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

```python
"""Finish all TODO items in this file to complete the isolation project, then
test your agent's strength against a set of known agents using tournament.py
and include the results in your report.
"""
import random


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


def custom_score(game, player):
    """Calculates the heuristic value of a game state from the point of view
    of the given player.

    This function is an improvement on the AB_Improved heuristic. In the
    beginning of the game, it tries aggresively to reduce the number of opponent
    moves. But, as the game goes on, it becomes increasingly aggressive at
    maximizing its own moves.

    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")

    # get current move count
    move_count = game.move_count

    # count number of moves available
    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))

    # calculate weight
    w = 10 / (move_count + 1)

    # return weighted delta of available moves
    return float(own_moves - (w * opp_moves))
```

AWESOME

Good! Remarkable implementation.

```python


def custom_score_2(game, player):
    """Calculates the heuristic value of a game state from the point of view
    of the given player.

    This function seeks to increase the number of spaces the active player can
    reach within two moves, while decreasing this number for the opponent.

    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")

    # player locations
    own_x, own_y = game.get_player_location(player)
    opp_x, opp_y = game.get_player_location(game.get_opponent(player))

    # relative coordinates player could reach within 2 moves
    directions = [
        (1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1), (-2, 1), (-1, 2),
        (-2, 0), (2, 0), (1, 1), (1, -1), (0, -2), (0, 2), (-1, -1), (-1, 1),
        (1, 3), (3, 3), (3, 1), (3, -1), (3, -3), (1, -3), (-1, -3), (-3, -3),
        (-3, -1), (-3, 1), (-3, 3), (-1, 3), (-2, -4), (0, -4), (2, -4), (-2, 4),
        (0, 4), (2, 4), (-4, -2), (-4, 0), (-4, 2), (4, -2), (4, 0), (4, 2)
        ]

    # squares reachable within 2 moves
    own_space = [(own_x+xd, own_y+yd) for xd, yd in directions
                          if game.move_is_legal((own_x+xd, own_y+yd))]
    opp_space = [(opp_x+xd, opp_y+yd) for xd, yd in directions
                          if game.move_is_legal((opp_x+xd, opp_y+yd))]

    # return delta of reachable squares
    return float(len(own_space) - len(opp_space))
```

AWESOME

Terrific! It's a chef's job.

