Isolation Heuristic Analysis

Ben Steadman

Table of Contents

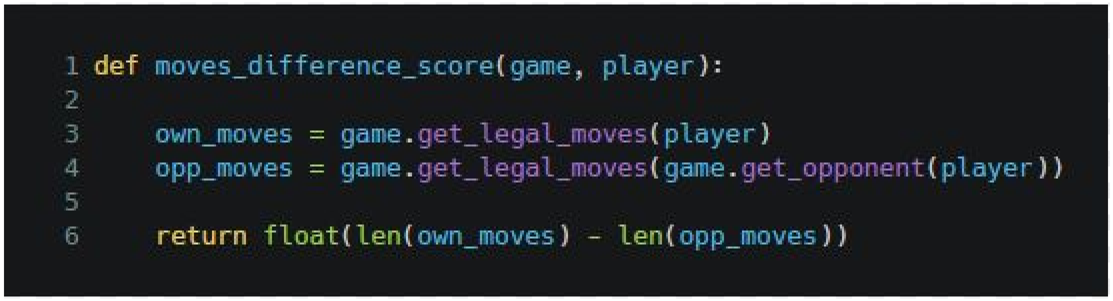
Baseline Heuristic

The basic heuristic for evaluating game states in Isolation revolves around maximising the moves of the player relative to their opponent:

$$Utility(game\ state) = |Moves(Player)| - |Moves(Opponent)|$$

This evaluation is surprisingly effective and ,due to its simplicity, very fast to compute. However, it is not perfect, as it doesn't take into account any factors specific to the game such as move-shape and the relative value of the possible moves (it may not matter if a player has many available moves if they all lead to being trapped and therefore losing in the next ply).

Basic Implementation:



```python
def moves_difference_score(game, player):

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

    return float(len(own_moves) - len(opp_moves))
```

N.B: All example code is simplified to remove code unnecessary to understanding the heuristics. Refer to the project source for full code.

Custom Heuristics

Improving on the basic **moves_difference_score()** heuristic requires taking into account more specific factors about the game.

Avoiding Corners

In Isolation the aim is to **not get trapped**, so it follows logically that the player should try to avoid any moves that may place them in any of the four corners of the board. When in a corner, the possible directions of movement for the player are greatly reduced:

