

## PROJECT

## Build a Game-Playing Agent

A part of the Artificial Intelligence Nanodegree Program

## PROJECT REVIEW

## CODE REVIEW 3

## NOTES

## ▼ game\_agent.py 3

```
1 """Finish all TODO items in this file to complete the isolation project, then
2 test your agent's strength against a set of known agents using tournament.py
3 and include the results in your report.
4 """
5 import random
6
7
8 class SearchTimeout(Exception):
9     """Subclass base exception for code clarity. """
10     pass
11
12
13 def custom_score(game, player):
14     """Calculates the heuristic value of a game state from the point of view
15     of the given player.
16
17     This function is an improvement on the AB_Improved heuristic. In the
18     beginning of the game, it tries aggressively to reduce the number of opponent
19     moves. But, as the game goes on, it becomes increasingly aggressive at
20     maximizing its own moves.
21
22     Parameters
23     -----
24     game : `isolation.Board`
25         An instance of `isolation.Board` encoding the current state of the
26         game (e.g., player locations and blocked cells).
27
28     player : object
29         A player instance in the current game (i.e., an object corresponding to
30         one of the player objects `game.__player_1__` or `game.__player_2__`.)
31
32     Returns
33     -----
34     float
35         The heuristic value of the current game state to the specified player.
36     """
37     if game.is_loser(player):
38         return float("-inf")
39
40     if game.is_winner(player):
41         return float("inf")
42
43     # get current move count
44     move_count = game.move_count
45
46     # count number of moves available
47     own_moves = len(game.get_legal_moves(player))
48     opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
49
50     # calculate weight
51     w = 10 / (move_count + 1)
52
53     # return weighted delta of available moves
54     return float(own_moves - (w * opp_moves))
```

AWESOME

Good! Remarkable implementation.

```

55
56
57 def custom_score_2(game, player):
58     """Calculates the heuristic value of a game state from the point of view
59     of the given player.
60
61     This function seeks to increase the number of spaces the active player can
62     reach within two moves, while decreasing this number for the opponent.
63
64     Parameters
65     -----
66     game : `isolation.Board`
67         An instance of `isolation.Board` encoding the current state of the
68         game (e.g., player locations and blocked cells).
69
70     player : object
71         A player instance in the current game (i.e., an object corresponding to
72         one of the player objects `game.__player_1__` or `game.__player_2__`.)
73
74     Returns
75     -----
76     float
77         The heuristic value of the current game state to the specified player.
78     """
79
80     if game.is_loser(player):
81         return float("-inf")
82
83     if game.is_winner(player):
84         return float("inf")
85
86     # player locations
87     own_x, own_y = game.get_player_location(player)
88     opp_x, opp_y = game.get_player_location(game.get_opponent(player))
89
90     # relative coordinates player could reach within 2 moves
91     directions = [
92         (1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1), (-2, 1), (-1, 2),
93         (-2, 0), (2, 0), (1, 1), (1, -1), (0, -2), (0, 2), (-1, -1), (-1, 1),
94         (1, 3), (3, 3), (3, 1), (3, -1), (3, -3), (1, -3), (-1, -3), (-3, -3),
95         (-3, -1), (-3, 1), (-3, 3), (-1, 3), (-2, -4), (0, -4), (2, -4), (-2, 4),
96         (0, 4), (2, 4), (-4, -2), (-4, 0), (-4, 2), (4, -2), (4, 0), (4, 2)
97     ]
98
99     # squares reachable within 2 moves
100     own_space = [(own_x+xd, own_y+yd) for xd, yd in directions
101                  if game.move_is_legal((own_x+xd, own_y+yd))]
102     opp_space = [(opp_x+xd, opp_y+yd) for xd, yd in directions
103                  if game.move_is_legal((opp_x+xd, opp_y+yd))]
104
105     # return delta of reachable squares
106     return float(len(own_space) - len(opp_space))

```

AWESOME

Terrific! It's a chef's job.

```

107
108
109 def custom_score_3(game, player):

```

AWESOME

Superb! The implementation of this function is impressive.

```

110     """Calculates the heuristic value of a game state from the point of view
111     of the given player.
112
113     This function rewards moves toward the center of the board and penalizes
114     moves along the edges and corners.
115
116     Parameters
117     -----
118     game : `isolation.Board`
119         An instance of `isolation.Board` encoding the current state of the
120         game (e.g., player locations and blocked cells).
121
122     player : object
123         A player instance in the current game (i.e., an object corresponding to
124         one of the player objects `game.__player_1__` or `game.__player_2__`.)
125
126     Returns
127     -----
128     float