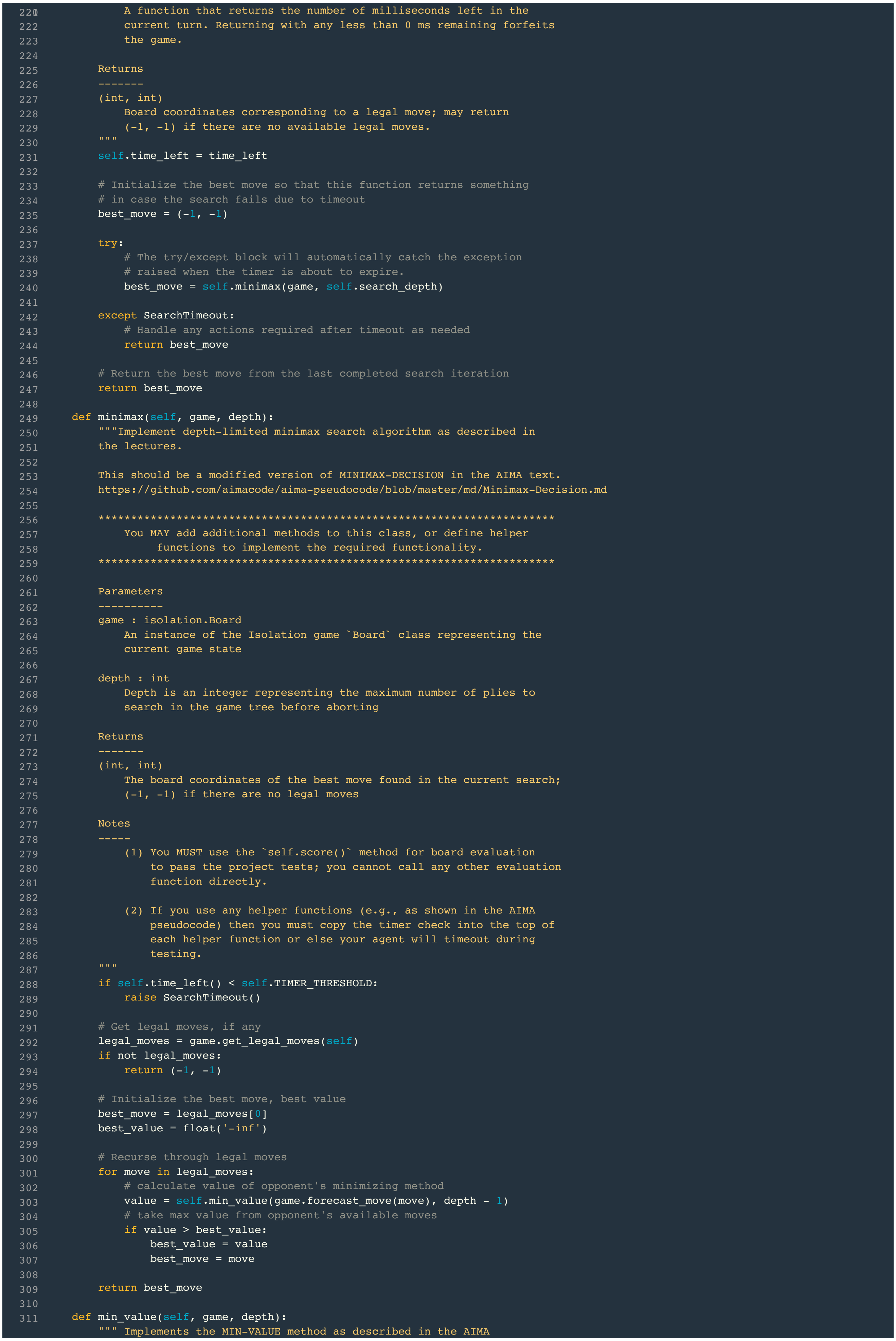
```python


def custom_score_3(game, player):
```

