プログラム中で変数名を参照していく以上、それらの値そのものがどのような値なのかは知らなくても問題はありませんね

また、これらはクラス変数と呼ばれ、「クラス名. 変数名」という形で使うことができて、このクラスから導出されるオブジェクトでは共通に持っている変数になります

オブジェクト毎に持たせているオブジェクト変数「self. 変数名」とは別の変数ですので、区別して理解するようにしましょう

1.1.2　Screen クラス

GUI にするので、pygame で Window 画面を用意するための Screen クラスを作ります

ソースコード 1.2　class Screen

```
1  import pygame
2
3  class Screen:
4      def __init__(self, wh=(600,600), bgcolor=(0,0,0)):
5          self.WIDTH = wh[0]
6          self.HEIGHT = wh[1]
7          self.COLOR = bgcolor
8          self.SIZE = (self.WIDTH, self.HEIGHT)
9          self.surface = pygame.display.set_mode(self.SIZE)
10         self.SLOTW = self.WIDTH//3
11         self.SLOTH = self.HEIGHT//3
12
13     def fill(self):
14         self.surface.fill(self.COLOR)
15         BLACK = (0,0,0)
16         for i in range(1,3):
17             startp = (0, self.SLOTH*i)
18             endp = (self.WIDTH, self.SLOTH*i)
19             pygame.draw.line(self.surface, BLACK, startp, endp)
20             startp = (self.SLOTW*i, 0)
21             endp = (self.SLOTW*i, self.HEIGHT)
22             pygame.draw.line(self.surface, BLACK, startp, endp)
23
24     def caption(self, str):
25         pygame.display.set_caption(str)
```

コンストラクタへの引数で、盤面のサイズと背景色を受け取ることができますが、特に指定しない場合にはデフォルト値が用いられる様にしています

三目並べは 3 x 3 のスロットを持つ盤面ですので、そのための描画を fill() メソッドで行っています

caption() メソッドでは、引数で受け取った文字列を Window のタイトルバーに表示できるようにしています

1.1.3　Board クラス

Board クラスは、Screen クラスを継承させるようにしていますので、Screen クラスもプロパティやメソッドを、あたかも Board クラスで用意したかのように使うことができます

Board クラスのコンストラクタでは、引数で盤面のサイズ、背景色、キャプションに表示する文字列を受け取ることができるようにしていますが、時に指定しなかった場合にデフォルト値が使われるように記述しています

Board クラスのコンストラクタの冒頭で、Screen クラスのコンストラクタを呼び出して、Screen の初期化を実行し、キャプションの表示、そして盤面の実態である 3 x 3 = 9 つの EMPTY 要素を持つ board という名のリストを用意しています

また、ゲームの状態を self.state に持たせることとし、ゲーム進行中の状態 GAME を初期状態として代入しています

ソースコード 1.3　class Board

```
1  import pygame
2  from Screen import Screen
3  from Params import State,Params
4
5  class Board( Screen ):
6      def __init__(self, wh=(300,300), color=(200,200,0), str='Tic-Tac-Toe'):
7          super().__init__(wh, color)
8          self.caption(str)
9          self.board = [Params.EMPTY_ID for _ in range(9)]
10         self.state = State.GAME
11
12     def draw(self):
13         self.fill()
14         N = 3
15         RADIUS = int(self.SLOTW*0.2)
16         BLACK = (0,0,0)
17         for ypos in range(N):
18             yc = ypos * self.SLOTH + self.SLOTH // 2
19             for xpos in range(N):
20                 xc = xpos*self.SLOTW + self.SLOTW // 2
21                 i = ypos * N + xpos
22                 if self.board[i] == Params.HUMAN_ID:
23                     pygame.draw.circle(self.surface, BLACK, (xc,yc), RADIUS, 3)
24                 elif  self.board[i] == Params.MACHINE_ID:
25                     startp = (xc - RADIUS, yc - RADIUS)
26                     endp = (xc + RADIUS, yc + RADIUS)
27                     pygame.draw.line(self.surface, BLACK, startp, endp, 3)
28                     startp = (xc + RADIUS, yc - RADIUS)
29                     endp = (xc - RADIUS, yc + RADIUS)
30                     pygame.draw.line(self.surface, BLACK, startp, endp, 3)
31
32     def winner(self):
33         def scan(n1, n2, n3):
34             p = self.board[n1] != 0 and \
35                 self.board[n1] == self.board[n2] and \
36                 self.board[n1] == self.board[n3]
37             if p:
38                 if self.board[n1]==Params.HUMAN_ID:
```

```
39                      self.state = State.HUMAN_WIN
40                  elif self.board[n1]==Params.MACHINE_ID:
41                      self.state = State.MACHINE_WIN
42              else:
43                  if not self.vacant():
44                      self.state = State.DRAW
45                  else:
46                      self.state = State.GAME
47              return p
48          #
49          return scan(0, 1, 2) or scan(3, 4, 5) or scan(6, 7, 8) or \
50                 scan(0, 3, 6) or scan(1, 4, 7) or scan(2, 5, 8) or \
51                 scan(0, 4, 8) or scan(2, 4, 6)
52
53      def vacant(self):
54          empty = []
55          for n, slot in enumerate(self.board):
56              if slot == Params.EMPTY_ID:
57                  empty.append(n)
58          return empty
```

