```
 1 # connect4.py
 2 # ---------
 3 # Licensing Information:  You are free to use or
   extend these projects for
 4 # educational purposes provided that (1) you do not
   distribute or publish
 5 # solutions, (2) you retain this notice, and (3) you
   provide clear
 6 # attribution to Clemson University and the authors.
 7 #
 8 # Authors: Pei Xu (peix@g.clemson.edu) and Ioannis
   Karamouzas (ioannis@g.clemson.edu)
 9 #
10
11 """
12 In this assignment, the task is to implement the
   minimax algorithm with depth
13 limit for a Connect-4 game.
14
15 To complete the assignment, you must finish these
   functions:
16     minimax (line 196), alphabeta (line 237), and
   expectimax (line 280)
17 in this file.
18
19 In the Connect-4 game, two players place discs in a 6
   -by-7 board one by one.
20 The discs will fall straight down and occupy the
   lowest available space of
21 the chosen column. The player wins if four of his or
   her discs are connected
22 in a line horizontally, vertically or diagonally.
23 See https://en.wikipedia.org/wiki/Connect_Four for
   more details about the game.
24
25 A Board() class is provided to simulate the game
   board.
26 It has the following properties:
27     b.rows          # number of rows of the game
   board
28     b.cols          # number of columns of the game
```

```
28 board
29     b.PLAYER1       # an integer flag to represent
   the player 1
30     b.PLAYER2       # an integer flag to represent
   the player 2
31     b.EMPTY_SLOT    # an integer flag to represent an
    empty slot in the board;
32
33 and the following methods:
34     b.terminal()            # check if the game is
   terminal
35                             # terminal means draw or
   someone wins
36
37     b.has_draw()            # check if the game is a
   draw
38
39     w = b.who_wins()        # return the winner of
   the game or None if there
40                             # is no winner yet
41                             # w should be in [b.
   PLAYER1,b.PLAYER2, None]
42
43     b.occupied(row, col)    # check if the slot at
   the specific location is
44                             # occupied
45
46     x = b.get(row, col)     # get the player
   occupying the given slot
47                             # x should be in [b.
   PLAYER1, b.PLAYER2, b.EMPTY_SLOT]
48
49     row = b.row(r)          # get the specific row of
    the game described using
50                             # b.PLAYER1, b.PLAYER2
   and b.EMPTY_SLOT
51
52     col = b.column(r)       # get a specific column
   of the game board
53
54     b.placeable(col)        # check if a disc can be
```

```
54 placed at the specific
55                                # column
56
57     b.place(player, col)    # place a disc at the
   specific column for player
58         # raise ValueError if the specific column
   does not have available space
59
60     new_board = b.clone()   # return a new board
   instance having the same
61                                # disc placement with b
62
63     str = b.dump()          # a string to describe
   the game board using
64                                # b.PLAYER1, b.PLAYER2
   and b.EMPTY_SLOT
65 Hints:
66     1. Depth-limited Search
67         We use depth-limited search in the current
   code. That is we
68     stop the search forcefully, and perform
   evaluation directly not only when a
69     terminal state is reached but also when the
   search reaches the specified
70     depth.
71     2. Game State
72         Three elements decide the game state. The
   current board state, the
73     player that needs to take an action (place a disc
   ), and the current search
74     depth (remaining depth).
75     3. Evaluation Target
76         The minimax algorithm always considers that
   the adversary tries to
77     minimize the score of the max player, for whom
   the algorithm is called
78     initially. The adversary never considers its own
   score at all during this
79     process. Therefore, when evaluating nodes, the
   target should always be
80     the max player.
```

```
    4. Search Result
        The pesudo code provided in the slides only returns the best utility value.
    However, in practice, we need to select the action that is associated with this
    value. Here, such action is specified as the column in which a disc should be
    placed for the max player. Therefore, for each search algorithm, you should
    consider all valid actions for the max player, and return the one that leads
    to the best value.

"""

# use math library if needed
import math

def get_child_boards(player, board):
    """
    Generate a list of successor boards obtained by placing a disc
    at the given board for a given player

    Parameters
    ----------
    player: board.PLAYER1 or board.PLAYER2
        the player that will place a disc on the board
    board: the current board instance

    Returns
    -------
    a list of (col, new_board) tuples,
    where col is the column in which a new disc is placed (left column has a 0 index),
    and new_board is the resulting board instance
    """
    res = []
    for c in range(board.cols):
        if board.placeable(c):
```

```
114             tmp_board = board.clone()
115             tmp_board.place(player, c)
116             res.append((c, tmp_board))
117     return res
118 
119 
120 def evaluate(player, board):
121     """
122     This is a function to evaluate the advantage of
    the specific player at the
123     given game board.