AWESOME

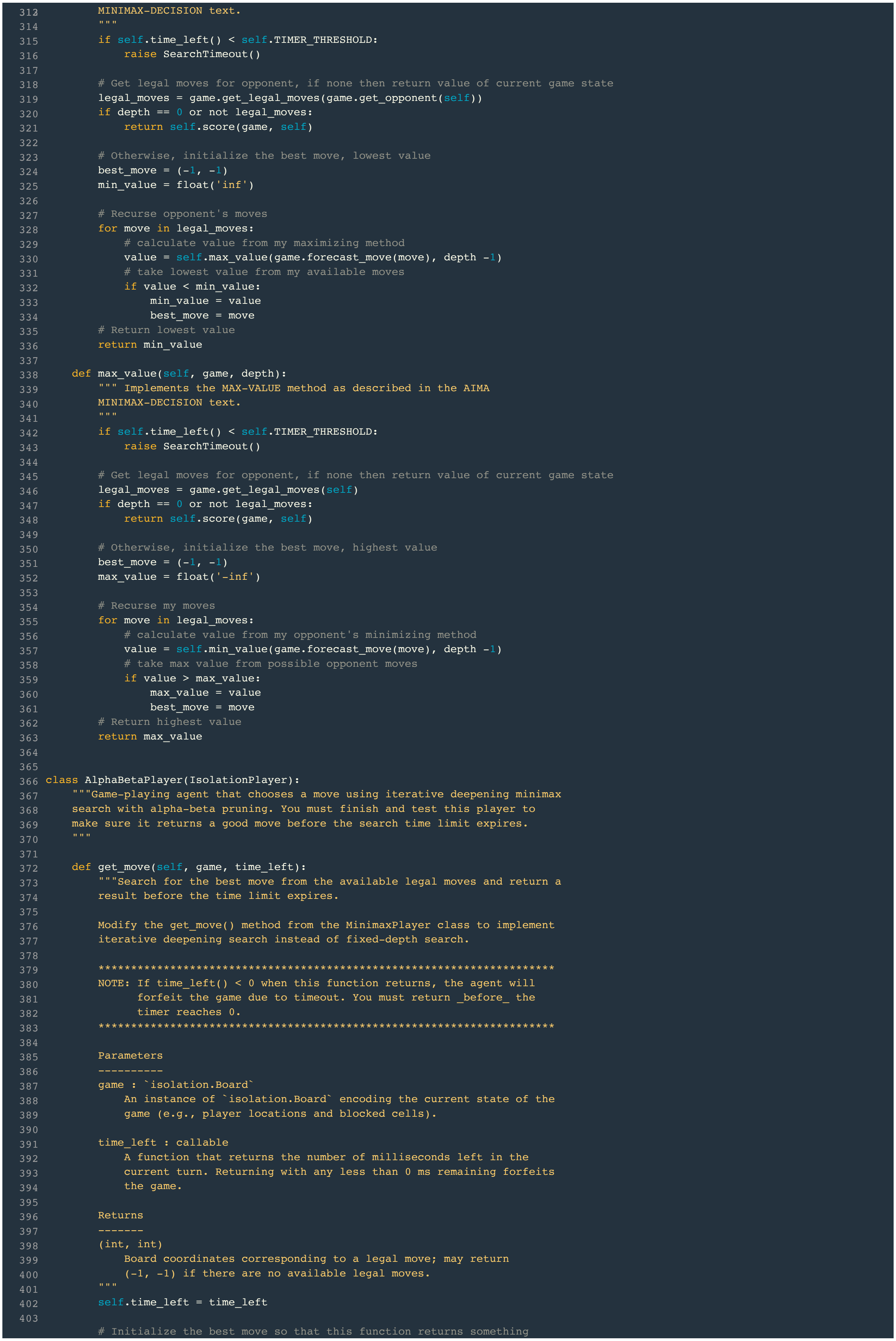
Superb! The implementation of this function is impressive.

```python
    """Calculates the heuristic value of a game state from the point of view
    of the given player.

    This function rewards moves toward the center of the board and penalizes
    moves along the edges and corners.

    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
```