```

```

129     The heuristic value of the current game state to the specified player.
130     """
131     if game.is_loser(player):
132         return float("-inf")
133
134     if game.is_winner(player):
135         return float("inf")
136
137     # get player location
138     x, y = game.get_player_location(player)
139
140     score = 1 # default score for moves along outer edge of board
141
142     # get list of empty spaces
143     empty_spaces = game.get_blank_spaces()
144
145     # reward moves in center of board at the beginning of game
146     if (len(empty_spaces) > 40) and (x >= 2 and x <= 4) and (y >= 2 and y <= 4):
147         score = 10
148         return score
149
150     # reward moves in center of board
151     if (x >= 2 and x <= 4) and (y >= 2 and y <= 4):
152         score = 5
153         return score
154
155     # reward moves in 2nd ring of board
156     if (x == 1 or x == 5) and (y >= 1 and y <= 5):
157         score = 3
158         return score
159
160     # penalize moves in corners
161     corners = [(0, 0), (0, 6), (6, 0), (6, 6)]
162     if (x, y) in corners:
163         score = 0
164
165     return float(score)
166
167 class IsolationPlayer:
168     """Base class for minimax and alphabeta agents -- this class is never
169     constructed or tested directly.
170
171     ***** DO NOT MODIFY THIS CLASS *****
172
173     Parameters
174     -----
175     search_depth : int (optional)
176         A strictly positive integer (i.e., 1, 2, 3,...) for the number of
177         layers in the game tree to explore for fixed-depth search. (i.e., a
178         depth of one (1) would only explore the immediate successors of the
179         current state.)
180
181     score_fn : callable (optional)
182         A function to use for heuristic evaluation of game states.
183
184     timeout : float (optional)
185         Time remaining (in milliseconds) when search is aborted. Should be a
186         positive value large enough to allow the function to return before the
187         timer expires.
188     """
189     def __init__(self, search_depth=3, score_fn=custom_score, timeout=10.):
190         self.search_depth = search_depth
191         self.score = score_fn
192         self.time_left = None
193         self.TIMER_THRESHOLD = timeout
194
195
196 class MinimaxPlayer(IsolationPlayer):
197     """Game-playing agent that chooses a move using depth-limited minimax
198     search. You must finish and test this player to make sure it properly uses
199     minimax to return a good move before the search time limit expires.
200     """
201
202     def get_move(self, game, time_left):
203         """Search for the best move from the available legal moves and return a
204         result before the time limit expires.
205
206         ***** YOU DO NOT NEED TO MODIFY THIS FUNCTION *****
207
208         For fixed-depth search, this function simply wraps the call to the
209         minimax method, but this method provides a common interface for all
210         Isolation agents, and you will replace it in the AlphaBetaPlayer with
211         iterative deepening search.
212
213     Parameters
214     -----
215     game : `Isolation.Board`
216         An instance of `Isolation.Board` encoding the current state of the
217         game (e.g., player locations and blocked cells).
218
219     time_left : callable

```

```

220         A function that returns the number of milliseconds left in the
221         current turn. Returning with any less than 0 ms remaining forfeits
222         the game.
223
224     Returns
225     -----
226     (int, int)
227         Board coordinates corresponding to a legal move; may return
228         (-1, -1) if there are no available legal moves.
229     """
230     self.time_left = time_left
231
232     # Initialize the best move so that this function returns something
233     # in case the search fails due to timeout
234     best_move = (-1, -1)
235
236     try:
237         # The try/except block will automatically catch the exception
238         # raised when the timer is about to expire.
239         best_move = self.minimax(game, self.search_depth)
240
241     except SearchTimeout:
242         # Handle any actions required after timeout as needed
243         return best_move
244
245     # Return the best move from the last completed search iteration
246     return best_move
247
248 def minimax(self, game, depth):
249     """Implement depth-limited minimax search algorithm as described in
250     the lectures.
251
252     This should be a modified version of MINIMAX-DECISION in the AIMA text.
253     https://github.com/aimacode/aima-pseudocode/blob/master/md/Minimax-Decision.md
254
255     *****
256     You MAY add additional methods to this class, or define helper
257     functions to implement the required functionality.
258     *****
259
260     Parameters
261     -----
262     game : isolation.Board
263         An instance of the Isolation game `Board` class representing the
264         current game state
265
266     depth : int
267         Depth is an integer representing the maximum number of plies to
268         search in the game tree before aborting
269
270     Returns
271     -----
272     (int, int)
273         The board coordinates of the best move found in the current search;
274         (-1, -1) if there are no legal moves
275
276     Notes
277     -----
278     (1) You MUST use the `self.score()` method for board evaluation
279         to pass the project tests; you cannot call any other evaluation
280         function directly.
281
282     (2) If you use any helper functions (e.g., as shown in the AIMA
283         pseudocode) then you must copy the timer check into the top of
284         each helper function or else your agent will timeout during
285         testing.
286     """
287     if self.time_left() < self.TIMER_THRESHOLD:
288         raise SearchTimeout()
289
290     # Get legal moves, if any
291     legal_moves = game.get_legal_moves(self)
292     if not legal_moves:
293         return (-1, -1)
294
295     # Initialize the best move, best value
296     best_move = legal_moves[0]
297     best_value = float('-inf')
298
299     # Recurse through legal moves
300     for move in legal_moves:
301         # calculate value of opponent's minimizing method
302         value = self.min_value(game.forecast_move(move), depth - 1)
303         # take max value from opponent's available moves
304         if value > best_value:
305             best_value = value
306             best_move = move
307
308     return best_move
309
310 def min_value(self, game, depth):
311     """ Implements the MIN-VALUE method as described in the AIMA

