UDACITY

PROJECT

Build a Game-Playing Agent

A part of the Artificial Intelligence Nanodegree Program

PROJECT REVIEW

CODE REVIEW 3

NOTES

game_agent.py 3

```
"""This file contains all the classes you must complete for this project.

You can use the test cases in agent_test.py to help during development, and
augment the test suite with your own test cases to further test your code.

You must test your agent's strength against a set of agents with known
relative strength using tournament.py and include the results in your report.
"""
import random
import logging
import typing; from typing import *
import itertools
from itertools import product
from sample_players import null_score, open_move_score, improved_score

class Timeout(Exception):
    """Subclass base exception for code clarity."""
    pass

def get_move_difference_factor(game, player) -> float:
    count_own_moves = len(game.get_legal_moves(player))
    count_opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
    return (count_own_moves - count_opp_moves)

def get_center_available_factor(game, player) -> float:
    own_moves = game.get_legal_moves(player)
    center_x, center_y = game.width / 2, game.height / 2
    center_available = -1
    # Center of grid is only available when odd width and odd height
    if not center_x.is_integer() and not center_y.is_integer():
        center_coords = (int(center_x), int(center_y))
        center_available = own_moves.index(center_coords) if center_coords in own_moves else -1
    # Next move should always be to center square if available
    return 2.0 if (center_available != -1) else 1.0

def is_empty_board(count_total_positions, count_empty_coords):
    all_empty = True if (count_total_positions == count_empty_coords) else False
    if all_empty:
        return 1.0

def get_reflection_available_factor(game, player) -> float:
    count_total_positions = game.height * game.width
    count_empty_coords = len(game.get_blank_spaces())

    # Return if no reflection move possible before first move
    if is_empty_board(count_total_positions, count_empty_coords):
        return 1.0

    own_moves = game.get_legal_moves(player)
    opp_moves = game.get_legal_moves(game.get_opponent(player))
    count_own_moves = len(game.get_legal_moves(player))
    count_opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
    all_coords = list(itertools.product(range((game.width)), range((game.height))))
    player_coords = (player_x, player_y) = game.get_player_location(player)
    opp_coords = (opp_x, opp_y) = game.get_player_location(game.get_opponent(player))
```

```
    player_index = all_coords.index(player_coords)
    opp_index = all_coords.index(opp_coords)
    mirrored_all_coords = all_coords[::-1]
    mirrored_player_coords = mirrored_all_coords[player_index]
    mirrored_opp_coords = mirrored_all_coords[opp_index]

    # Return high Reflection Available Factor if the mirror coords that
    # correspond to the oppositions current coords is an available legal move for current player
    for legal_player_move_coords in own_moves:
        if legal_player_move_coords == mirrored_opp_coords:
            return 2.0
    return 1.0

def get_partition_possible_factor(game, player):
    count_total_positions = game.height * game.width
    count_empty_coords = len(game.get_blank_spaces())

    empty_coords = game.get_blank_spaces()

    # Return if no partition possible before first move
    if is_empty_board(count_total_positions, count_empty_coords):
        return 1.0

    own_moves = game.get_legal_moves(player)
    opp_moves = game.get_legal_moves(game.get_opponent(player))

    for move in own_moves:
        cell_left = (move[0]-1, move[1])
        cell_right = (move[0]+1, move[1])
        cell_below = (move[0], move[1]-1)
        cell_above = (move[0], move[1]+1)

        cell_left_x2 = (move[0]-2, move[1])
        cell_right_x2 = (move[0]+2, move[1])
        cell_below_x2 = (move[0], move[1]-2)
        cell_above_x2 = (move[0], move[1]+2)

        is_cell_left = cell_left not in empty_coords
        is_cell_right = cell_right not in empty_coords
        is_cell_below = cell_below not in empty_coords
        is_cell_above = cell_above not in empty_coords

        is_cell_left_x2 = cell_left_x2 not in empty_coords
        is_cell_right_x2 = cell_right_x2 not in empty_coords
        is_cell_below_x2 = cell_below_x2 not in empty_coords
        is_cell_above_x2 = cell_above_x2 not in empty_coords