draw() メソッドは、盤面 board リストの状態を Window の Screen 上に描画しています

board リストの要素が HUMAN_ID と一致するなら draw.circle() で○を描き、MACHINE_ID と一致するなら draw.line() を使って×を描いています

winner() メソッドでは、ゲームの勝敗、引き分け、ゲーム進行中の判別をして、self.state プロパティにそれぞれの値を設定しています

winner() メソッドの中に持っているローカルな scan() 関数が受け取る 3 つのスリット番号を見て、board リストの該当するスロットに入っている石が同じ識別子かどうかを判定しています

同じだったら、その石が HUMAN_ID なのか MACHINE_ID なのかを判別して、勝者を self.state に設定します

違っていたら、石を置く場所が残っているかどうかを調べ、残っていないないなら引き分け、残っているようならゲーム継続の状態だとしています

scan() が返す値が True なら、winner の返す値も True になるので、このとき勝敗がついたという意味になります

vacant() メソッドは、CUI でも説明した空きスロットのリストを返すメソッドです

1.1.4　Machine クラス

Machine クラスの put_stone() メソッドでは、まず、空きスロットのリストを取得し、そのリスト内の要素を乱数で選んで、board リスト上のその位置に、MACHINE_ID を代入しています

True を返した場合は石を普通に置いた場合、False を返すのは盤面に空きがなかった場合です

ソースコード 1.4　class Machine

```
1  from random import randint
2  from Params import Params
3
4  class Machine():
5      def __init__(self, name):
6          self.name = name
```

```
7
8       def put_stone(self, board):
9           vacant = board.vacant()
10          if not vacant:
11              return False
12          n = randint(0, len(vacant)-1)
13          board.board[ vacant[n] ] = Params.MACHINE_ID
14          return True
```

1.1.5　Human クラス

Human クラスの put_stone() メソッドでは、まず空きスロットのリストを取得し、空きがないなら False で戻ります

空きがある場合は、put_stone() メソッドの中に、ローカルに定義した get_slot() 関数を呼びだし、その引数に、マウスがクリックした盤面の座標をタプルの形で渡します

get_slot() 関数は、board やスロットのサイズを元に、マウスがクリックしたのがどのスロット上なのかを求めて、そのスロット番号を返しています

戻されたスロット番号が、空きスロットのリストに含まれていたなら、board のその場所に HUMAN_ID を代入して True で戻ります

マウスでクリックしたスロット番号が、空きスロットリストに含まれていないなら、それは、既に他の石が置かれているスロットだと言うことになりますから、もう一度クリックでスロットを選び直して下さいという意味で、None を返しています

