Othello (CUI - Python)

S.Matoike

目次

第 1 章

Othello(リバーシ)：N × N

8 × 8 サイズのゲームが一般的ですが、6 × 6 盤の縮小オセロでは必勝法が見つかっているようです

1993 年にイギリスの Joel Feinstein が、6 × 6 盤の縮小オセロで、先手が最善手を尽くしたとしても後手が「20 対 16」で勝つ、(先手 16 対、後手 20 で、後手の 4 目勝ち) という事を示したということです

ここでは、4 × 4「以上」の大きさの盤面に対応するオセロを作ります。あなたにゲームを最後まで続ける根気があるなら、10 × 10 でも 20 × 20 でもゲームすることができるかもしれません

1.1　CUI 版

以下のゲーム例では、×（BLACK）の手番ですが、○（WHITE）を裏返せないところに×を置こうとしたり、×（BLACK）を置ける場所があるのに pass を宣言したりすると、再入力を促されています

○（WHITE）の手番で、まずは乱数によってコンピュータが石を置くスロットを決めていますが、後ほど Strategy クラスで、コンピュータの強い戦略について考察していきます

```
WHITE:2, BLACK:2
  0 1 2 3 →x
0 . . . .
1 . X O .
2 . O X .
3 . . . .
↓
y
Your turn. [yx] or "pass": 23
Your turn. [yx] or "pass": pass
Your turn. [yx] or "pass": 13

WHITE:1, BLACK:4
  0 1 2 3 →x
0 . . . .
1 . X X X
2 . O X .
3 . . . .
↓
y
WHITE:3, BLACK:3
  0 1 2 3 →x
0 . . . .
1 . X X X
```

```
2 . O O O
3 . . . .
↓
y
Your turn. [yx] or "pass": 33

WHITE:1, BLACK:6
  0 1 2 3 →x
0 . . . .
1 . X X X
2 . O X X
3 . . . X
↓
y
WHITE:3, BLACK:5
  0 1 2 3 →x
0 . O . .
1 . O X X
2 . O X X
3 . . . X
↓
y
Your turn. [yx] or "pass": 00
```

（　・・　途中略　・・　）

```
WHITE:5, BLACK:9
  0 1 2 3 →x
0 X X X O
1 O X X X
2 O O X X
3 O . . X
↓
y
Your turn. [yx] or "pass": 31

WHITE:4, BLACK:11
  0 1 2 3 →x
0 X X X O
1 O X X X
2 O X X X
3 O X . X
↓
y
WHITE:7, BLACK:9
  0 1 2 3 →x
0 X X X O
1 O X X X
2 O O X X
3 O O O X
↓
y

Winner : BLACK
```

1.1.1　Game クラス

メインでは、Game クラスのコンストラクタでへの引数で盤面のサイズ（N=4 以上の偶数)、先手を'human', 'machine' から、戦略を'random', 'maxflip', 'minimax', 'alphabeta' から指定します

ソースコード 1.1　main

```
1  from Game import Game
2
3  if __name__ == '__main__':
4      sente = 'human'          # 'human', 'machine'
5      senryaku = 'maxflip'     # 'random', 'maxflip', 'minimax', 'alphabeta'
6      game = Game( N=6, turn=sente, strategy=senryaku )       # board size -> N x N
7      game.start()
```

Game クラスのコンストラクタで、盤面クラス（Board）のインスタンス（board)、人クラス（Human）のインスタンス（human)、コンピュータクラス（Machine）のインスタンス（machine）を生成し、インスタンス変数 self.player には、human と machine の 2 つのインスタンスを要素とするリストを用意しています

start() メソッドで、ゲームの進行の主要部分を記述しています

①盤面を印刷（board.print()）し、②プレーヤの打つ手を選び（player[turn].Te(self.board))、③もし、パス（Board.PASS）でなかった場合は、④盤面に石を置く（board.putStone(te,color)）ことになります。⑤パスならば盤面に石を置かずに、プレーヤを交代（turn = (turn+1)%2）します