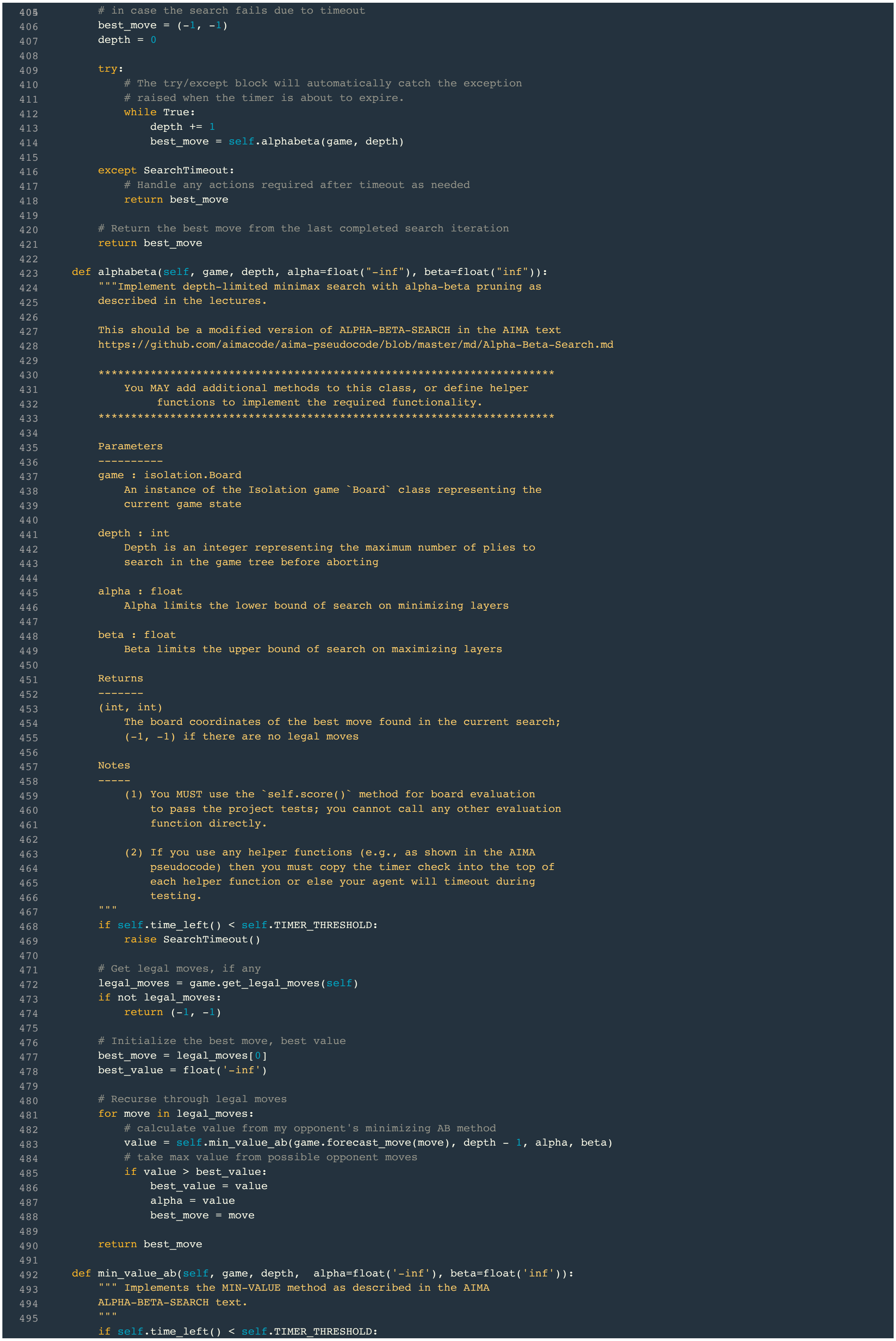
```python
        The heuristic value of the current game state to the specified player.
    """
    if game.is_loser(player):
        return float("-inf")

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

    # get player location
    x, y = game.get_player_location(player)

    score = 1  # default score for moves along outer edge of board

    # get list of empty spaces
    empty_spaces = game.get_blank_spaces()

    # reward moves in center of board at the beginning of game
    if (len(empty_spaces) > 40) and (x >= 2 and x <= 4) and (y >= 2 and y <= 4):
            score = 10
            return score

    # reward moves in center of board
    if (x >= 2 and x <= 4) and (y >= 2 and y <= 4):
        score = 5
        return score

    # reward moves in 2nd ring of board
    if (x == 1 or x == 5) and (y >= 1 and y <=5):
        score = 3
        return score

    # penalize moves in corners
    corners = [(0, 0), (0, 6), (6, 0), (6, 6)]
    if (x, y) in corners:
        score = 0

    return float(score)


class IsolationPlayer:
    """Base class for minimax and alphabeta agents -- this class is never
    constructed or tested directly.

    ********************  DO NOT MODIFY THIS CLASS  ********************

    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 sucessors of the
        current state.)

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

    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, timeout=10.):
        self.search_depth = search_depth
        self.score = score_fn
        self.time_left = None
        self.TIMER_THRESHOLD = timeout


class MinimaxPlayer(IsolationPlayer):
    """Game-playing agent that chooses a move using depth-limited minimax
    search. You must finish and test this player to make sure it properly uses
    minimax to return a good move before the search time limit expires.
    """

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

        **************  YOU DO NOT NEED TO MODIFY THIS FUNCTION  *************

        For fixed-depth search, this function simply wraps the call to the
        minimax method, but this method provides a common interface for all
        Isolation agents, and you will replace it in the AlphaBetaPlayer with
        iterative deepening search.

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

        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

        # Initialize the best move so that this function returns something
        # in case the search fails due to timeout
        best_move = (-1, -1)

        try:
            # The try/except block will automatically catch the exception
            # raised when the timer is about to expire.
            best_move = self.minimax(game, self.search_depth)

        except SearchTimeout:
            # Handle any actions required after timeout as needed
            return best_move

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

    def minimax(self, game, depth):
        """Implement depth-limited minimax search algorithm as described in
        the lectures.

        This should be a modified version of MINIMAX-DECISION in the AIMA text.
        https://github.com/aimacode/aima-pseudocode/blob/master/md/Minimax-Decision.md

        **********************************************************************
            You MAY add additional methods to this class, or define helper
                 functions to implement the required functionality.
        **********************************************************************

        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

        Returns
        -------
        (int, int)
            The board coordinates of the best move found in the current search;
            (-1, -1) if there are no legal moves

        Notes
        -----
            (1) You MUST use the `self.score()` method for board evaluation
                to pass the project tests; you cannot call any other evaluation
                function directly.

            (2) If you use any helper functions (e.g., as shown in the AIMA
                pseudocode) then you must copy the timer check into the top of
                each helper function or else your agent will timeout during
                testing.
        """