```

```

313     MINIMAX-DECISION text.
314     """
315     if self.time_left() < self.TIMER_THRESHOLD:
316         raise SearchTimeout()
317
318     # Get legal moves for opponent, if none then return value of current game state
319     legal_moves = game.get_legal_moves(game.get_opponent(self))
320     if depth == 0 or not legal_moves:
321         return self.score(game, self)
322
323     # Otherwise, initialize the best move, lowest value
324     best_move = (-1, -1)
325     min_value = float('inf')
326
327     # Recurse opponent's moves
328     for move in legal_moves:
329         # calculate value from my maximizing method
330         value = self.max_value(game.forecast_move(move), depth - 1)
331         # take lowest value from my available moves
332         if value < min_value:
333             min_value = value
334             best_move = move
335     # Return lowest value
336     return min_value
337
338     def max_value(self, game, depth):
339         """ Implements the MAX-VALUE method as described in the AIMA
340         MINIMAX-DECISION text.
341         """
342         if self.time_left() < self.TIMER_THRESHOLD:
343             raise SearchTimeout()
344
345         # Get legal moves for opponent, if none then return value of current game state
346         legal_moves = game.get_legal_moves(self)
347         if depth == 0 or not legal_moves:
348             return self.score(game, self)
349
350         # Otherwise, initialize the best move, highest value
351         best_move = (-1, -1)
352         max_value = float('-inf')
353
354         # Recurse my moves
355         for move in legal_moves:
356             # calculate value from my opponent's minimizing method
357             value = self.min_value(game.forecast_move(move), depth - 1)
358             # take max value from possible opponent moves
359             if value > max_value:
360                 max_value = value
361                 best_move = move
362         # Return highest value
363         return max_value
364
365
366     class AlphaBetaPlayer(IsolationPlayer):
367         """Game-playing agent that chooses a move using iterative deepening minimax
368         search with alpha-beta pruning. You must finish and test this player to
369         make sure it returns a good move before the search time limit expires.
370         """
371
372         def get_move(self, game, time_left):
373             """Search for the best move from the available legal moves and return a
374             result before the time limit expires.
375
376             Modify the get_move() method from the MinimaxPlayer class to implement
377             iterative deepening search instead of fixed-depth search.
378
379             *****
380             NOTE: If time_left() < 0 when this function returns, the agent will
381             forfeit the game due to timeout. You must return _before_ the
382             timer reaches 0.
383             *****
384
385             Parameters
386             -----
387             game : `isolation.Board`
388                 An instance of `isolation.Board` encoding the current state of the
389                 game (e.g., player locations and blocked cells).
390
391             time_left : callable
392                 A function that returns the number of milliseconds left in the
393                 current turn. Returning with any less than 0 ms remaining forfeits
394                 the game.
395
396             Returns
397             -----
398             (int, int)
399                 Board coordinates corresponding to a legal move; may return
400                 (-1, -1) if there are no available legal moves.
401             """
402             self.time_left = time_left
403
404             # Initialize the best move so that this function returns something