⑥盤面の状況から勝敗の判定 (board.Winner()) を行い、戻り値が Board.GAME ならばゲームを継続、それ以外（BLACK_WIN か WHITE_WIN か DRAW）ならば、ゲームを終了して結果の勝敗を表示します

ソースコード 1.2　Game class

```
1  from Board import Board
2  from Human import Human
3  from Machine import Machine
4  from Stone import Stone
5
6  class Game():
7      def __init__(self, N=4, turn='human', strategy='random'):
8          self.board = Board(NW=N)
9          human = Human(color=Stone.BLACK)
10         machine = Machine(color=Stone.WHITE, strategy=strategy)
11         self.player = [human, machine]
12         self.turn = 0
13         if turn=='machine':
14             self.turn = 1
15
16     def start(self):
17         turn = self.turn
18         winner = Board.GAME
19         while winner==Board.GAME:
20             self.board.print()                    # ①
21
22             te = self.player[turn].Te(self.board) # ②
```

```
23              if te!=Board.PASS:                      # ③
24                  color = self.player[turn].color
25                  self.board.put_stone(te,color)      # ④
26
27              turn = (turn+1)%2                       # ⑤
28              winner = self.board.winner()            # ⑥
29
30          self.board.print()
31          print()
32          if winner==Board.BLACK_WIN:
33              print('Winner : BLACK')
34          elif winner==Board.WHITE_WIN:
35              print('Winner : WHITE')
36          elif winner==Board.DRAW:
37              print('DRAW')
```

1.1.2　Stone クラス

石のクラスでは、BLACK、WHITE、EMPTY の中の何れかの色特性（self.color）、及び盤面上の位置（self.locate）をプロパティとして持たせています

用意しているメソッドは、各プロパティに関するセッターやゲッターです

ソースコード 1.3　Stone class

```
1  class Stone():
2      EMPTY = 0
3      BLACK = 1
4      WHITE = 3 - BLACK
5      MACHINE_ID = WHITE
6      HUMAN_ID = BLACK
7      def __init__(self, te, color):
8          self.color = color
9          self.locate = te
10
11     def getLocate(self):
12         return self.locate
13
14     def setColor(self, color):
15         self.color = color
16
17     def getColor(self):
18         return self.color
19
20     def flipColor(self):
21         self.color = 3 - self.color
22
23     def getFigure(self):
24         if self.color==Stone.BLACK:
25             return 'X'
26         elif  self.color==Stone.WHITE:
27             return 'O'
28         return '.'
```

1.1.3　Board クラス

このクラスでは、盤面の状態をリスト（self.stones）に保持しています

コンストラクタでは、まず引数で示されたサイズ（NW * NW）の空の（Stone.EMPTY）盤面を用意しています

次に、盤面の中央に BLACK と WHITE の石をそれぞれ 2 個ずつ配置しますが、配置のパタンは乱数によって決めています

石の数を nWhite,nBlack,nEmpty に保持させ、winner() メソッドの最初でそれぞれの値を数えています

nWhite ＋ nBlack ＋ nEmpty は盤面のスロットの総数になりますから、nEmpty がゼロになった段階で、nWhite と nBlack の大小関係から勝敗を決定しています（nEmpty がゼロになるより前に勝敗の判定がつく場合について、具体的にプログラムする必要があるかもしれません）

print() メソッドは、画面に盤面の状態を表示しています

is_empty() メソッドは、引数で指定したスロットが空（Stone.EMPTY）であるか否かを判定しています

empty_list() メソッドは、現在の盤面において、空（Stone.EMPTY）のスロット番号の一覧をリストにして返します

can_put(te, color) メソッドは、引数 te で示されたスロット番号の場所に、第 2 引数の color で示された色の石を置けるかどうかを判定しています（具体的には、self.check_flip(Stone(te, color)) を呼び出して、色を反転できる石が何個かあるなら、そこには石を置けるという判断をしています）

check_flip(stone) メソッドでは、引数で受け取っている stone オブジェクトは、その石を置こうとしているスロットの位置番号、及びその石の色をプロパティとして持っていますから、盤面のその位置に指定された色の石を置いたときに、上下、左右、右斜め上、右斜め下、左斜め上、左斜め下の 8 つの方向で何個の石を反転できるかを数えています（最後に、反転できる石の総数を返しています）