↖	↑	↗
←	p	→
↙	↓	↘

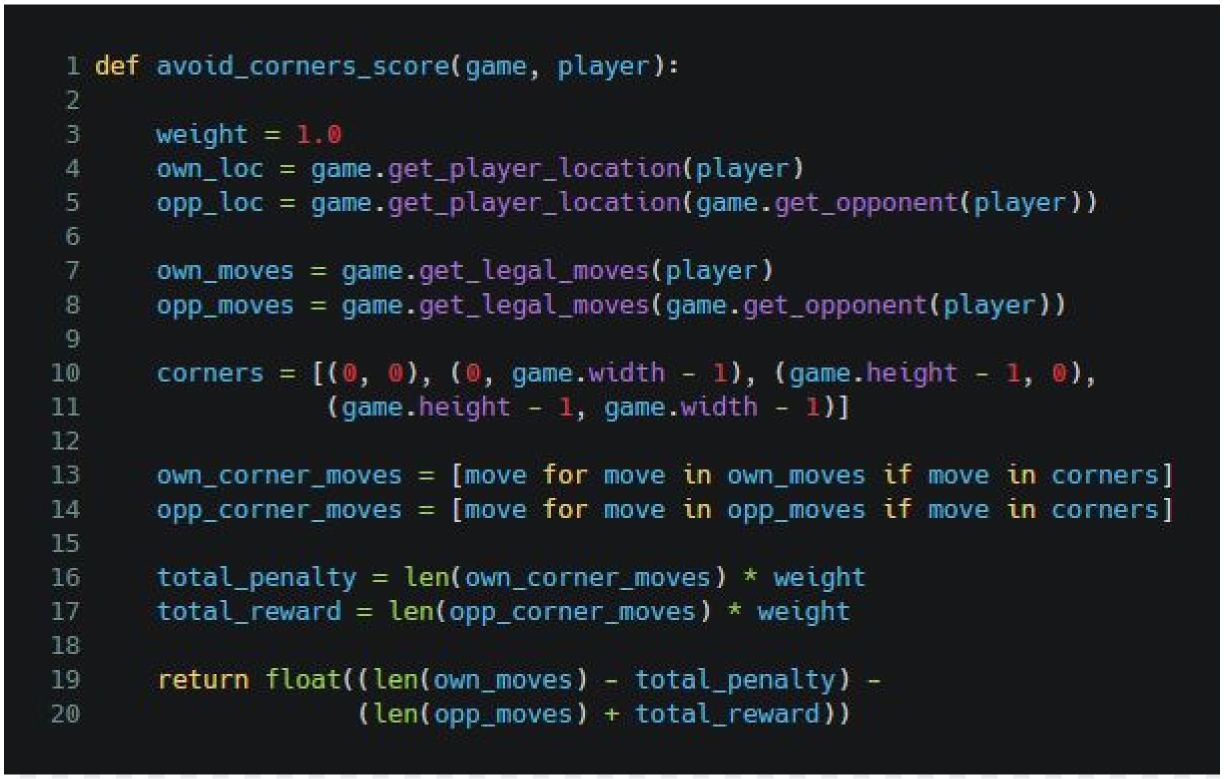
p	→
↓	↘

p = player location

For a given game state, the heuristic **penalises** the player for each possible move that would place the player in a corner and **rewards** the player for each of their *opponents* moves which place them in a corner:

$$W = weight\ per\ corner\ move$$
$$penalty\ =\ |CornerMoves(Player)| \times W$$
$$reward\ =\ |CornerMoves(Opponent)| \times W$$
$$Utility(game\ state)\ = (|Moves(Player)|\ -\ penalty)\ -\ (|Moves(Opponent)| +\ reward)$$

Implementation:



```python
def avoid_corners_score(game, player):

    weight = 1.0
    own_loc = game.get_player_location(player)
    opp_loc = game.get_player_location(game.get_opponent(player))

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

    corners = [(0, 0), (0, game.width - 1), (game.height - 1, 0),
               (game.height - 1, game.width - 1)]

    own_corner_moves = [move for move in own_moves if move in corners]
    opp_corner_moves = [move for move in opp_moves if move in corners]

    total_penalty = len(own_corner_moves) * weight
    total_reward = len(opp_corner_moves) * weight

    return float((len(own_moves) - total_penalty) -
                 (len(opp_moves) + total_reward))
```

Chase Opponent

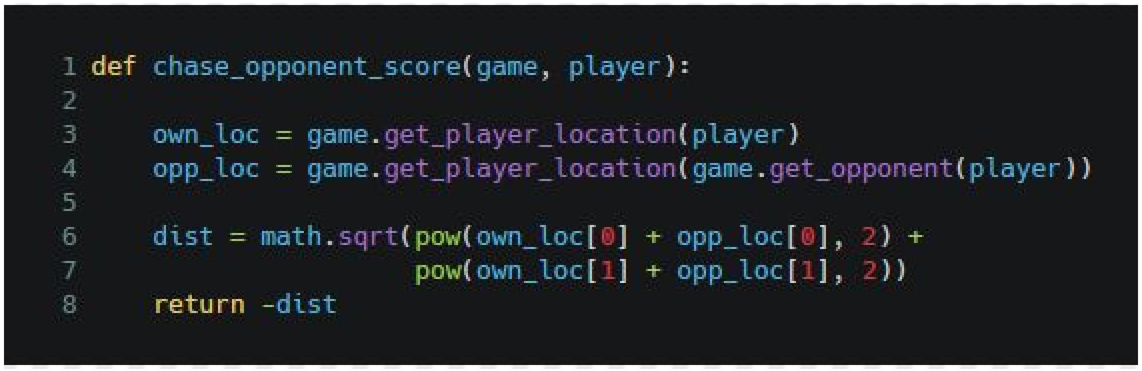
A simple, yet effective strategy for trapping one's opponent in Isolation is simple to **chase** them. That is, the player should always choose moves which places them closest to their opponent. On each turn, the player is blocking their opponents potential moves simply by being close to them:

$$d = \sqrt{(player_x + opponent_x)^2 + (player_y + opponent_y)^2}$$

$$Utility(game\ state) = -d$$

The **negation** of the distance is returned as the utility of the game state, meaning that the further away that two players are the lower the score (further from 0) and the closer the player the higher the score (closer to 0).