        if self.time_left() < self.TIMER_THRESHOLD:
            raise SearchTimeout()

        # Get legal moves, if any
        legal_moves = game.get_legal_moves(self)
        if not legal_moves:
            return (-1, -1)

        # Initialize the best move, best value
        best_move = legal_moves[0]
        best_value = float('-inf')

        # Recurse through legal moves
        for move in legal_moves:
            # calculate value of opponent's minimizing method
            value = self.min_value(game.forecast_move(move), depth - 1)
            # take max value from opponent's available moves
            if value > best_value:
                best_value = value
                best_move = move

        return best_move

    def min_value(self, game, depth):
        """ Implements the MIN-VALUE method as described in the AIMA
```



```
        MINIMAX-DECISION text.
        """
        if self.time_left() < self.TIMER_THRESHOLD:
            raise SearchTimeout()

        # Get legal moves for opponent, if none then return value of current game state
        legal_moves = game.get_legal_moves(game.get_opponent(self))
        if depth == 0 or not legal_moves:
            return self.score(game, self)

        # Otherwise, initialize the best move, lowest value
        best_move = (-1, -1)
        min_value = float('inf')

        # Recurse opponent's moves
        for move in legal_moves:
            # calculate value from my maximizing method
            value = self.max_value(game.forecast_move(move), depth -1)
            # take lowest value from my available moves
            if value < min_value:
                min_value = value
                best_move = move
        # Return lowest value
        return min_value

    def max_value(self, game, depth):
        """ Implements the MAX-VALUE method as described in the AIMA
        MINIMAX-DECISION text.
        """
        if self.time_left() < self.TIMER_THRESHOLD:
            raise SearchTimeout()

        # Get legal moves for opponent, if none then return value of current game state
        legal_moves = game.get_legal_moves(self)
        if depth == 0 or not legal_moves:
            return self.score(game, self)

        # Otherwise, initialize the best move, highest value
        best_move = (-1, -1)
        max_value = float('-inf')

        # Recurse my moves
        for move in legal_moves:
            # calculate value from my opponent's minimizing method
            value = self.min_value(game.forecast_move(move), depth -1)
            # take max value from possible opponent moves
            if value > max_value:
                max_value = value
                best_move = move
        # Return highest value
        return max_value


class AlphaBetaPlayer(IsolationPlayer):
    """Game-playing agent that chooses a move using iterative deepening minimax
    search with alpha-beta pruning. You must finish and test this player to
    make sure it returns a good move before the search time limit expires.
    """

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

        Modify the get_move() method from the MinimaxPlayer class to implement
        iterative deepening search instead of fixed-depth search.

        **********************************************************************
        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).

        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

        # Initialize the best move so that this function returns something
```



```
        # in case the search fails due to timeout
        best_move = (-1, -1)
        depth = 0

        try:
            # The try/except block will automatically catch the exception
            # raised when the timer is about to expire.
            while True:
                depth += 1
                best_move = self.alphabeta(game, depth)

        except SearchTimeout:
            # Handle any actions required after timeout as needed
            return best_move

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

    def alphabeta(self, game, depth, alpha=float("-inf"), beta=float("inf")):
        """Implement depth-limited minimax search with alpha-beta pruning as
        described in the lectures.

        This should be a modified version of ALPHA-BETA-SEARCH in the AIMA text