flip_list(stone) メソッドでは、check_flip(stone) メソッドで数えた各方向での反転できる石の数に基づいて、反転する石のスロット番号のリストを作って返しています

put_stone(te, color) メソッドは、まずは、指定されたスロット位置に指定の石を置いた後に、反転する石のリストを flip_list(stone) メソッドで作って、そのリストを set_stones(flist,color) に渡して、実際に盤面の石を反転させるようにします

set_stones(flist,color) メソッドは、受け取ったリストを基に盤面のデータを更新し、石を反転させています

reset_stones() メソッドは、set_stones() メソッドで反転させた盤面上の石を元に戻します

puttable() メソッドは、can_put() メソッドが True だった場合の nKoma リストを取得します

puttable_list() メソッドは、puttbale() メソッドの戻り値から、盤面上で石を置ける場所のリストを取得します

ソースコード 1.4　Board class

```
1 from random import randint
2 from Stone import Stone
3
4 class Board:
```

```
5       PASS = -10
6       GAME =  -1
7       DRAW =   0
8       HUMAN_WIN = BLACK_WIN = Stone.BLACK * 100
9       MACHINE_WIN = WHITE_WIN = Stone.WHITE * 100
10      UNITV = ((0,-1),(1,-1),(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1))
11
12      def __init__(self, NW=4):
13          self.nKoma = [0 for _ in range(9)]
14          self.NW = NW
15          self.NxN = NW * NW
16          self.stones = [ Stone(i, Stone.EMPTY) for i in range(self.NxN) ]
17          m = NW//2
18          n = m - 1
19          self.stones[NW*n+n].setColor( Stone.WHITE )
20          self.stones[NW*n+m].setColor( Stone.BLACK )
21          self.stones[NW*m+n].setColor( Stone.BLACK )
22          self.stones[NW*m+m].setColor( Stone.WHITE )
23          nrnd = randint(0,1)
24          if nrnd==0:
25              self.stones[NW * n + n].flipColor()
26              self.stones[NW * n + m].flipColor()
27              self.stones[NW * m + n].flipColor()
28              self.stones[NW * m + m].flipColor()
29          self.nBlack = self.nWhite = 2
30          self.nEmpty = self.NxN - (self.nBlack + self.nWhite)
31          self.state = Board.GAME
32
33      def winner(self):
34          self.nBlack = self.nWhite = self.nEmpty = 0
35          for i in range(self.NxN):
36              if self.stones[i].getColor() == Stone.HUMAN_ID:
37                  self.nBlack += 1
38              elif self.stones[i].getColor() == Stone.MACHINE_ID:
39                  self.nWhite += 1
40              else:
41                  self.nEmpty += 1
42          if self.nBlack+self.nWhite == self.NxN:
43              if self.nBlack < self.nWhite:
44                  return Board.MACHINE_WIN
45              elif self.nBlack > self.nWhite:
46                  return Board.HUMAN_WIN
47              else:
48                  return Board.DRAW
49          return Board.GAME
50
51      def print(self):
52          print(f"WHITE:{self.nWhite}, BLACK:{self.nBlack}")
53          print(' ',end=' ')
54          for x in range(self.NW):
55              print(f"{x}", end=' ')
56          print("→x")
57          for y in range(self.NW):
58              print(f"{y:}", end=' ')
59              for x in range(self.NW):
60                  print(f"{self.stones[ y*self.NW+x ].getFigure()}", end=' ')
61              print()
62          print("↓\ny")
63
```

```
64      def check_flip(self, stone):
65          y = stone.getLocate()//self.NW
66          x = stone.getLocate()%self.NW
67          self.nKoma[8] = 0
68          if self.stones[y*self.NW+x].getColor() == Stone.EMPTY:
69              for i1 in range( len(self.nKoma)-1 ):   # 0,1,2,3,4,5,6,7
70                  m1, m2 = x, y
71                  self.nKoma[i1] = s = ct = 0
72                  while s<2:
73                      m1 += Board.UNITV[i1][0]
74                      m2 += Board.UNITV[i1][1]
75                      if (0<=m1<self.NW) and (0<=m2<self.NW):
76                          color = self.stones[m2*self.NW+m1].getColor()