Implementation:

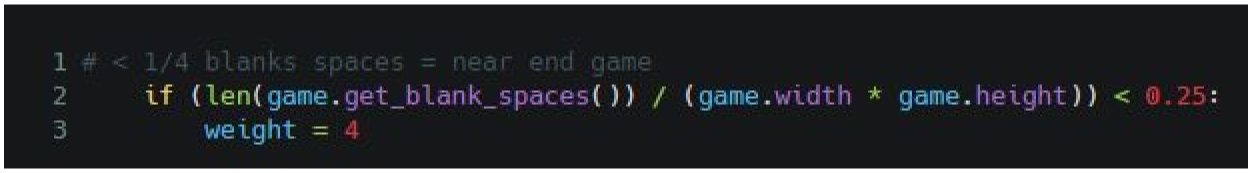


```python
def chase_opponent_score(game, player):

    own_loc = game.get_player_location(player)
    opp_loc = game.get_player_location(game.get_opponent(player))

    dist = math.sqrt(pow(own_loc[0] + opp_loc[0], 2) +
                     pow(own_loc[1] + opp_loc[1], 2))
    return -dist
```

Weighted Chase Opponent

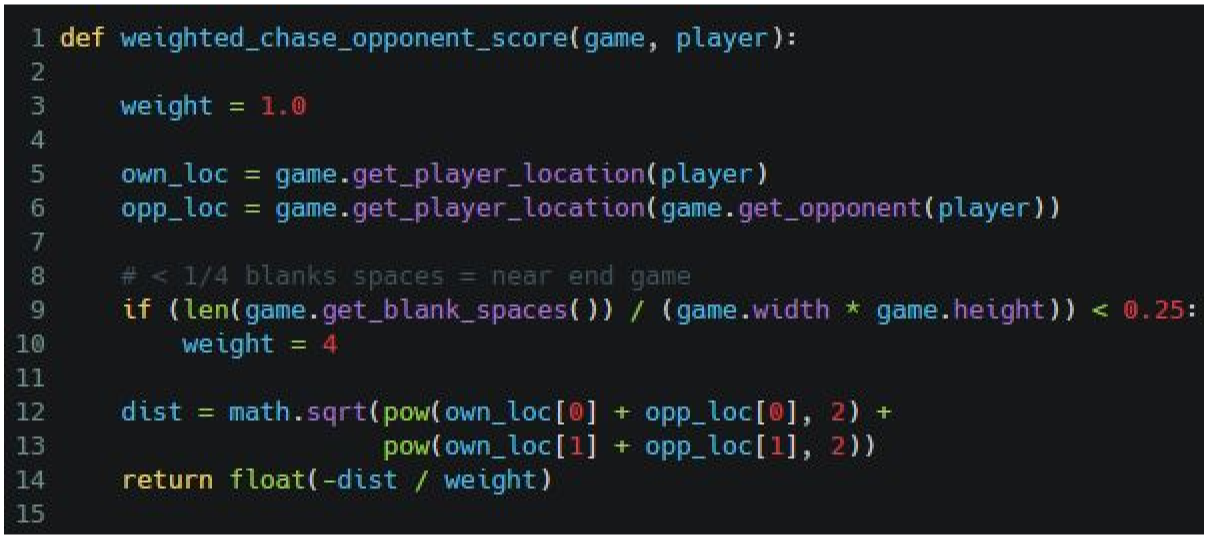
This heuristic is the same as Chase Opponent, except that it is weighted in order to heavily encourage the agent to get as close to the player as possible **towards the end of the game.** As the game progresses, more squares in the board will become ‘used’ and therefore illegal locations for moves. Therefore, being closer to the opponent towards the end of the game has a much higher chance of actually **trapping them completely.**

The weight is adjusted to a higher value when the board is **75%** filled:



```
# < 1/4 blanks spaces = near end game
    if (len(game.get_blank_spaces()) / (game.width * game.height)) < 0.25:
        weight = 4
```

Implementation:



```
def weighted_chase_opponent_score(game, player):

    weight = 1.0

    own_loc = game.get_player_location(player)
    opp_loc = game.get_player_location(game.get_opponent(player))

    # < 1/4 blanks spaces = near end game
    if (len(game.get_blank_spaces()) / (game.width * game.height)) < 0.25:
        weight = 4

    dist = math.sqrt(pow(own_loc[0] + opp_loc[0], 2) +
                     pow(own_loc[1] + opp_loc[1], 2))
    return float(-dist / weight)