ソースコード 1.5　class Human

```
1   from Params import Params
2
3   class Human:
4       def __init__(self, name):
5           self.name = name
6
7       def put_stone(self, board, pos=(-1,-1)):
8           def get_slot(xy):
9               x, y = xy[0], xy[1]
10              N = 3
11              for ypos in range(N):
12                  y0 = board.SLOTH * ypos
13                  y1 = board.SLOTH * (ypos + 1)
14                  for xpos in range(N):
15                      x0 = board.SLOTW * xpos
16                      x1 = board.SLOTW * (xpos + 1)
17                      if y0 < y < y1 and x0 < x < x1:
18                          return ypos * N + xpos
19              return -1
20          vacant = board.vacant()
21          if not vacant:
22              return False
23          n = get_slot(pos)
24          if n in vacant:
25              board.board[n] = Params.HUMAN_ID
26          else:
27              return None
```

```
28          return True
```

1.1.6　Game クラス

Game クラスのコンストラクタでは、引数 turn に文字列を受け取り、それによって先手と後手を決めています

change_tuen() メソッドは、手番を交代します

fine() メソッドは、このアプリケーションを終了させるときのメソッドで、Window 上の QUIT のエベントを受け取ったときに呼び出されます

キーボードイベントの取得部分で、MOUSEBUTTONUP を拾っており、human オブジェクトの put_stone() メソッドにクリックした座標 event.pos を渡しています

judge() メソッドは、盤面の状態 board オブジェクトの state プロパティの値を見て、勝敗が決したのか、引き分けなのか、ゲーム思考の継続なのかを判定し、ゲーム進行中の時だけ True を、そうでない時は False を返しています

start() メソッドはゲームの進行そのもので、以下を繰り返しています

①キーイベントをチェックして HUMAN の手番ならこの中で HUMAN がスロットを選ぶことになります②盤面の状態を描画します③ MACHINE の手番なら、machine オブジェクトの put_stone() メソッドを呼び出しています④勝敗の判定をして、⑤画面を更新します

ソースコード 1.6　class Game

```
1  import sys
2  import pygame
3  from pygame.locals import QUIT, MOUSEBUTTONUP
4  from Board import Board
5  from Human import Human
6  from Machine import Machine
7  from Params import State
8
9  class Game:
10     def __init__(self, turn='human'):
11         pygame.init()
12         self.board = Board()
13         self.human = Human('Taro')
14         self.machine = Machine('Computer')
15         self.turn = False if turn=='human' else True
16         self.clock = pygame.time.Clock()
17         self.FPS = 10
18
19     def change_turn(self):
20         self.turn = not self.turn
21
22     def fine(self):
23         pygame.quit()
24         sys.exit()
25
26     def key_event(self):
27         for event in pygame.event.get():
28             if event.type == QUIT:
29                 self.fine()
30             elif event.type == MOUSEBUTTONUP:
```

```
31                 if not self.turn:
32                     p = self.human.put_stone(self.board, event.pos)
33                     if p:
34                         self.change_turn()
35                     elif p==False:
36                         pass     # DRAW
37                     else:
38                         pass
39 
40     def judge(self):
41         result = False
42         if self.board.winner():
43             if self.board.state == State.MACHINE_WIN:
44                 self.board.caption('Computer won the game!')
45             elif self.board.state == State.HUMAN_WIN:
46                 self.board.caption('You won!')
47         else:
48             if self.board.state == State.DRAW:
49                 self.board.caption('Draw!')
50             elif self.board.state == State.GAME:
51                 result = True
52         return result
53 
54     def start(self):
55         while True:
56             self.key_event()
57             self.board.draw()
58             if self.turn:
59                 if self.machine.put_stone(self.board):
60                     self.change_turn()
61             self.judge()
62             pygame.display.update()
63             self.clock.tick(self.FPS)
```

1.1.7　main.py

先手が誰なのかを、Game クラスのコンストラクタに指示しています

ソースコード 1.7　main.py

```
1 from Game import Game
2 
3 if __name__ == '__main__':
4     sente = 'human'
5     game = Game( turn=sente )
6     game.start()
```

1.2　MiniMax 法

ここまでコンピュータが採用してきた戦略は、「空いているスロットの中から乱数で選んだ場所に石を置く」という単純なものでした

ここでは、絶対に負けないコンピュータ、（人が最善の手をとった場合に引き分けたとしても負けることはありません）を実現していきます

1.2.1　main の変更

Game クラスのインスタンス game を生成する際、コンストラクタへの引数で、先手は'human' か'machine' か、コンピュータの採る戦略は乱数'random' か'minimax' かを指定するように直します