        # Firstly check if two cells in sequence on either side of possible move
        # If so give double bonus points
        if ( (is_cell_left and is_cell_left_x2) or
             (is_cell_right and is_cell_right_x2) or
             (is_cell_below and is_cell_below_x2) or
             (is_cell_above and is_cell_above_x2) ):
            return 4.0

        # Secondly check if just one cell surrounding possible move
        if (is_cell_left or
            is_cell_right or
            is_cell_below or
            is_cell_above):
            return 2.0

    return 1.0

def heuristic_1_center(game, player) -> float:
    """
    Evaluation function outputs a
    score equal to the Center Available Factor
    that has higher weight when center square still available on any move

    Parameters
    ----------
    game : `isolation.Board`
        An instance of `isolation.Board` encoding the current state of the
        game (e.g., player locations and blocked cells).

    player : hashable
        One of the objects registered by the game object as a valid player.
        (i.e., `player` should be either game.__player_1__ or
        game.__player_2__).

    Returns
    ----------
    float
        The heuristic value of the current game state
    """
    center_available_factor = get_center_available_factor(game, player)
```

```
144     # Heuristic score output
145     return float(center_available_factor)
146 
147 def heuristic_2_reflection(game, player) -> float:
148     """
149     Heuristic 2's Reflection Available Factor
150     has higher weight when reflection of opposition player
151     position is available on other side of board.
152     i.e. In game tree, for all available opposition in coordinates,
153     count how many available reflection moves (on opposite side of board)
154     are available as a legal moves for the current player. These should result in
155     higher weight if available
156 
157     Parameters
158     ----------
159     game : `isolation.Board`
160         An instance of `isolation.Board` encoding the current state of the
161         game (e.g., player locations and blocked cells).
162 
163     player : hashable
164         One of the objects registered by the game object as a valid player.
165         (i.e., `player` should be either game.__player_1__ or
166         game.__player_2__).
167 
168     Returns
169     ----------
170     float
171         The heuristic value of the current game state
172     """
173 
174     reflection_available_factor = get_reflection_available_factor(game, player)
175 
176     return float(reflection_available_factor)
177 
178 def heuristic_3_partition(game, player) -> float:
179     """
180     Heuristic 3's Partition Growth Factor
181     has higher weight when available moves are
182     vertically or horizontally (not diagonally) adjacent
183     to a sequence of one or two blocked locations
184 
185     Parameters
186     ----------
187     game : `isolation.Board`
188         An instance of `isolation.Board` encoding the current state of the
189         game (e.g., player locations and blocked cells).
190 
191     player : hashable
192         One of the objects registered by the game object as a valid player.
193         (i.e., `player` should be either game.__player_1__ or
194         game.__player_2__).
195 
196     Returns
197     ----------
198     float
199         The heuristic value of the current game state
200     """
201 
202     partition_possible_factor = get_partition_possible_factor(game, player)
203 
204     return float(partition_possible_factor)
205 
206 def heuristic_combined_1_2(game, player) -> float:
207     """
208     Combines Heuristics 1 and 2
209 
210     Parameters
211     ----------
212     game : `isolation.Board`
213         An instance of `isolation.Board` encoding the current state of the
214         game (e.g., player locations and blocked cells).
215 
216     player : hashable
217         One of the objects registered by the game object as a valid player.
218         (i.e., `player` should be either game.__player_1__ or
219         game.__player_2__).
220 
221     Returns
222     ----------
223     float
224         The heuristic value of the current game state
225     """
226 
227     center_available_factor = get_center_available_factor(game, player)
228     reflection_available_factor = get_reflection_available_factor(game, player)
