HEURISTIC ANALYSIS

For Isolation game playing agent

SUBMITTED BY: AYUSH CHAUHAN

(Part of Artificial Intelligence Nanodegree)

Synopsis

In this project, I have developed an adversarial search agent to play the game "Isolation". Isolation is a deterministic, two-player game of perfect information in which the players alternate turns moving a single piece from one cell to another on a board. Whenever either player occupies a cell, that cell becomes blocked for the remainder of the game. The first player with no remaining legal moves loses, and the opponent is declared the winner. These rules are implemented in the isolation.

This project uses a version of Isolation where each agent is restricted to L-shaped movements (like a knight in chess) on a rectangular grid (like a chess or checkerboard). The agents can move to any open cell on the board that is 2-rows and 1-column or 2-columns and 1-row away from their current position on the board. Movements are blocked at the edges of the board (the board does not wrap around), however, the player can "jump" blocked or occupied spaces (just like a knight in chess).

Additionally, agents will have a fixed time limit each turn to search for the best move and respond. If the time limit expires during a player's turn, that player forfeits the match, and the opponent wins.

Custom Heuristics

1) **Heuristic 1 – Maximizing player moves**

This heuristic is based on the logic that the difference between opponent and current player moves should be maximum

```
def maxDiffOppOwnMoves(game, player):
      """
      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.

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

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

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

      return float(own_moves - opp_moves)
```

2) **Heuristic 2 – Weighted maximum of player moves**

This heuristic is the weighted version of heuristic 1. This heuristic maximises the difference between player and weighted opponent moves. This result in selecting more moves that will make our agent wins more matches.

```
def weightedmaxDiffOppOwnMoves(game, player):
   """
   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.

   Returns
   ----------
   float
   The heuristic value of the current game state
   """

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

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

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

   return float(own_moves - 2*opp_moves)
```

3) Heuristic 3 – Maximizing ratio of player moves to opposition moves

This heuristic is based on the logic that the ratio of player moves to opposition moves should be maximized.

```
def maxRatioOwnToOppMoves(game, player):
      """
      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.

      Returns
      ----------
      float
      The heuristic value of the current game state
      """

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

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

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

      if not opp_moves:
            return float("inf")

      return float(own_moves/opp_moves)
```

4) <u>Heuristic 4 – Weighted maximum ratio</u>

This heuristic is the weighted combination of maximum ratio of player moves to opposition moves and minimum ratio of opposition moves to player moves.

```
def weightedMaxRatio(game, player):
      """
      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.

      Returns
      ----------
      float
      The heuristic value of the current game state
      """

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

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

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

      if not own_moves:
            return float("-inf")

      if not opp_moves:
            return float("inf")

      return float(own_moves/opp_moves - 2*opp_moves/own_moves)
```

Evaluating Heuristics

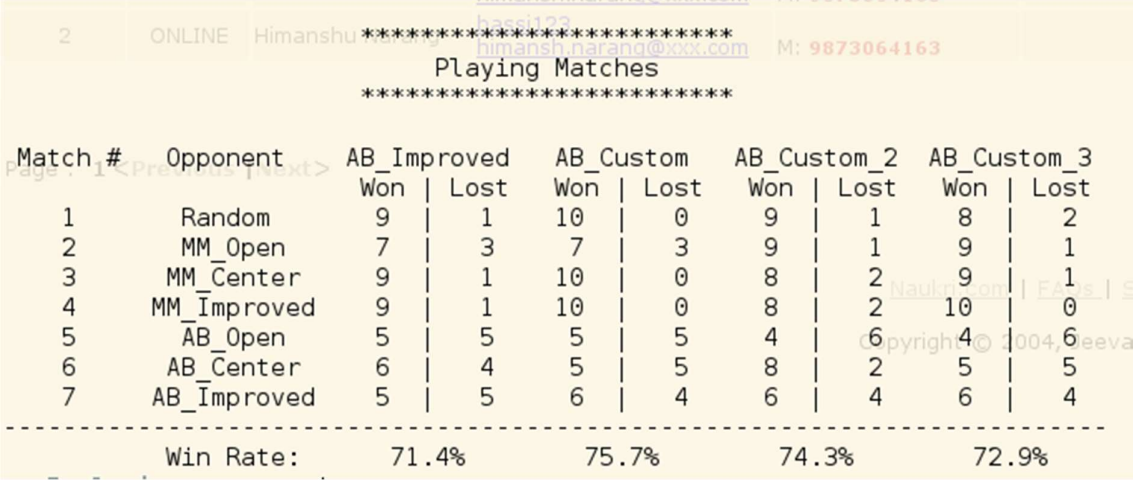
The tournament.py script is used to evaluate the effectiveness of custom heuristics. The script measures relative performance of agent (named "Student" in the tournament) in a round-robin tournament against several other pre-defined agents. The Student agent uses time-limited Iterative Deepening along with your custom heuristics.

The performance of time-limited iterative deepening search is hardware dependent (faster hardware is expected to search deeper than slower hardware in the same amount of time). The script controls for these effects by also measuring the baseline performance of an agent called "ID_Improved" that uses Iterative Deepening and the improved_score heuristic defined in sample_players.py. My goal was to develop a heuristic such that Student outperforms ID_Improved.

The tournament opponents are listed below.

- Random: An agent that randomly chooses a move each turn.
- MM_Open: MinimaxPlayer agent using the open_move_score heuristic with search depth 3
- MM_Center: MinimaxPlayer agent using the center_score heuristic with search depth 3
- MM_Improved: MinimaxPlayer agent using the improved_score heuristic with search depth 3
- AB_Open: AlphaBetaPlayer using iterative deepening alpha-beta search and the open_move_score heuristic
- AB_Center: AlphaBetaPlayer using iterative deepening alpha-beta search and the center_score heuristic
- AB_Improved: AlphaBetaPlayer using iterative deepening alpha-beta search and the improved_score heuristic

Result



```
**************************
  Playing Matches
**************************

 Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random       9  |   1     10  |   0     9  |   1     8  |   2
    2       MM_Open      7  |   3      7  |   3     9  |   1     9  |   1
    3      MM_Center     9  |   1     10  |   0     8  |   2     9  |   1
    4     MM_Improved    9  |   1     10  |   0     8  |   2    10  |   0
    5       AB_Open      5  |   5      5  |   5     4  |   6     4  |   6
    6      AB_Center     6  |   4      5  |   5     8  |   2     5  |   5
    7     AB_Improved    5  |   5      6  |   4     6  |   4     6  |   4
--------------------------------------------------------------------------
          Win Rate:      71.4%        75.7%        74.3%        72.9%
```

All the custom heuristics performed better than the AB_IMPROVED by a good margin as it can been seen in the above image