124 
125     Parameters
126     ----------
127     player: board.PLAYER1 or board.PLAYER2
128         the specific player
129     board: the board instance
130 
131     Returns
132     -------
133     score: float
134         a scalar to evaluate the advantage of the
    specific player at the given
135         game board
136     """
137     adversary = board.PLAYER2 if player == board.
    PLAYER1 else board.PLAYER1
138     # Initialize the value of scores
139     # [s0, s1, s2, s3, --s4--]
140     # s0 for the case where all slots are empty in a
     4-slot segment
141     # s1 for the case where the player occupies one
    slot in a 4-slot line, the rest are empty
142     # s2 for two slots occupied
143     # s3 for three
144     # s4 for four
145     score = [0]*5
146     adv_score = [0]*5
147 
148     # Initialize the weights
149     # [w0, w1, w2, w3, --w4--]
```

```
150     # w0 for s0, w1 for s1, w2 for s2, w3 for s3
151     # w4 for s4
152     weights = [0, 1, 4, 16, 1000]
153 
154     # Obtain all 4-slot segments on the board
155     seg = []
156     invalid_slot = -1
157     left_revolved = [
158         [invalid_slot]*r + board.row(r) + \
159         [invalid_slot]*(board.rows-1-r) for r in
    range(board.rows)
160     ]
161     right_revolved = [
162         [invalid_slot]*(board.rows-1-r) + board.row(
    r) + \
163         [invalid_slot]*r for r in range(board.rows)
164     ]
165     for r in range(board.rows):
166         # row
167         row = board.row(r)
168         for c in range(board.cols-3):
169             seg.append(row[c:c+4])
170     for c in range(board.cols):
171         # col
172         col = board.col(c)
173         for r in range(board.rows-3):
174             seg.append(col[r:r+4])
175     for c in zip(*left_revolved):
176         # slash
177         for r in range(board.rows-3):
178             seg.append(c[r:r+4])
179     for c in zip(*right_revolved):
180         # backslash
181         for r in range(board.rows-3):
182             seg.append(c[r:r+4])
183     # compute score
184     for s in seg:
185         if invalid_slot in s:
186             continue
187         if adversary not in s:
188             score[s.count(player)] += 1
```

```
189         if player not in s:
190             adv_score[s.count(adversary)] += 1
191     reward = sum([s*w for s, w in zip(score, weights
    )])
192     penalty = sum([s*w for s, w in zip(adv_score,
    weights)])
193     return reward - penalty
194 
195 #Depth limited minimax search
196 
197 def minimax(player, board, depth_limit):
198 
199     """
200     Minimax algorithm with limited search depth.
201 
202     Parameters
203     ----------
204     player: board.PLAYER1 or board.PLAYER2
205         the player that needs to take an action (
    place a disc in the game)
206     board: the current game board instance
207     depth_limit: int
208         the tree depth that the search algorithm
    needs to go further before stopping
209     max_player: boolean
210 
211     Returns
212     -------
213     placement: int or None
214         the column in which a disc should be placed
    for the specific player
215         (counted from the most left as 0)
216         None to give up the game
217     """
218     max_player = player
219     next_player = board.PLAYER2 if player == board.
    PLAYER1 else board.PLAYER1
220     placement = None
221 
222 ### Please finish the code below
     ##############################################
```

```
223 ##################################################
    ############################
224     def value(player, board, depth_limit):
225         if board.terminal() or depth_limit == 0:
226             return (evaluate(player, board), None)
227         elif player == max_player:
228             return max_value(player, board,
    depth_limit)
229         elif player == next_player:
230             return min_value(player, board,
    depth_limit)
231 
232     def max_value(player, board, depth_limit):
233         l = []
234         for c, b in get_child_boards(player, board):
235             v, a = value(next_player, b, depth_limit
    -1)
236             l.append((v, c))
237         return max(l)
238 
239     def min_value(player, board, depth_limit):
240         l = []
241         for c, b in get_child_boards(player,board):
242             v, a = value(max_player, b, depth_limit-
    1)
243             l.append((v, c))
244         return min(l)
245 
246     score, placement = value(max_player, board,
    depth_limit)
247 
248 ##################################################
    ############################
249     return placement
250 
251 #Alpha beta pruning of minimax search for efficiency
252 def alphabeta(player, board, depth_limit):
253     """
254     Minimax algorithm with alpha-beta pruning.
255 
256      Parameters
```

```
257     ----------
258     player: board.PLAYER1 or board.PLAYER2
259         the player that needs to take an action (
    place a disc in the game)
260     board: the current game board instance
261     depth_limit: int
262         the tree depth that the search algorithm
    needs to go further before stopping
263     alpha: float
264     beta: float
265     max_player: boolean
266 
267 
268     Returns
269     -------
270     placement: int or None
271         the column in which a disc should be placed
    for the specific player
272         (counted from the most left as 0)
273         None to give up the game
274     """
275     max_player = player
276     next_player = board.PLAYER2 if player == board.