77                          if color==Stone.EMPTY:
78                              s = 3
79                          elif color==stone.getColor():
80                              s=2 if s==1 else 3
81                          else:
82                              s=1
83                              ct += 1
84                      else:
85                          s=3
86                  if s==2:
87                      self.nKoma[8] += ct
88                      self.nKoma[i1] = ct
89          return self.nKoma[8]
90
91      def flip_list(self, stone):
92          self.check_flip(stone)
93          y = stone.getLocate()//self.NW
94          x = stone.getLocate()%self.NW
95          flipped = []
96          for i1 in range( len(self.nKoma)-1 ):   # 0,1,2,3,4,5,6,7
97              m1, m2 = x, y
98              for i2 in range( self.nKoma[i1] ):
99                  m1 += Board.UNITV[i1][0]
100                 m2 += Board.UNITV[i1][1]
101                 if (0 <= m1 < self.NW) and (0 <= m2 < self.NW):
102                     z = m2*self.NW+m1
103                     self.stones[z].setColor( stone.getColor() )
104                     flipped.append( z )
105         z = y*self.NW+x
106         self.stones[z].setColor( stone.getColor() )
107         flipped.append(z)
108         return flipped
109
110     def empty_list(self):
111         list = [v for v in range(self.NxN) if self.is_empty(v)]
112         return list
113
114     def is_empty(self, te):
115         return True if self.stones[te].getColor()==Stone.EMPTY else False
116
117     def can_put(self, te, color):
118         if 0<=te<self.NxN:
119             if 0<self.check_flip( Stone(te, color) ):
120                 return True
121         return False
122
```

```
123     def puttable(self, te, color):
124         if self.can_put(te, color):
125             return self.nKoma
126 
127     def puttable_list(self, color):
128         emptyl = self.empty_list()
129         komal = []
130         telist = []
131         for n in emptyl:
132             w = self.puttable(n, color)
133             if w:
134                 telist.append(n)
135                 komal.append(w)
136         return telist, komal
137 
138     def put_stone(self, te, color):
139         stone = Stone(te, color)
140         flist = self.flip_list( stone )
141         self.set_stones(flist, color)
142         self.stones[te] = stone
143         if color==Stone.BLACK:
144             self.nBlack = len(flist) + 1
145         else:
146             self.nWhite = len(flist) + 1
147         return flist
148 
149     def set_stones(self, flist, color):
150         for z in flist:
151             self.stones[z] = Stone(z, color)
152         self.stones[z] = Stone(z, Stone.EMPTY)
153 
154     def reset_stones(self, te, flist, color):
155         if color==Stone.WHITE:
156             c = Stone.BLACK
157         elif color==Stone.BLACK:
158             c = Stone.WHITE
159         self.stones[te] = Stone(te,c)
160         for z in flist:
161             self.stones[z] = Stone(z, c)
162         self.stones[z] = Stone(z, Stone.EMPTY)
163 
164     def debug_print(self):
165         for y in range(self.NW):
166             for x in range(self.NW):
167                 n = y*self.NW + x
168                 print(self.stones[n].color, end=' ')
169             print()
170         print('\n')
```

1.1.4　Human クラス

人の打つ石の手を決めているのは、このクラスのメソッド Te() です

プロンプトメッセージ（'Your turn. [yx] or "pass": '）を表示して、入力文字列を受け取ります（instr）

入力文字列（instr）の中に文字列'pass' が含まれていたなら、まず、石を置ける場所があったにもかかわらず、誤ってパスを指示したかもしれない可能性をチェック（puttable()）します

