

ARTIFICIAL INTELLIGENCE NANODEGREE

PROJECT-2

ISOLATION AGENT

heuristic_analysis

-GRANDHE SAI CHETAN

In order to win against 'AB_improved', We use the heuristic no.of my_moves- no.of opp_moves and simply play with the weights for no.of my_moves and the no.of opp_moves. In the final run, I came up with the following custom_score functions/weights:

#Code

```
def custom_score(game, player):
```

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

```
    This should be the best heuristic function for your project submission.
```

```
    Note: this function should be called from within a Player instance as
```

```
    `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.
```

```
    """
```

```
    # TODO: finish this function!
```

```
    if game.is_loser(player):
```

```
return float("-inf")
```

```
if game.is_winner(player):
```

```
    return float("inf")
```

```
my_moves = len(game.get_legal_moves(player))
```

```
opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
```

```
return float(my_moves - 5 * opp_moves)
```

```
def custom_score_2(game, player):
```

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

Note: this function should be called from within a Player instance as

``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.

"""

TODO: finish this function!

if game.is_loser(player):

return float("-inf")

if game.is_winner(player):

return float("inf")

my_moves = len(game.get_legal_moves(player))

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

return float(my_moves - 4 * opp_moves)

def custom_score_3(game, player):

"""Calculate the heuristic value of a game state from the point of view
 of the given player.

Note: this function should be called from within a Player instance as

``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")
```

```
my_moves = len(game.get_legal_moves(player))
```

```
opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
```

```
return float(3 * my_moves - opp_moves)
```

Goal of of all the player is to maximize their own custom score function, in order to ultimately win the game. The three score functions which are implemented are supposed to maximize the no.of my_moves and to minimize no.of opp_moves at the same time. But each function defines a different weight/importance to the remaining own moves respectively opponent's moves.

Hence if the weights of the remaining my_moves are high in comparison to the weights of the opponent's remaining moves, the player tries to move to fields which offer mainly many

remaining moves, which means to move away from the opponent. If the weights are distributed the other way round, the player will try to "chase" the opponent and isolate him.

Upon running tournament.py:-

```
"""Estimate the strength rating of a student defined heuristic by competing
against fixed-depth minimax and alpha-beta search agents in a round-robin
tournament.
```

NOTE: All agents are constructed from the student CustomPlayer implementation, so any errors present in that class will affect the outcome.

The student agent plays a number of "fair" matches against each test agent.

The matches are fair because the board is initialized randomly for both players, and the players play each match twice -- once as the first player and once as the second player. Randomizing the openings and switching the player order corrects for imbalances due to both starting position and initiative.

```
"""
```

```
import itertools
```

```
import random
```

```
import warnings
```

```
from collections import namedtuple
```

```
from isolation import Board
```

```
from sample_players import (RandomPlayer, open_move_score,
```

```
        improved_score, center_score)

from game_agent import (MinimaxPlayer, AlphaBetaPlayer, custom_score,
                        custom_score_2, custom_score_3)
```

```
NUM_MATCHES = 5 # number of matches against each opponent
```

```
TIME_LIMIT = 150 # number of milliseconds before timeout
```

```
DESCRIPTION = """
```

```
This script evaluates the performance of the custom_score evaluation
function against a baseline agent using alpha-beta search and iterative
deepening (ID) called `AB_Improved`. The three `AB_Custom` agents use
ID and alpha-beta search with the custom_score functions defined in
game_agent.py.
```

```
"""
```

```
Agent = namedtuple("Agent", ["player", "name"])
```

```
def play_round(cpu_agent, test_agents, win_counts, num_matches):
```

```
    """Compare the test agents to the cpu agent in "fair" matches.
```

```
"Fair" matches use random starting locations and force the agents to
play as both first and second player to control for advantages resulting
from choosing better opening moves or having first initiative to move.
```

```
"""
```

```
timeout_count = 0
```

```
forfeit_count = 0
```

```
for _ in range(num_matches):
```

```
    games = sum([[Board(cpu_agent.player, agent.player),
```

```
                  Board(agent.player, cpu_agent.player)]
```

```
                for agent in test_agents], [])
```

```
    # initialize all games with a random move and response
```

```
    for _ in range(2):
```

```
        move = random.choice(games[0].get_legal_moves())
```

```
        for game in games:
```

```
            game.apply_move(move)
```

```
    # play all games and tally the results
```

```
    for game in games:
```

```
        winner, _, termination = game.play(time_limit=TIME_LIMIT)
```

```
        win_counts[winner] += 1
```

```
    if termination == "timeout":
```

```
        timeout_count += 1
```

```
    elif termination == "forfeit":
```

```
        forfeit_count += 1
```



```
return timeout_count, forfeit_count
```

```
def update(total_wins, wins):
```

```
    for player in total_wins:
```

```
        total_wins[player] += wins[player]
```

```
    return total_wins
```

```
def play_matches(cpu_agents, test_agents, num_matches):
```

```
    """Play matches between the test agent and each cpu_agent individually. """
```