```

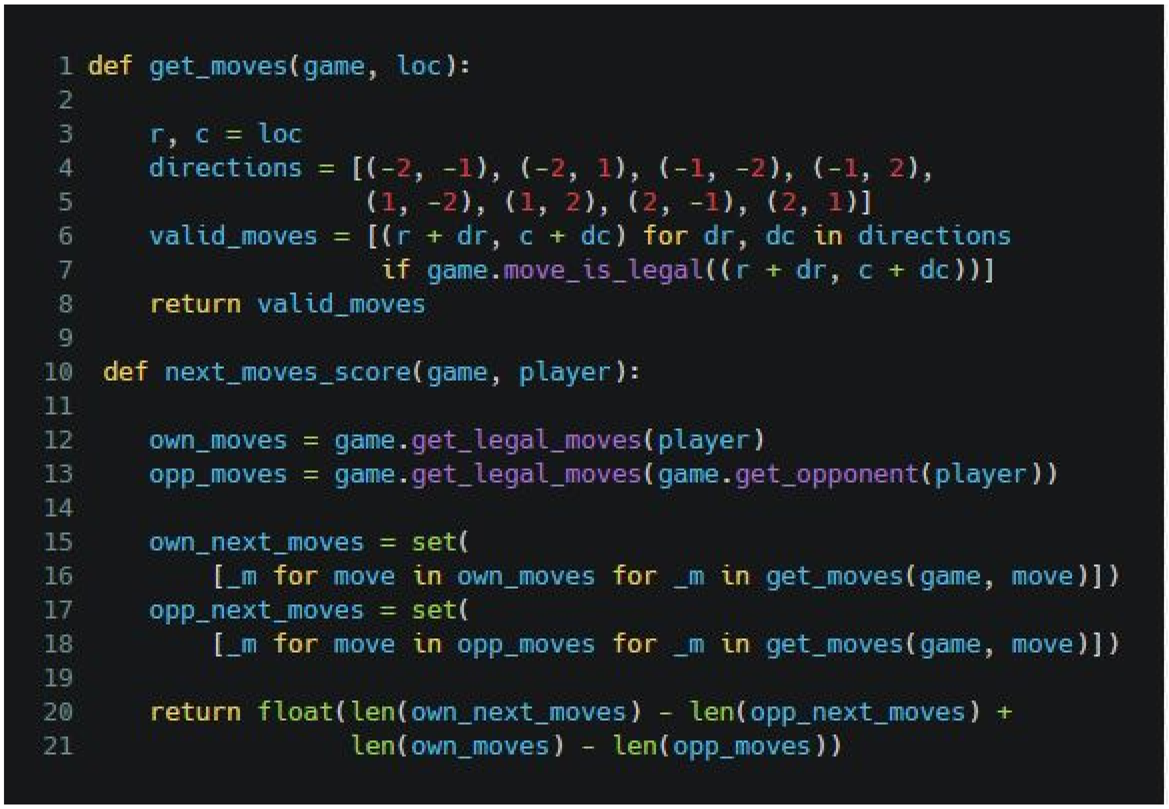
Next Moves

Being trapped is equivalent to there being 0 possible moves and having 1 possible move means being nearly trapped and so on up to the maximum number of moves and being as ‘not trapped’ as possible. Therefore it holds that the more possible moves a player has in a given board state, the less ‘trapped’ that player is and therefore the more value that move has.

This heuristic uses the the total number of possible moves in the next ply after the move being evaluated is chosen for each player to augment the basic heuristic of difference between number of moves. This **rewards** mobility for the player and **penalises** mobility for the opponent:

$$GetMoves(loc) \ \rightarrow returns\ list\ of\ possible\ moves\ from\ given\ location$$
$$NextMoves(Player) \ = \ \{nextMove\} \ \forall\, nextMove\ \varepsilon\ GetMoves(move) \ \forall\, move\ \varepsilon\ Moves(Player)$$
$$NextMoves(Opponent) \ = \ \{nextMove\} \ \forall\, nextMove\ \varepsilon\ GetMoves(move) \ \forall\, move\ \varepsilon\ Moves(Opponent)$$
$$nextMovesWeight = \ |NextMoves(Player)| \ - \ |NextMoves(Opponent)|$$
$$Utility(game\ state) \ = nextMovesWeight \ + \ |Moves(Player) - |Moves(Opponent)|$$

Implementation:



```
def get_moves(game, loc):

    r, c = loc
    directions = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),
                  (1, -2), (1, 2), (2, -1), (2, 1)]
    valid_moves = [(r + dr, c + dc) for dr, dc in directions
                   if game.move_is_legal((r + dr, c + dc))]
    return valid_moves

 def next_moves_score(game, player):

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

    own_next_moves = set(
        [_m for move in own_moves for _m in get_moves(game, move)])
    opp_next_moves = set(
        [_m for move in opp_moves for _m in get_moves(game, move)])

    return float(len(own_next_moves) - len(opp_next_moves) +
                 len(own_moves) - len(opp_moves))
```

Evaluating Custom Heuristics

Each custom heuristic is tested using a script which ‘plays’ a round-robin tournament against fixed-depth minimax and alpha-beta search agents using different heuristics. The strength/success of the heuristic is then estimated by the **percentage of wins** achieved by the agents using the heuristic over the course of the tournament.