もし、石を置く場所がある局面なら、パスをしてはいけないので、プロンプトメッセージまで処理を戻します（continue）

石を置く場所がなくてパスの指示をしたのなら、パスの指示を返します（return Board.PASS）

入力文字列にスロット番号として、行番号ｙと列番号ｘが [yx] として指定されたならば、ｙｘという 2 桁の数字文字を、それぞれ整数に変換（serial_num()）しています

ｙとｘが数字から整数値に変換できたなら、スロットの位置番号を計算（slot = cy * board.NW + cx）できます

計算したスロット番号に、その石を置けるかどうかをチェック（board.can_put(slot, self.color)）して、置けない場合は最初の文字列入力プロンプトまで処理を戻します

チェックした結果、石を置くことのできるスロット番号だったなら、反復（while True）を抜けて（break）、そのスロット番号を返します（return slot）

ソースコード 1.5　Human class

```
1  from Board import Board
2  from Stone import Stone
3
4  class Human:
5      def __init__(self, color=Stone.BLACK):
6          self.color = color
7
8      def Te(self, board):
9          def puttable():
10             elist =  board.empty_list()
11             for v in elist:
12                 if board.can_put(v, self.color):
13                     return True
14             return False
15
16         while True:
17             instr = input('Your turn. [yx] or "pass": ')
18             if 'pass' in instr:
19                 if puttable():
20                     continue
21                 return Board.PASS
22             if 1<len(instr) and self.numP( instr[0] ) and self.numP( instr[1] ):
23                 cy = self.serial_num( instr[0] )
24                 cx = self.serial_num( instr[1] )
25                 slot = cy * board.NW + cx
26                 if board.can_put(slot, self.color):
27                     break
28         print()
29         return slot
30
31     def numP(self, xy):
32         if '0' <= xy <= '9':
33             return True
34         return False
35
36     def serial_num(self, xy):
37         def fromA(xy):
38             return int(xy - 'a')
39
40         if self.numP(xy):
```

```
41              nxy = int( xy )
42          else:
43              nxy = fromA(xy)+10
44          return nxy
```

1.1.5　Machine クラス

コンピュータ（Machine）の打つ手を決めているのが、このクラスのメソッド、Te() です

①まず最初に、board.puttable_list() によって石を置ける場所の一覧を telist というリストとして取得します

②打つ手のリストの要素がない場合は、Board.PASS を返します

③打つ手のリストの要素が一つしかない場合は、迷うことなくその手を返します

④'random' 戦略が選ばれている場合は、telist の中から乱数で選んだ手を返します

⑤'maxflip' 戦略が選ばれている場合は、Board クラスの nKoma リストの一番最後の要素に、反転できる石の数の総数が入っているので、最も多く反転できる手を返します

⑥'minimax' 戦略が選ばれている場合は、MINIMAX 法による手が選ばれますが、MINIMAX 法は、勝敗がつくまで、最後まで対戦を試行するので、盤面のサイズが最小値（N=4）の場合でも、とても応答に時間がかかります。

⑦'alphabeta' 戦略が選ばれている場合は、MINIMAX 法を途中で打ち切る手立てを含んでいるので、多少高速な応答を期待できますが、、、、、

ソースコード 1.6　Machine class

```
1  from random import randint
2  from Board import Board
3  from Stone import Stone
4  from Strategy import Strategy
5
6  class Machine( Strategy ):
7      def __init__(self, color=Stone.WHITE, strategy = 'random'):
8          self.color = color
9          self.strategy = strategy
10
11     def Te(self, board):
12         telist, komalist = board.puttable_list( self.color )
13         if not telist:
14             return Board.PASS
15         if len(telist)==1:
16             return telist[0]
17         if self.strategy == 'random':
18             n = randint( 0, len(telist)-1 )
19             return telist[n]
20         elif self.strategy == 'maxflip':
21             n = v = -1
22             for i, k in enumerate(komalist):
23                 if n < k[-1]:
24                     n = k[-1]
25                     v = telist[i]
26             return v
27         elif self.strategy == 'minimax':
28             print('thinking ...')
```

```
29              n = self.bestMove(board, telist)
30              print('Your turn.')
31              return n
32          elif self.strategy == 'alphabeta':
33              n = self.bestMoveAB(board, telist)
34              return n
```

