UDACITY

Back to Artificial Intelligence Nanodegree

Build a Game Playing Agent

REVIEW

CODE REVIEW 7

HISTORY

my_custom_player.py 7

```
import math
import random

from sample_players import DataPlayer

class CustomPlayer(DataPlayer):
    """ Implement your own agent to play knight's Isolation
    The get_action() method is the only *required* method. You can modify
    the interface for get_action by adding named parameters with default
    values, but the function MUST remain compatible with the default
    interface.
    **********************************************************************
    NOTES:
    - You should **ONLY** call methods defined on your agent class during
    search; do **NOT** add or call functions outside the player class.
    The isolation library wraps each method of this class to interrupt
    search when the time limit expires, but the wrapper only affects
    methods defined on this class.
    - The test cases will NOT be run on a machine with GPU access, nor be
    suitable for using any other machine learning techniques.
    **********************************************************************
    """
    def get_action(self, state):
        """ Employ an adversarial search technique to choose an action
        available in the current state calls self.queue.put(ACTION) at least
        This method must call self.queue.put(ACTION) at least once, and may
        call it as many times as you want; the caller is responsible for
        cutting off the function after the search time limit has expired.
        See RandomPlayer and GreedyPlayer in sample_players for more examples.
        **********************************************************************
        NOTE:
        - The caller is responsible for cutting off search, so calling
```

```
33          get_action() from your own code will create an infinite loop!
34          Refer to (and use!) the Isolation.play() function to run games.
35          ************************************************************
36          """
37          # TODO: Replace the example implementation below with your own search
38          #       method by combining techniques from lecture
39          #
40          # EXAMPLE: choose a random move without any search--this function MUST
41          #          call self.queue.put(ACTION) at least once before time expires
42          #          (the timer is automatically managed for you)
43          depth_limit = 100
44          if state.ply_count < 2:
45              self.queue.put(random.choice(state.actions()))
46          else:
47              for depth in range(1, depth_limit + 1):
48                  action = self.alpha_beta(state, depth)
49                  if action is not None:
50                      self.queue.put(action)
```

AWESOME

`get_action()` method calls `self.queue.put()` correctly, well done.

```
51
52      def alpha_beta(self, state, depth):
53          """Alpha beta pruning with iterative deepening"""
54          beta = float("inf")
55          best_score = float("-inf")
56          best_move = None
57          for a in state.actions():
58              v = self.min_value(state.result(a), best_score, beta, depth - 1)
59              if v > best_score:
60                  best_score = v
61                  best_move = a
62          # writing depth and ply count info
63          #DEBUG_INFO = open("depth,ply_count.txt", "a")
64          #DEBUG_INFO.write(str(depth) + ", " + str(state.ply_count) + "\n")
65          #DEBUG_INFO.close()
66          return best_move
67
```

AWESOME

Nice work in implementing `alphabeta` function! I can see that it keeps track of the best and worst valı appropriately. Good job!

```
68      def min_value(self, state, alpha, beta, depth):
69          if depth <= 0:
70              return self.custom_heuristics_2(state)
71          if state.terminal_test():
72              return state.utility(self.player_id)
73
74          v = float("inf")
75          for a in state.actions():
76              v = min(v, self.max_value(state.result(a), alpha, beta, depth - 1))
77              if v <= alpha:
78                  return v
79              beta = min(beta, v)
80          return v
```

SUGGESTION

Notes

Check this discussion to have an insight on how we can boost our alphabeta with iterative deepening. want to use iterative deepening. 😃

```
81
82      def max_value(self, state, alpha, beta, depth):
83          if depth <= 0:
84              return self.custom_heuristics_2(state)
85          if state.terminal_test():
86              return state.utility(self.player_id)
87
88          v = float("-inf")
89          for a in state.actions():
90              v = max(v, self.min_value(state.result(a), alpha, beta, depth - 1))
91              if v >= beta:
92                  return v
93              alpha = max(alpha, v)
94          return v
95
96      def score(self, state):
97          """ own moves - opponent moves heuristic """
98          own_loc = state.locs[self.player_id]
99          opp_loc = state.locs[1 - self.player_id]
```