Competing Heuristics

Each custom heuristic is tested against agents using one of the following heuristics:

- open_move_score
    - Score equal to the number of moves open for the current player on the board
- center_score
    - Score equal to square of the distance from the center of the board to the position of the player.
- improved_score
    - Same as **moves_difference_score()** described earlier.
    - Score equal to the difference in the number of moves available to the
    - two players.

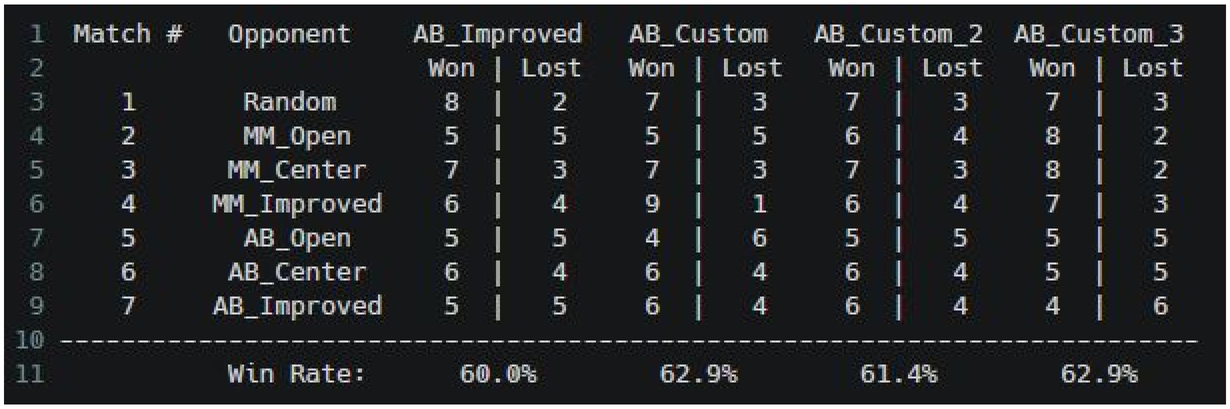
Competing Agents

The custom heuristics are tested against the following agents:

Name	Search Algorithm	Heuristic
Random	None, chooses a move at random	None
MM_Open	Fixed-depth Minimax	open_move_score
MM_Center	Fixed-depth Minimax	center_score
MM_Improved	Fixed-depth Minimax	improved_score
AB_Open	Iterative-deepening Alpha Beta	open_move_score
AB_Center	Iterative-deepening Alpha Beta	center_score
AB_Improved	Iterative-deepening Alpha Beta	improved_score

Example Test

Each test run has the following format:



```
 Match #    Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                         Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1        Random       8  |  2      7  |  3      7  |  3      7  |  3
    2        MM_Open      5  |  5      5  |  5      6  |  4      8  |  2
    3       MM_Center     7  |  3      7  |  3      7  |  3      8  |  2
    4      MM_Improved    6  |  4      9  |  1      6  |  4      7  |  3
    5        AB_Open      5  |  5      4  |  6      5  |  5      5  |  5
    6       AB_Center     6  |  4      6  |  4      6  |  4      5  |  5
    7      AB_Improved    5  |  5      6  |  4      6  |  4      4  |  6
--------------------------------------------------------------------------
           Win Rate:      60.0%        62.9%        61.4%        62.9%
```

Where **AB_Custom, AB_Custom_2** and **AB_Custom_3** are agents using Iterative deepening Alpha Beta search with a different custom heuristic.

Results

The custom heuristics are aiming to perform better than the **AB_Improved** agent, as it is the highest performing of the given agents and uses the heuristic described at the beginning. The test tournament was run with **50 matches against each opponent** to improve the reliability of the results.

Agent	Performance
AB_Improved	**65.88%**
AB_avoid_corners	66.37%
AB_chase_opponent	60.7%
AB_weighted_chase_opponent	62.9%
AB_next_moves	**71.2%**

Analysis

The **Next Moves** heuristic should be used:

- Most effective of the custom heuristics and achieves the goal of beating the **AB_Improved** agent.
- Takes into account the possible moves created by the **current move being evaluated**, it is less likely to be affected by the **horizon effect** - picking a move which is detrimental later on.
- Increases the player's possible moves, whilst decreasing the opponent's possible moves (due to the rewards/penalties applied for each)