1.1.6　Strategy クラス

このクラスは、Machine クラスのスーパークラスとして、Machine クラスに継承させて使うようにします

ここには、MINIMAX 法と Alpha-Beta 刈りの 2 つを記述しています

ソースコード 1.7　Strategy class

```
1  from Board import Board
2  from Stone import Stone
3
4  class Strategy:
5      INFINITY = 100000
6      def __init__(self):
7          pass
8
9      # MiniMax
10     def bestMove(self, board, telist):
11         bestEval = -Strategy.INFINITY
12         bestMove = -1
13         for n, v in enumerate(telist):
14             flist = board.put_stone(v, self.color)
15             eval = self.minimax(board, 0, False)
16             if bestEval<eval:
17                 bestEval = eval
18                 bestMove = v
19             board.reset_stones(v, flist, self.color)
20         return bestMove
21
22     def minimax(self, board, depth, isMaximizingPlayer):
23         def evaluate(depth):
24             if board.state == Board.MACHINE_WIN:
25                 board.state = Board.GAME
26                 return board.NxN-depth      #return NxN
27             elif board.state == Board.HUMAN_WIN:
28                 board.state = Board.GAME
29                 return depth-board.NxN      #return -NxN
30             else:
31                 self.state = Board.GAME
32                 return 0
33
34         board.winner()
35         if board.state != Board.GAME:
36             return evaluate(depth)
37
38         color = Stone.MACHINE_ID if isMaximizingPlayer else Stone.HUMAN_ID
39         telist, _ = board.puttable_list(color)
40         bestVal = -Strategy.INFINITY if isMaximizingPlayer else +Strategy.INFINITY
41         for n, v in enumerate(telist):
```

```
42              flist = board.put_stone(v, color)
43              value = self.minimax(board, depth + 1, not isMaximizingPlayer)
44              bestVal = max(value, bestVal) if isMaximizingPlayer else min(value,
                    bestVal)
45              board.reset_stones(v, flist, color)
46          return bestVal
47
48      # Alpha-Beta
49      def bestMoveAB(self, board, telist):
50          bestEval = -Strategy.INFINITY
51          bestMove = -1
52          for n, v in enumerate(telist):
53              flist = board.put_stone(v, self.color)
54              eval = self.minimaxab(board, 8, 0, False, -Strategy.INFINITY, +Strategy
                    .INFINITY)
55              if bestEval<eval:
56                  bestEval = eval
57                  bestMove = v
58              board.reset_stones(v, flist, self.color)
59          return bestMove
60
61      def minimaxab(self, board, node, depth, isMaximizingPlayer, alpha, beta):
62          def evaluate(depth):
63              if board.state == Board.MACHINE_WIN:
64                  board.state = Board.GAME
65                  return board.NxN-depth      #return NxN
66              elif board.state == Board.HUMAN_WIN:
67                  board.state = Board.GAME
68                  return depth-board.NxN      #return -NxN
69              else:
70                  self.state = Board.GAME
71                  return 0
72
73          board.winner()
74          if board.state != Board.GAME:
75              return evaluate(depth)
76          elif node==0:
77              n = board.nBlack - board.nWhite
78              return n if isMaximizingPlayer else -n
79
80          color = Stone.MACHINE_ID if isMaximizingPlayer else Stone.HUMAN_ID
81          telist, _ = board.puttable_list(color)
82          bestVal = -Strategy.INFINITY if isMaximizingPlayer else +Strategy.INFINITY
83          for n, v in enumerate(telist):
84              flist = board.put_stone(v, color)
85              value = self.minimaxab(board, node-1, depth+1, not isMaximizingPlayer,
                    alpha, beta)
86              board.reset_stones(v, flist, color)
87              if isMaximizingPlayer:
88                  bestVal = max(value, bestVal)
89                  alpha = max(alpha, bestVal)
90              else:
91                  bestVal = min(value, bestVal)
92                  beta = min(beta, bestVal)
93              if beta<=alpha:
94                  break
95          return bestVal
```

参考文献

[1]