229 
230     return float(center_available_factor + reflection_available_factor)
```

```
232 def heuristic_combined_1_3(game, player) -> float:
233     """
234     Combines Heuristics 1 and 3
235 
236     Parameters
237     ----------
238     game : `isolation.Board`
239         An instance of `isolation.Board` encoding the current state of the
240         game (e.g., player locations and blocked cells).
241 
242     player : hashable
243         One of the objects registered by the game object as a valid player.
244         (i.e., `player` should be either game.__player_1__ or
245         game.__player_2__).
246 
247     Returns
248     ----------
249     float
250         The heuristic value of the current game state
251     """
252 
253     center_available_factor = get_center_available_factor(game, player)
254     partition_possible_factor = get_partition_possible_factor(game, player)
255 
256     return float(center_available_factor + partition_possible_factor)
257 
258 def heuristic_combined_2_3(game, player) -> float:
259     """
260     Combines Heuristics 2 and 3
261 
262     Parameters
263     ----------
264     game : `isolation.Board`
265         An instance of `isolation.Board` encoding the current state of the
266         game (e.g., player locations and blocked cells).
267 
268     player : hashable
269         One of the objects registered by the game object as a valid player.
270         (i.e., `player` should be either game.__player_1__ or
271         game.__player_2__).
272 
273     Returns
274     ----------
275     float
276         The heuristic value of the current game state
277     """
278 
279     reflection_available_factor = get_reflection_available_factor(game, player)
280     partition_possible_factor = get_partition_possible_factor(game, player)
281 
282     return float(reflection_available_factor + partition_possible_factor)
283 
284 def heuristic_combined_1_2_3(game, player) -> float:
285     """
286     Combines Heuristics 1, 2 and 3
287 
288     Parameters
289     ----------
290     game : `isolation.Board`
291         An instance of `isolation.Board` encoding the current state of the
292         game (e.g., player locations and blocked cells).
293 
294     player : hashable
295         One of the objects registered by the game object as a valid player.
296         (i.e., `player` should be either game.__player_1__ or
297         game.__player_2__).
298 
299     Returns
300     ----------
301     float
302         The heuristic value of the current game state
303     """
304 
305     center_available_factor = get_center_available_factor(game, player)
306     reflection_available_factor = get_reflection_available_factor(game, player)
307     partition_possible_factor = get_partition_possible_factor(game, player)
308 
309     return float(center_available_factor +
310                  reflection_available_factor +
311                  partition_possible_factor)
312 
313 def custom_score(game, player):
314     """Calculate the heuristic value of a game state from the point of view
315     of the given player.
316 
317     Note: this function should be called from within a Player instance as
318     `self.score()` -- you should not need to call this function directly.
```

```
    Parameters
    ----------
    game : `isolation.Board`
        An instance of `isolation.Board` encoding the current state of the
        game (e.g., player locations and blocked cells).

    player : object
        A player instance in the current game (i.e., an object corresponding to
        one of the player objects `game.__player_1__` or `game.__player_2__`.)

    Returns
    -------
    float
        The heuristic value of the current game state to the specified player.
    """

    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    heuristics_options = {
        "null_score": null_score,
        "open_move_score": open_move_score,
        "improved_score": improved_score,
        "heuristic_1_center": heuristic_1_center,
        "heuristic_2_reflection": heuristic_2_reflection,
        "heuristic_3_partition": heuristic_3_partition,
        "heuristic_combined_1_2": heuristic_combined_1_2,
        "heuristic_combined_1_3": heuristic_combined_1_3,
        "heuristic_combined_2_3": heuristic_combined_2_3,
        "heuristic_combined_1_2_3": heuristic_combined_1_2_3
    }

    return heuristics_options["heuristic_2_reflection"](game, player)
```