```
    total_wins = {agent.player: 0 for agent in test_agents}
```

```
    total_timeouts = 0.
```

```
    total_forfeits = 0.
```

```
    total_matches = 2 * num_matches * len(cpu_agents)
```

```
    print("\n{:^9}{:~13}".format("Match #", "Opponent") + ".join(['{:~13}'.format(x[1].name) for x  
in enumerate(test_agents)]))
```

```
    print("{:~9}{:~13}".format("", "") + ' '.join(['{:~5}| {:~5}'.format("Won", "Lost") for x in  
enumerate(test_agents)]))
```

```
    for idx, agent in enumerate(cpu_agents):
```

```
        wins = {key: 0 for (key, value) in test_agents}
```

```
        wins[agent.player] = 0
```

```

print("{!s:^9}{: ^13}".format(idx + 1, agent.name), end="", flush=True)

counts = play_round(agent, test_agents, wins, num_matches)
total_timeouts += counts[0]
total_forfeits += counts[1]
total_wins = update(total_wins, wins)
_total = 2 * num_matches
round_totals = sum([[wins[agent.player], _total - wins[agent.player]]
                    for agent in test_agents], [])
print(' ' + ' '.join([
    '{:^5}| {: ^5}'.format(
        round_totals[i], round_totals[i+1]
    ) for i in range(0, len(round_totals), 2)
]))

print("-" * 74)
print('{!s:^9}{: ^13}'.format("", "Win Rate:") +
    ".join([
        '{:^13}'.format(
            "{:.1f}%".format(100 * total_wins[x[1].player] / total_matches)
        ) for x in enumerate(test_agents)
    ]))

```

```

if total_timeouts:

    print("\nThere were {} timeouts during the tournament -- make sure " +
          "your agent handles search timeout correctly, and consider " +
          "increasing the timeout margin for your agent.\n").format(
        total_timeouts))

if total_forfeits:

    print("\nYour agents forfeited {} games while there were still " +
          "legal moves available to play.\n").format(total_forfeits))


def main():

    # Define two agents to compare -- these agents will play from the same
    # starting position against the same adversaries in the tournament
    test_agents = [
        Agent(AlphaBetaPlayer(score_fn=improved_score), "AB_Improved"),
        Agent(AlphaBetaPlayer(score_fn=custom_score), "AB_Custom"),
        Agent(AlphaBetaPlayer(score_fn=custom_score_2), "AB_Custom_2"),
        Agent(AlphaBetaPlayer(score_fn=custom_score_3), "AB_Custom_3")
    ]

    # Define a collection of agents to compete against the test agents
    cpu_agents = [
        Agent(RandomPlayer(), "Random"),

```

```
    Agent(MinimaxPlayer(score_fn=open_move_score), "MM_Open"),
    Agent(MinimaxPlayer(score_fn=center_score), "MM_Center"),
    Agent(MinimaxPlayer(score_fn=improved_score), "MM_Improved"),
    Agent(AlphaBetaPlayer(score_fn=open_move_score), "AB_Open"),
    Agent(AlphaBetaPlayer(score_fn=center_score), "AB_Center"),
    Agent(AlphaBetaPlayer(score_fn=improved_score), "AB_Improved")
]
```

```
print(DESCRIPTION)

print("{:^74}".format("*****"))

print("{:^74}".format("Playing Matches"))

print("{:^74}".format("*****"))

play_matches(cpu_agents, test_agents, NUM_MATCHES)
```

```
if __name__ == "__main__":
    main()
```

TEXTUAL FORM OF THE OUTPUT:-

Playing Matches:-

Opponent AB_Improved AB_Custom AB_Custom_2 AB_Custom_3

	Won Lost	Won Lost	Won Lost	Won Lost
1 Random	9 1	9 1	6 4	9 1
2 MM_Open	6 4	6 4	6 4	8 2
3 MM_Center	8 2	8 2	9 1	6 4
4 MM_Improved	5 5	6 4	8 2	5 5
5 AB_Open	6 4	5 5	6 4	6 4
6 AB_Center	5 5	8 2	4 6	6 4
7 AB_Improved	5 5	5 5	4 6	7 3

Win Rate:	62.9%	67.1%	61.4%	67.1%
-----------	-------	-------	-------	-------

We see that AB_Custom and AB_Custom_3 score function outperform AB_improved. AB_custom is quite aggressive and tries to isolate the opponent on the other hand AB_custom_3 is quite the opposite. It seems that it is good to have a strategy which is either very aggressive or very defensive. It is very interesting to see that the defensive player (AB_custom_3) clearly wins against AB_improved in a direct match, whereas the aggressive player does not. I chose AB_Custom_3 player as the best player, because:

- it has together with the player AB_custom the highest win rate.
- the speed of the algorithm was fast enough to beat AB_improved.
- it beats compared to AB_custom_3, AB_Improved more often.