    PLAYER1 else board.PLAYER1
277     alpha = -math.inf
278     beta = math.inf
279     placement = None
280 
281 ### Please finish the code below
     ##############################################
282 ##########################################################
    ############################
283     def value(player, board, depth_limit, alpha,
    beta):
284         if board.terminal():
285             return (evaluate(player, board), None)
286         elif depth_limit == 0:
287             return (evaluate(player, board), None)
288         elif player == max_player:
289             return max_value(player, board,
    depth_limit, alpha, beta)
```

```
290         elif player == next_player:
291             return min_value(player, board,
    depth_limit, alpha, beta)
292 
293     def max_value(player, board, depth_limit, alpha
    , beta):
294         l = []
295         for c, b in get_child_boards(player, board):
296             v, a = value(next_player, b, depth_limit
    -1, alpha, beta)
297             if v >= beta:
298                 return (v, c)
299             l.append((v, c))
300             alpha = max(alpha, v)
301         return max(l)
302 
303     def min_value(player, board, depth_limit, alpha
    , beta):
304         l = []
305         for c, b in get_child_boards(player,board):
306             v, a = value(max_player, b, depth_limit-
    1, alpha, beta)
307             if v <= alpha:
308                 return (v, c)
309             l.append((v, c))
310             beta = min(beta, v)
311         return min(l)
312 
313     score, placement = value(max_player, board,
    depth_limit, -math.inf, math.inf)
314 
315 ##########################################################
    ############################
316     return placement
317 
318 #Expectimax search with uniform chance of selecting
    available actions
319 def expectimax(player, board, depth_limit):
320     """
321     Expectimax algorithm.
322     We assume that the adversary of the initial
```

```
322 player chooses actions
323     uniformly at random.
324     Say that it is the turn for Player 1 when the
    function is called initially,
325     then, during search, Player 2 is assumed to pick
     actions uniformly at
326     random.
327
328     Parameters
329     ----------
330     player: board.PLAYER1 or board.PLAYER2
331         the player that needs to take an action (
    place a disc in the game)
332     board: the current game board instance
333     depth_limit: int
334         the tree depth that the search algorithm
    needs to go before stopping
335     max_player: boolean
336
337     Returns
338     -------
339     placement: int or None
340         the column in which a disc should be placed
    for the specific player
341         (counted from the most left as 0)
342         None to give up the game
343     """
344     max_player = player
345     next_player = board.PLAYER2 if player == board.
    PLAYER1 else board.PLAYER1
346     placement = None
347
348     ### Please finish the code below
     ###############################################
349
    ###############################################
    ############################
350     def value(player, board, depth_limit):
351         if board.terminal() or depth_limit == 0:
352             return (evaluate(player, board), None)
353         elif player == max_player:
```

```
354             return max_value(player, board,
    depth_limit)
355         elif player == next_player:
356             return exp_value(player, board,
    depth_limit)
357 
358     def prob_action(player, board):
359         possible_Actions = 0
360         for c in range(board.cols):
361             if board.placeable(c):
362                 possible_Actions += 1
363         return (1/possible_Actions)
364 
365     def max_value(player, board, depth_limit):
366         l = []
367         for c, b in get_child_boards(player, board):
368             v, a = value(next_player, b, depth_limit
     - 1)
369             l.append((v, c))
370         return max(l)
371 
372     def exp_value(player, board, depth_limit):
373         l = 0
374         for c, b in get_child_boards(player, board):
375             v, a = value(max_player, b, depth_limit
     - 1)
376             l = prob_action(player, board)*v + l
377         return (l, None)
378 
379     score, placement = value(max_player, board,
    depth_limit)
380 
381 
    ##################################################
    ############################
382     return placement
383 
384 if __name__ == "__main__":
385     from utils.app import App
386     import tkinter
387 
```

```
388     algs = {
389         "Minimax": minimax,
390         "Alpha-beta pruning": alphabeta,
391         "Expectimax": expectimax
392     }
393 
394     root = tkinter.Tk()
395     App(algs, root)
396     root.mainloop()
397 
```