AWESOME

Excellent job! Your evaluation functions are very well written and documented ⭐

```
class CustomPlayer:
    """Game-playing agent that chooses a move using your evaluation function
    and a depth-limited minimax algorithm with alpha-beta pruning. You must
    finish and test this player to make sure it properly uses minimax and
    alpha-beta to return a good move before the search time limit expires.

    Parameters
    ----------
    search_depth : int (optional)
        A strictly positive integer (i.e., 1, 2, 3,...) for the number of
        layers in the game tree to explore for fixed-depth search. (i.e., a
        depth of one (1) would only explore the immediate successors of the
        current state.)

    score_fn : callable (optional)
        A function to use for heuristic evaluation of game states.

    iterative : boolean (optional)
        Flag indicating whether to perform fixed-depth search (False) or
        iterative deepening search (True).

    method : {'minimax', 'alphabeta'} (optional)
        The name of the search method to use in get_move().

    timeout : float (optional)
        Time remaining (in milliseconds) when search is aborted. Should be a
        positive value large enough to allow the function to return before the
        timer expires.
    """

    def __init__(self, search_depth=3, score_fn=custom_score,
                 iterative=True, method='minimax', timeout=10.):
        self.search_depth = search_depth
        self.iterative = iterative
        self.score = score_fn
        self.method = method
        self.time_left = None
        self.TIMER_THRESHOLD = timeout

    def get_move(self, game, legal_moves, time_left):
        """Search for the best move from the available legal moves and return a
        result before the time limit expires.
```

```
        This function must perform iterative deepening if self.iterative=True,
        and it must use the search method (minimax or alphabeta) corresponding
        to the self.method value.

        ********************************************************************
        NOTE: If time_left < 0 when this function returns, the agent will
              forfeit the game due to timeout. You must return _before_ the
              timer reaches 0.
        ********************************************************************

        Parameters
        ----------
        game : `isolation.Board`
            An instance of `isolation.Board` encoding the current state of the
            game (e.g., player locations and blocked cells).

        legal_moves : list<(int, int)>
            A list containing legal moves. Moves are encoded as tuples of pairs
            of ints defining the next (row, col) for the agent to occupy.

        time_left : callable
            A function that returns the number of milliseconds left in the
            current turn. Returning with any less than 0 ms remaining forfeits
            the game.

        Returns
        -------
        (int, int)
            Board coordinates corresponding to a legal move; may return
            (-1, -1) if there are no available legal moves.
        """

        self.time_left = time_left

        # Perform any required initializations, including selecting an initial
        # move from the game board (i.e., an opening book), or returning
        # immediately if there are no legal moves

        remaining_legal_moves = legal_moves
        no_legal_moves = (-1, -1)
        best_move = no_legal_moves
        if not remaining_legal_moves:
            logging.debug("Get Moves - Terminated due to no remaining legal moves")
            return no_legal_moves

        # Flag indicating Iterative Deepening Search - Initialise Depth at 0 (to later be incremented)
        #   - Reference: https://github.com/aimacode/aima-pseudocode/blob/master/md/Iterative-Deepening-Search
        # Flag otherwise indicates Fixed-Depth Search (FDS) - Set to Search Depth parameter (only for FDS)
        depth = 0 if self.iterative else self.search_depth

        try:
            # The search method call (alpha beta or minimax) should happen in
            # here in order to avoid timeout. The try/except block will
            # automatically catch the exception raised by the search method
            # when the timer gets close to expiring

            # Flag indicates perform Iterative Deepening Search
            if self.iterative:
                logging.debug("Get Moves - Performing Iterative Deepening Search to depth %r: ", depth)
                while True:
                    # logging.debug("Time left is: %r", self.time_left())
                    depth += 1
                    if self.method == 'minimax':
                        _, best_move = self.minimax(game, depth)
                    elif self.method == 'alphabeta':
                        _, best_move = self.alphabeta(game, depth)
                    else:
                        raise ValueError("Invalid method")

                    # Check remaining time between depth iterations and
                    # return the best move when less than 1ms to avoid
                    # running out of time and forfeiting the game
                    if self.time_left() <= 0.001:
                        return best_move

            # Flag indicates perform Fixed-Depth Search
            else:
                logging.debug("Get Moves - Performing Fixed-Depth Search to depth %r: ", depth)
                # logging.debug("Time left is: %r", self.time_left())
                if self.method == 'minimax':
                    _, best_move = self.minimax(game, depth)
                elif self.method == 'alphabeta':
                    _, best_move = self.alphabeta(game, depth)
                else:
                    raise ValueError("Invalid method")
                return best_move

        except Timeout:
```

```
            # Handle any actions required at timeout, if necessary
            # logging.warning("Get Moves - Timeout reached")

            return best_move

        # Return the best move from the last completed search iteration
        return best_move
```