ソースコード 1.8　main.py

```
from Game import Game
from Params import Params

if __name__ == '__main__':
    sente = 'machine'           # 'machine' or 'human'
    senryaku = 'minimax'        # 'random' or 'minimax'
    game = Game( turn=sente, strategy=senryaku )
    game.start()
```

1.2.2　Game クラスの変更

Game クラスのコンストラクタにおいて、MACHINE が採る戦略を引数 strategy で受け取り、それを Machine クラスのコンストラクタに渡しています

ソースコード 1.9　Game.py

```
import sys
import pygame
from pygame.locals import QUIT, MOUSEBUTTONUP
from Board import Board
from Human import Human
from Machine import Machine
from Params import State

class Game:
    def __init__(self, turn='human', strategy='random'):
        pygame.init()
        self.board = Board()
        self.human = Human('Taro')
        self.machine = Machine('Computer', strategy)
        self.turn = False if turn=='human' else True
        self.clock = pygame.time.Clock()
        self.FPS = 10
    ・
    以下に変更なし
    ・
    ・
```

1.2.3　Board クラスの変更

次の 2 つのメソッドは、MiniMax 戦略で使うために追加します

can_put() メソッドは、引数で受け取った番号のスロットが空いているかどうかを返します

undo() メソッドは、一旦石を置いたスロットを、元の空きスロットに戻すメソッドです

デバッグのために用意したメソッドです（完成したら不要です）
debug_board() メソッドは、デバッグ用に現在の盤面の状態をファイル出力するメソッドです

ソースコード 1.10　Board.py

```
import pygame
from Screen import Screen
from Params import State,Params

class Board( Screen ):
    def __init__(self, wh=(300,300), color=(200,200,0), str='Tic-Tac-Toe'):
        ・
        変更なし
        ・
        ・
    def draw(self):
        ・
        変更なし
        ・
        ・
    def winner(self):
        ・
        変更なし
        ・
        ・
    def vacant(self):
        ・
        変更なし
        ・
        ・
    # 以下を追加
    def undo(self, n):
        self.board[n] = Params.EMPTY_ID

    def can_put(self, n):
        if self.board[n]==Params.EMPTY_ID:
            return True
        return False

    # ボードを表示する for Debug
    def debug_board(self, seq, debugstring):
        str0 = '\n('+str(seq)+')'
        tmp = []
        for i in range(9):
            if self.board[i] == Params.EMPTY_ID:
                tmp.append(' ')
            elif self.board[i] == Params.HUMAN_ID:
                tmp.append('o')
            elif self.board[i] == Params.MACHINE_ID:
                tmp.append('x')
        str1 =  '\n{0[0]}|{0[1]}|{0[2]}\t<-----\t0|1|2'\
                '\n{0[3]}|{0[4]}|{0[5]}\t<-----\t3|4|5'\
                '\n{0[6]}|{0[7]}|{0[8]}\t<-----\t6|7|8\n'.format(tmp)
        str2 = str0 + str1 + debugstring + '\n'
        with open('textfile.txt', 'a', encoding='utf-8') as f:
            f.write(str2)
```

1.2.4　Machine クラスの変更

Machine クラスのコンストラクタで、戦略が'random' か'minimax' かを受け取っています

Strategy クラスを継承しており、MiniMax の具体的な戦略のプログラムコードは上位クラスの Strategy に記述することにします

put_stone() メソッドでは、①空きスロットのリスト vacant を取得し、②空きがないなら③石を置けないので False で返ります。④ strategy が'random' だった場合、⑤乱数で vacant のインデックス n を選び、⑥ vacant の n 番目にあるスロット番号に石を置きます。⑦ strategy が'minimax' だった場合、⑧ MiniMax 法によるベストな手をスロット番号 n で受け取り、⑨そこに石を置きます