SUGGESTION

Notes

Another way to implement this is to use the custom score after first moves are completed and high sc available. If the game is coming to a close and own move is less than 5, it becomes a less aggressive p See code below for suggestions:

```
def score(self, state):
    own_liberties = state.liberties(state.locs[self.player_id])
    opp_liberties = state.liberties(state.locs[1 - self.player_id])
    own_moves = len(own_liberties)
    opp_moves = len(opp_liberties)
        if own_moves < 5:
            return (1.5 * own_moves) - (opp_moves)
        else:
            return (own_moves) - (1.5 * opp_moves)
```

```
100         own_liberties = state.liberties(own_loc)
101         #Weight the Baseline
102         opp_liberties = state.liberties(opp_loc)*(4)
103         #opp_liberties = state.liberties(opp_loc)
104         return len(own_liberties) - len(opp_liberties)
105
106     #def custom_heuristics(self, state):
107     #    """Linear combinations of features can be effective.
108     #    Features for Isolation can include the ply, the distance between the play
109     #    distance from the edge (or center), and more (be creative)."""
110     #    own_loc = state.locs[self.player_id]
```

```
111     #    opp_loc = state.locs[1 - self.player_id]
112     #    player_distance = self.manhattan_distance(self.get_coordinates(own_loc),
113     #    own_moves_minus_opp_moves = self.score(state)
```

SUGGESTION

Notes

Some codes were a big help to us in some ways. But as a developer, it is a best practice to remove con code clean and to avoid giving confusion to other developers reading the code. Check out these links r

- Why Comment-out Code?
- Is it bad practice to leave commented-out code?

```
114
115     #    if state.ply_count < 30:
116     #        # chase the opponent for the first 30 moves
117     #        return own_moves_minus_opp_moves - player_distance
118     #    elif state.ply_count < 45:
119     #        # move away from the opponent and the center (presumably using up cor
120     #        return player_distance + own_moves_minus_opp_moves + self.distance_to
121     #    else:
122     #        # endgame get close to the opponent and the center
123     #        return 0 - player_distance + own_moves_minus_opp_moves - self.distanc
124
125     def custom_heuristics_2(self, state):
126         """Linear combinations of features can be effective.
127         Features for Isolation can include the ply, the distance between the playe
128         distance from the edge (or center), and more (be creative)."""
129         own_loc = state.locs[self.player_id]
130         opp_loc = state.locs[1 - self.player_id]
131         player_distance = self.manhattan_distance(self.get_coordinates(own_loc), s
132         own_moves_minus_opp_moves = self.score(state)
133
134         if state.ply_count < 30:
135             # chase the opponent for the first 30 moves
136             return own_moves_minus_opp_moves - player_distance
137         elif state.ply_count < 50:
138             # stay close to the opponent and the center up to 50 moves
139             return 0 - player_distance + own_moves_minus_opp_moves + self.distance
140         else:
141             # endgame - stay away from the oponent but still try to stay close to
142             # increace the effect of own_moves_minus_opp_moves value times 2 since
143             # we'd like to keep its influence a bit higher in the endgame
144             return player_distance + (own_moves_minus_opp_moves * 2) - self.distan
145
146     def manhattan_distance(self, loc1, loc2):
147         """Returns the manhattan distance between two points (loc1 and loc2)"""
148         return abs(loc1[0] - loc2[0]) + abs(loc1[1] - loc2[1])
149
```

AWESOME

I like the simplicity of the custom heuristic implementation. This really helps the game to search deepe increasing winning chances.

```
150     def get_coordinates(self, int_location):
```

```
151         """Gets x,y coordinates out of an integer location"""
152         x = int_location % 13 # get column
153         y = math.floor(int_location/13) # get row
154         return x, y
155 
156     def distance_to_center(self, location):
157         """Manhattan distance to center from given location"""
158         return self.manhattan_distance(location, (5, 4))
159 
160     def own_moves(self, state):
161         own_loc = state.locs[self.player_id]
162         own_liberties = state.liberties(own_loc)
163         return len(own_liberties)
```

SUGGESTION

Notes

You can also add a function that the score can use at the start of the game where it returns a high valu available. Here's a code suggestion:

```
def start_score(self, state):
    own_liberties = state.liberties(state.locs[self.player_id])
    center_x, center_y = 6, 4
    center = (center_x, center_y)
    if center in own_liberties:
        return float("inf")
    else:
        return float("-inf")
```

RETURN TO PATH

Rate this review