AWESOME

Excellent work coding the `get_move` method! Using the `logging` module throughout your code takes it to the next level!! ⭐

```
    def minimax(self, game, depth, maximizing_player=True):
        """Implement the minimax search algorithm as described in the lectures.

        Parameters
        ----------
        game : isolation.Board
            An instance of the Isolation game `Board` class representing the
            current game state

        depth : int
            Depth is an integer representing the maximum number of plies to
            search in the game tree before aborting

        maximizing_player : bool
            Flag indicating whether the current search depth corresponds to a
            maximizing layer (True) or a minimizing layer (False)

        Returns
        -------
        float
            The score for the current search branch

        tuple(int, int)
            The best move for the current branch; (-1, -1) for no legal moves

        Notes
        -----
            (1) You MUST use the `self.score()` method for board evaluation
                to pass the project unit tests; you cannot call any other
                evaluation function directly.
        """
        if self.time_left() < self.TIMER_THRESHOLD:
            raise Timeout()

        # Reference: https://github.com/aimacode/aima-pseudocode/blob/master/md/Minimax-Decision.md

        # Initialise variable for no legal moves
        no_legal_moves = (-1, -1)
        best_move = no_legal_moves
        best_utility = float('-inf') if maximizing_player else float('inf')
        current_player = game.active_player if maximizing_player else game.inactive_player
        remaining_legal_moves = game.get_legal_moves(game.active_player)

        logging.debug("Current player is Maximizing: %r", maximizing_player)
        logging.debug("Current depth: %r", depth)
        logging.debug("Best utility: %r", best_utility)
        logging.debug("Remaining legal moves: %r", remaining_legal_moves)

        # Recursion function termination conditions when legal moves exhausted or no plies left
        if not remaining_legal_moves:
            logging.debug("Recursion terminated due to no remaining legal moves")
            return game.utility(current_player), no_legal_moves
        elif depth == 0:
            logging.debug("Recursion terminated due to no more plies to search")
            return self.score(game, current_player), remaining_legal_moves[0]

        # Recursively alternate between Maximise and Minimise calculations for decrementing depths
        for move in remaining_legal_moves:
            # logging.debug("Recursion with time left is: %r", self.time_left())
            logging.debug("Recursion with move: %r", move)
            logging.debug("Best utility: %r", best_utility)
            logging.debug("Best move: %r", best_move)

            # Obtain successor of current state by creating copy of board and applying a move.
            next_state = game.forecast_move(move)
            forecast_utility, _ = self.minimax(next_state, depth - 1, not maximizing_player)
            logging.debug("Forecast utility: %r", forecast_utility)

            if maximizing_player:
                logging.debug("Checking move with Maximising player, forecast_utility > best_utility? : %r", (
                if forecast_utility > best_utility:
                    best_utility, best_move = forecast_utility, move
```

```
568             else:
569                 logging.debug("Checking move with Minimising player, forecast_utility < best_utility? : %r", (
570                 if forecast_utility < best_utility:
571                     best_utility, best_move = forecast_utility, move
572 
573         return best_utility, best_move
574 
575     def alphabeta(self, game, depth, alpha=float("-inf"), beta=float("inf"), maximizing_player=True):
576         """Implement minimax search with alpha-beta pruning as described in the
577         lectures.
578 
579         Parameters
580         ----------
581         game : isolation.Board
582             An instance of the Isolation game `Board` class representing the
583             current game state
584 
585         depth : int
586             Depth is an integer representing the maximum number of plies to
587             search in the game tree before aborting
588 
589         alpha : float
590             Alpha limits the lower bound of search on minimizing layers
591 
592         beta : float
593             Beta limits the upper bound of search on maximizing layers
594 
595         maximizing_player : bool
596             Flag indicating whether the current search depth corresponds to a
597             maximizing layer (True) or a minimizing layer (False)
598 
599         Returns
600         -------
601         float
602             The score for the current search branch
603 
604         tuple(int, int)
605             The best move for the current branch; (-1, -1) for no legal moves
606 
607         Notes
608         -----
609             (1) You MUST use the `self.score()` method for board evaluation
610                 to pass the project unit tests; you cannot call any other
611                 evaluation function directly.
612         """
613         if self.time_left() < self.TIMER_THRESHOLD:
614             raise Timeout()
615 
616         # TODO - Refactor duplicate from minimax and alphabeta into helper function
617         # Reference: https://github.com/aimacode/aima-pseudocode/blob/master/md/Alpha-Beta-Search.md
618 
619         # Initialise variable for no legal moves
620         no_legal_moves = (-1, -1)
621         best_move = no_legal_moves
622         best_utility = float('-inf') if maximizing_player else float('inf')
623         current_player = game.active_player if maximizing_player else game.inactive_player
624         remaining_legal_moves = game.get_legal_moves(game.active_player)
625 
626         logging.debug("Current player is Maximizing: %r", maximizing_player)
627         logging.debug("Current depth: %r", depth)
628         logging.debug("Best utility: %r", best_utility)
629         logging.debug("Remaining legal moves: %r", remaining_legal_moves)
630 
631         # Recursion function termination conditions when legal moves exhausted or no plies left
632         if not remaining_legal_moves:
633             logging.debug("Recursion terminated due to no remaining legal moves")
634             return game.utility(current_player), no_legal_moves
635         elif depth == 0:
636             logging.debug("Recursion terminated due to no more plies to search")
637             return self.score(game, current_player), remaining_legal_moves[0]
638 
639         # Recursively alternate between Maximise and Minimise calculations for decrementing depths
640         for move in remaining_legal_moves:
641             # logging.debug("Recursion with time left is: %r", self.time_left())
642             logging.debug("Recursion with move: %r", move)
643             logging.debug("Best utility: %r", best_utility)
644             logging.debug("Best move: %r", best_move)
645 
646             # Obtain successor of current state by creating copy of board and applying a move.
647             next_state = game.forecast_move(move)
648             forecast_utility, _ = self.alphabeta(next_state, depth - 1, alpha, beta, not maximizing_player)
649             logging.debug("Forecast utility: %r", forecast_utility)
650 
651             if maximizing_player:
652                 logging.debug("Checking move with Maximising player, forecast_utility > best_utility? : %r", (
653                 if forecast_utility > best_utility:
654                     best_utility, best_move = forecast_utility, move
```

```
655                     # Prune next successor node if possible
656                     if best_utility >= beta:
657                         break
658                     alpha = max(alpha, best_utility)
659             else:
660                 logging.debug("Checking move with Minimising player, forecast_utility < best_utility? : %r", (
661                 if forecast_utility < best_utility:
662                     best_utility, best_move = forecast_utility, move
663 
664                     # Prune next successor node if possible
665                     if best_utility <= alpha:
666                         break
667                     beta = min(beta, best_utility)
668 
669         return best_utility, best_move
670 
```