ソースコード 1.11　Machine.py

```
1  from random import randint
2  from Params import Params
3  from Strategy import Strategy
4
5  class Machine( Strategy ):
6      def __init__(self, name, strategy):
7          super().__init__()
8          self.name = name
9          self.strategy = strategy
10
11     def put_stone(self, board):
12         vacant = board.vacant()
13         if not vacant:
14             return False
15         if self.strategy=='random':
16             n = randint(0, len(vacant)-1)
17             board.board[ vacant[n] ] = Params.MACHINE_ID
18         elif self.strategy=='minimax':
19             n = self.bestMove(board, vacant)
20             board.board[ n ] = Params.MACHINE_ID
21         return True
```

1.2.5　Strategy クラスを追加

ここに MiniMax 戦略のコードをまとめて記述します

bestMove() メソッドでは、①引数で空のスロット番号のリスト vacant を受け取り、②そのリストの要素の全てについて③順に試して④その手を評価していきます（Maximizer である Machine が③で石を置いた直後ですから、次は Human 即ち Minimizer の手番ですよ、という意味で False を引数に指示しています）⑤高い評価値の得られたスロット番号は、⑥ bestMove の候補として更新していきます⑦③で選んだ手を元の空のスロットに戻して、③次の手を置いてみて④評価することを繰り返します

minimax() メソッドでは、①まず、board インスタンスの winner() メソッドを呼んで現在の盤面の状態が、ゲーム進行中（GAME）なのか、勝敗が決した（MACHINE_WIN か HUMAN_WIN）のか、引き分け（DRAW）だったのかを判定します②ゲーム進行中（GAME）でないなら、③評価関数であるローカルな evaluate() メソッドを呼びだし、結果を返します④ゲーム進行中ならば、⑤全てのスロット番号を順に選んで、⑥そのスロットが空で石を置けるかどうかを判定し、置けるならば、⑦そのスロットに石を

置きます（Maximizer=Machine=’ ×’ か、Minimizer=Human=’ ○’ かによって置く石は違います）⑧再帰的に minimax() メソッドを呼び出します（isMaximizingPlayer を not として引数に指示し、ターンを交互に割り当てるようにしています）⑨ bestVal を更新していきます。⑩石を置いた番号のスロットを空に戻します

minimax() メソッドの中に定義された評価関数の evaluate() は、ゲームが進行状態（GAME）でない場合に呼び出されるので、Machine が勝った場合（MACHINE_WIN）、Human が勝った場合（HUMAN_WIN）、引き分けだった場合（DRAW）に、それぞれ評価値を返しています

ソースコード 1.12　Strategy.py

```
1  from Params import Params, State
2
3  class Strategy:
4      INFINITY = 10000
5      def __init__(self):
6          pass
7
8      def bestMove(self, board, vacant):
9          bestEval = -Strategy.INFINITY
10         bestMove = -1
11         self.debug_seq = 0
12         for n in vacant:
13             board.board[n] = Params.MACHINE_ID
14             eval = self.minimax(board, 0, False)
15             if bestEval<eval:
16                 bestEval = eval
17                 bestMove = n
18             board.undo(n)
19         return bestMove
20
21     def minimax(self, board, depth, isMaximizingPlayer):
22         def evaluate(depth):
23             if board.state == State.MACHINE_WIN:
24                 board.state = State.GAME
25                 return 10-depth      #return 10
26             elif board.state == State.HUMAN_WIN:
27                 board.state = State.GAME
28                 return depth-10      #return -10
29             else:
30                 self.state = State.GAME
31                 return 0
32
33         board.winner()
34         if board.state != State.GAME:
35             return evaluate(depth)
36         bestVal = -Strategy.INFINITY if isMaximizingPlayer else +Strategy.INFINITY
37         for n in range(9):
38             if board.can_put(n):
39                 board.board[n] = Params.MACHINE_ID if isMaximizingPlayer else
                       Params.HUMAN_ID
40                 value = self.minimax(board, depth + 1, not isMaximizingPlayer)
41                 bestVal = max(value, bestVal) if isMaximizingPlayer else min(value,
                        bestVal)
42                 board.undo(n)
43         return bestVal
```

1.3　Alpha Bata Pruning

Alpha Beta Pruning は新しい手法のアルゴリズムではなく、MiniMax 法を最適化したものにすぎません

MiniMax 法は、勝敗が決着するまで全ての手を試行して、その上で最善手を選ぶため、引き分けることはあっても負けることはありません。

しかし、全ての手を試行するというのは、多くのゲームで現実的な方法にはなりません。三目並べの様な小規模のゲームでは、それほど問題になりませんが、リバーシなどになると思考時間がかかりすぎて実用的ではありません