```

```

405     # in case the search fails due to timeout
406     best_move = (-1, -1)
407     depth = 0
408
409     try:
410         # The try/except block will automatically catch the exception
411         # raised when the timer is about to expire.
412         while True:
413             depth += 1
414             best_move = self.alphabeta(game, depth)
415
416     except SearchTimeout:
417         # Handle any actions required after timeout as needed
418         return best_move
419
420     # Return the best move from the last completed search iteration
421     return best_move
422
423 def alphabeta(self, game, depth, alpha=float("-inf"), beta=float("inf")):
424     """Implement depth-limited minimax search with alpha-beta pruning as
425     described in the lectures.
426
427     This should be a modified version of ALPHA-BETA-SEARCH in the AIMA text
428     https://github.com/aimacode/aima-pseudocode/blob/master/md/Alpha-Beta-Search.md
429
430     *****
431     You MAY add additional methods to this class, or define helper
432     functions to implement the required functionality.
433     *****
434
435     Parameters
436     -----
437     game : isolation.Board
438         An instance of the Isolation game `Board` class representing the
439         current game state
440
441     depth : int
442         Depth is an integer representing the maximum number of plies to
443         search in the game tree before aborting
444
445     alpha : float
446         Alpha limits the lower bound of search on minimizing layers
447
448     beta : float
449         Beta limits the upper bound of search on maximizing layers
450
451     Returns
452     -----
453     (int, int)
454         The board coordinates of the best move found in the current search;
455         (-1, -1) if there are no legal moves
456
457     Notes
458     -----
459     (1) You MUST use the `self.score()` method for board evaluation
460         to pass the project tests; you cannot call any other evaluation
461         function directly.
462
463     (2) If you use any helper functions (e.g., as shown in the AIMA
464         pseudocode) then you must copy the timer check into the top of
465         each helper function or else your agent will timeout during
466         testing.
467     """
468     if self.time_left() < self.TIMER_THRESHOLD:
469         raise SearchTimeout()
470
471     # Get legal moves, if any
472     legal_moves = game.get_legal_moves(self)
473     if not legal_moves:
474         return (-1, -1)
475
476     # Initialize the best move, best value
477     best_move = legal_moves[0]
478     best_value = float('-inf')
479
480     # Recurse through legal moves
481     for move in legal_moves:
482         # calculate value from my opponent's minimizing AB method
483         value = self.min_value_ab(game.forecast_move(move), depth - 1, alpha, beta)
484         # take max value from possible opponent moves
485         if value > best_value:
486             best_value = value
487             alpha = value
488             best_move = move
489
490     return best_move
491
492 def min_value_ab(self, game, depth, alpha=float('-inf'), beta=float('inf')):
493     """ Implements the MIN-VALUE method as described in the AIMA
494     ALPHA-BETA-SEARCH text.
495     """
496     if self.time_left() < self.TIMER_THRESHOLD:

```

```
497         raise SearchTimeout()
498
499     # Get legal moves for the opponent, if any
500     legal_moves = game.get_legal_moves(game.get_opponent(self))
501     if depth == 0 or not legal_moves:
502         return self.score(game, self)
503
504     # Initialize best value for opponent
505     min_value = beta
506
507     # Recurse legal moves
508     for move in legal_moves:
509         # calculate value for each of my opponent's moves
510         value = self.max_value_ab(game.forecast_move(move), depth - 1, alpha, beta)
511         # return value if <= alpha
512         if value <= alpha:
513             return value
514         # update min_value
515         if value < min_value:
516             min_value = value
517         # update beta
518         if value < beta:
519             beta = value
520
521     return min_value
522
523 def max_value_ab(self, game, depth, alpha=float('-inf'), beta=float('inf')):
524     """ Implements the MAX-VALUE method as described in the AIMA
525     ALPHA-BETA-SEARCH text.
526     """
527     if self.time_left() < self.TIMER_THRESHOLD:
528         raise SearchTimeout()
529
530     # Get my legal moves, if any
531     legal_moves = game.get_legal_moves(self)
532     if depth == 0 or not legal_moves:
533         return self.score(game, self)
534
535     # Initialize best value for me
536     max_value = alpha
537
538     # Recurse my legal moves
539     for move in legal_moves:
540         # calculate value from my opponent's minimizing method
541         value = self.min_value_ab(game.forecast_move(move), depth - 1, alpha, beta)
542         # return value if >= beta
543         if value >= beta:
544             return value
545         # update max_value
546         if value > max_value:
547             max_value = value
548         # update alpha
549         if value > alpha:
550             alpha = value
551
552     return max_value
553
```

[RETURN TO PATH](#)

Rate this review

[Student FAQ](#)[Reviewer Agreement](#)