AWESOME

Again, awesome job here! Both the `minimax` and `alphabeta` methods are implemented in a very clear with perfect descriptions. And the logging features are absolutely spectacular!

```
671 
672 def run():
673     try:
674         # Copy of minimax Unit Test for debugging only
675         import isolation
676         h, w = 7, 7
677         test_depth = 1
678         starting_location = (5, 3)
679         adversary_location = (0, 0)
680         iterative_search = False
681         search_method = "minimax"
682         heuristic = lambda g, p: 0.
683         agentUT = CustomPlayer(
684             test_depth, heuristic, iterative_search, search_method)
685         agentUT.time_left = lambda: 99
686         board = isolation.Board(agentUT, 'null_agent', w, h)
687         board.apply_move(starting_location)
688         board.apply_move(adversary_location)
689         legal_moves = board.get_legal_moves()
690 
691         # for move in legal_moves:
692         #     next_state = board.forecast_move(move)
693         #     v, _ = agentUT.minimax(next_state, test_depth)
694         #     assert type(v) is float, "Minimax function should return a floating point value approximating the
695 
696         move = agentUT.get_move(board, legal_moves, lambda: 99)
697         assert move in legal_moves, "The get_move() function failed as player 1 on a game in progress. It shou
698 
699         return
700     except SystemExit:
701         logging.exception('SystemExit occurred')
702     except:
703         logging.exception('Unknown exception occurred.')
704 
705 if __name__ == '__main__':
706     run()
```

Have a question about your review? Email us at review-support@udacity.com and include the link to this review.

RETURN TO PATH

Student FAQ