そこで、MiniMax 法の試行を途中で合理的に打ち切る手法がアルファ刈りベータ刈りと呼ばれる手法です

1.3.1　main の変更

Game クラスのインスタンス game を生成する際、コンストラクタへの引数で、先手は'human' か'machine' か、コンピュータの採る戦略は乱数'random' か'minimax' か'alphabeta' の何れかを指定するようにします

ソースコード 1.13　main.py

```
1  from Game import Game
2  from Params import Params
3
4  if __name__ == '__main__':
5      sente = 'human'            # 'machine' or 'human'
6      senryaku = 'alphabeta'         # 'random', 'minimax' or 'alphabeta'
7      game = Game( turn=sente, strategy=senryaku )
8      game.start()
```

1.3.2　Machine クラスの変更

戦略として、alphabeta を選べるように直します

ソースコード 1.14　class Machine

```
1  from random import randint
2  from Params import Params
3  from Strategy import Strategy
4
5  class Machine( Strategy ):
6      def __init__(self, name, strategy):
7          super().__init__()
8          self.name = name
9          self.strategy = strategy
10
11     def put_stone(self, board):
12         vacant = board.vacant()
13         if not vacant:
```

```
14              return False
15          if self.strategy=='random':
16              n = randint(0, len(vacant)-1)
17              board.board[ vacant[n] ] = Params.MACHINE_ID
18          elif self.strategy=='minimax':
19              n = self.bestMove(board, vacant)
20              board.board[ n ] = Params.MACHINE_ID
21          elif self.strategy=='alphabeta':
22              n = self.bestMoveAB(board, vacant)
23              board.board[ n ] = Params.MACHINE_ID
24          return True
```

1.3.3　Strategy クラスの変更

具体的な戦略は、こちらのクラスに記述しています

ソースコード 1.15　class Strategy

```
1   from Params import Params, State
2
3   class Strategy:
4       INFINITY = 10000
5       def __init__(self):
6           pass
7
8       def bestMove(self, board, vacant):
9               ・
10              変更なし
11              ・
12              ・
13
14      def minimax(self, board, depth, isMaximizingPlayer):
15              ・
16              変更なし
17              ・
18              ・
19
20      def bestMoveAB(self, board, vacant):
21          bestEval = -Strategy.INFINITY
22          bestMove = -1
23          for n in vacant:
24              board.board[n] = Params.MACHINE_ID
25              eval = self.minimaxab(board, 0, 0, False, -Strategy.INFINITY, +Strategy
                    .INFINITY)
26              if bestEval<eval:
27                  bestEval = eval
28                  bestMove = n
29              board.undo(n)
30          return bestMove
31
32      def minimaxab(self, board, node, depth, isMaximizingPlayer, alpha, beta):
33          def evaluate(depth):
34              if board.state == State.MACHINE_WIN:
35                  board.state = State.GAME
36                  return 10-depth      #return 10
37              elif board.state == State.HUMAN_WIN:
38                  board.state = State.GAME
```

```
39                  return depth-10      #return -10
40              else:
41                  self.state = State.GAME
42                  return 0
43
44          board.winner()
45          if board.state != State.GAME:
46              return evaluate(depth)
47
48          bestVal = -Strategy.INFINITY if isMaximizingPlayer else +Strategy.INFINITY
49          for n in range(9):
50              if board.can_put(n):
51                  board.board[n] = Params.MACHINE_ID if isMaximizingPlayer else
                        Params.HUMAN_ID
52                  value = self.minimaxab(board, node+1, depth+1, not
                        isMaximizingPlayer, alpha, beta)
53                  board.undo(n)
54                  if isMaximizingPlayer:
55                      bestVal = max(value, bestVal)
56                      alpha = max(alpha, bestVal)
57                  else:
58                      bestVal = min(value, bestVal)
59                      beta = min(beta, bestVal)
60                  if beta<=alpha:
61                      #print('depth=',depth)
62                      break
63          return bestVal
```

参考文献

[1]