        https://github.com/aimacode/aima-pseudocode/blob/master/md/Alpha-Beta-Search.md

        **********************************************************************
            You MAY add additional methods to this class, or define helper
                 functions to implement the required functionality.
        **********************************************************************

        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

        alpha : float
            Alpha limits the lower bound of search on minimizing layers

        beta : float
            Beta limits the upper bound of search on maximizing layers

        Returns
        -------
        (int, int)
            The board coordinates of the best move found in the current search;
            (-1, -1) if there are no legal moves

        Notes
        -----
            (1) You MUST use the `self.score()` method for board evaluation
                to pass the project tests; you cannot call any other evaluation
                function directly.

            (2) If you use any helper functions (e.g., as shown in the AIMA
                pseudocode) then you must copy the timer check into the top of
                each helper function or else your agent will timeout during
                testing.
        """
        if self.time_left() < self.TIMER_THRESHOLD:
            raise SearchTimeout()

        # Get legal moves, if any
        legal_moves = game.get_legal_moves(self)
        if not legal_moves:
            return (-1, -1)

        # Initialize the best move, best value
        best_move = legal_moves[0]
        best_value = float('-inf')

        # Recurse through legal moves
        for move in legal_moves:
            # calculate value from my opponent's minimizing AB method
            value = self.min_value_ab(game.forecast_move(move), depth - 1, alpha, beta)
            # take max value from possible opponent moves
            if value > best_value:
                best_value = value
                alpha = value
                best_move = move

        return best_move

    def min_value_ab(self, game, depth,  alpha=float('-inf'), beta=float('inf')):
        """ Implements the MIN-VALUE method as described in the AIMA
        ALPHA-BETA-SEARCH text.
        """
        if self.time_left() < self.TIMER_THRESHOLD:
            raise SearchTimeout()

        # Get legal moves for the opponent, if any
        legal_moves = game.get_legal_moves(game.get_opponent(self))
        if depth == 0 or not legal_moves:
            return self.score(game, self)

        # Initialize best value for opponent
        min_value = beta

        # Recurse legal moves
        for move in legal_moves:
            # calculate value for each of my opponent's moves
            value = self.max_value_ab(game.forecast_move(move), depth - 1, alpha, beta)
            # return value if <= alpha
            if value <= alpha:
                return value
            # update min_value
            if value < min_value:
                min_value = value
            # update beta
            if value < beta:
                beta = value

        return min_value

    def max_value_ab(self, game, depth,  alpha=float('-inf'), beta=float('inf')):
        """ Implements the MAX-VALUE method as described in the AIMA
        ALPHA-BETA-SEARCH text.
        """
        if self.time_left() < self.TIMER_THRESHOLD:
            raise SearchTimeout()

        # Get my legal moves, if any
        legal_moves = game.get_legal_moves(self)
        if depth == 0 or not legal_moves:
            return self.score(game, self)

        # Initialize best value for me
        max_value = alpha

        # Recurse my legal moves
        for move in legal_moves:
            # calculate value from my opponent's minimizing method
            value = self.min_value_ab(game.forecast_move(move), depth - 1, alpha, beta)
            # return value if >= beta
            if value >= beta:
                return value
            # update max_value
            if value > max_value:
                max_value = value
            # update alpha
            if value > alpha:
                alpha = value

        return max_value
```

RETURN TO PATH

Rate this review

☆☆☆☆☆

Student FAQ Reviewer Agreement