Heuristics Analysis

For this Isolation game, we examined six different heuristic functions. Every heuristic is using difference in number of moves as the base evaluation value, with weights 0.4 and 0.6 on current player's number of moves and opponent's number of moves respectively. The weights are chosen to make the heuristic slightly aggressive by putting more weights on opponent moves.

Heuristic 1

In addition to the difference in number of moves, we use the difference in their distance to center of board as another measurement for the probability of winning. The closer a player is to the center, the more space it has to move around. This measurement is used when the difference in weighted moves is zero. It is first divided by 10 since any advantage in position should have less effect than a lead in number of available moves. Then it is divided by half to keep it consistent with the weighted number of moves.

Implementation

Below is the implementation of this heuristic function:

```
def custom_score(game, 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)))
      if not 0.4 * my_moves == 0.6 * opp_moves:
          return float(0.4 * my_moves - 0.6 * opp_moves)

      # Calculate the Manhattan distance to the center of the board
      # The closer a player is to the center, the larger probability that the
player would win
      center_y, center_x = int(game.height / 2), int(game.width / 2)
      my_y, my_x = game.get_player_location(player)
      opp_y, opp_x = game.get_player_location(game.get_opponent(player))
      my_dist = abs(my_y - center_y) + abs(my_x - center_x)
      opp_dist = abs(opp_y - center_y) + abs(opp_x - center_x)
      # Divide the distance by 10 since positional advantage is less effective
      # than the difference in number of moves left
      return float(opp_dist - my_dist)/10 * (1/2)
```

Heuristic 2

Instead of using the distance to the center of board, we count the total number of blank spaces that are within the range of a 5x5 patch around the current player position. The more spaces there is around the player, the longer the player is likely to survive. As before, to keep it consistent with the number of moves function, it is divided by 60.

Implementation

```
def custom_score_2(game, 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)))
  if not 0.4*my_moves == 0.6*opp_moves:
      return float(0.4*my_moves - 0.6*opp_moves)
  #Use number of blank spaces in the surrounding as a measurement
  #The larger the number of blank spaces around, the greater the advantage
  my_space = _num_of_nearby_spaces(game, player)
  opp_space = _num_of_nearby_spaces(game, game.get_opponent(player))
  return float(my_space - opp_space)/30 * (1/2)


def _num_of_nearby_spaces(game, player):
    spaces = game.get_blank_spaces()
    player_y, player_x = game.get_player_location(player)
    score = 0
    for space in spaces:
        dist_y = abs(player_y - space[0])
        dist_x = abs(player_x - space[1])
        if dist_y < 3 and dist_x < 3:
            score += 1
    return score
```

Heuristic 3

We incorporates both distance to center and the number of nearby spaces into heuristic 3.

Implementation

```
def custom_score_3(game, 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)))
  if not 0.4*my_moves == 0.6*opp_moves:
      return float(0.4*my_moves - 0.6*opp_moves)
  # A mixture of positional advantage and spacial advantage
  c_y, c_x = int(game.height / 2), int(game.width / 2)
```

```
  my_y, my_x = game.get_player_location(player)
  opp_y, opp_x = game.get_player_location(game.get_opponent(player))
  my_dist = abs(my_y - c_y) + abs(my_x - c_x)
  opp_dist = abs(opp_y - c_y) + abs(opp_x - c_x)

  my_space = _num_of_nearby_spaces(game, player)
  opp_space = _num_of_nearby_spaces(game, game.get_opponent(player))
  return 1/2 * (float(my_space - opp_space)/30 + float(opp_dist - my_dist)/10)
```

Results and Analysis

We changed the NUM_MATCHES in tournament.py to 15 so that each game agent runs 15*7 games against other agents. The larger number of runs give us a more realistic performance evaluation. We could see that all customer heuristics outperformed the original AB_Improved function which only counts the difference in number of available moves. The additional evaluation functions make the heuristic score more accurate by adding more information when the weighted number of moves can not tell whether a move is winning or losing. Heuristic 3 has the most information, therefore gives the best performance.

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

 Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random      29  |   1    28  |   2    29  |   1    28  |   2
    2       MM_Open     24  |   6    26  |   4    23  |   7    26  |   4
    3      MM_Center    27  |   3    25  |   5    26  |   4    27  |   3
    4     MM_Improved   18  |  12    21  |   9    22  |   8    24  |   6
    5       AB_Open     16  |  14    15  |  15    20  |  10    18  |  12
    6      AB_Center    17  |  13    20  |  10    17  |  13    18  |  12
    7     AB_Improved   13  |  17    16  |  14    15  |  15    15  |  15
--------------------------------------------------------------------------
          Win Rate:      68.6%        71.9%        72.4%        74.3%
```

Heuristic 4, 5, and 6

We use the longest path current player can take as the heuristic score when total number of spaces on board is less than 15. This is applied on all previous customer heuristics, giving us heuristic 4,5 and 6.

Implementation for Heuristic 4

```
def custom_score(game, player):
  if game.is_loser(player):
      return float("-inf")

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

```
  # Using the longest path as evaluation when total number of blank spaces on board
is less than 15
  blank_spaces = game.get_blank_spaces()
  if len(blank_spaces) < 15:
      my_loc = game.get_player_location(player)
      opp_loc = game.get_player_location(game.get_opponent(player))
      return 1/2*(float(_longest_path(game, blank_spaces, my_loc)) - float(
          _longest_path(game, blank_spaces, opp_loc)))

  my_moves = len(game.get_legal_moves(player))
  opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
  if not 0.4 * my_moves == 0.6 * opp_moves:
      return float(0.4 * my_moves - 0.6 * opp_moves)
  # Calculate the Manhattan distance to the center of the board
  # The closer a player is to the center, the larger probability that the player
would win
  center_y, center_x = int(game.height / 2), int(game.width / 2)
  my_y, my_x = game.get_player_location(player)
  opp_y, opp_x = game.get_player_location(game.get_opponent(player))
  my_dist = abs(my_y - center_y) + abs(my_x - center_x)
  opp_dist = abs(opp_y - center_y) + abs(opp_x - center_x)
  # Divide the distance by 10 since positional advantage is less effective
  # than the difference in number of moves left
  return float(opp_dist - my_dist)/10 * (1/2)
```

```
def _longest_path(game, blank_spaces, loc):
    directions = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),
                  (1, -2), (1, 2), (2, -1), (2, 1)]
    r, c = loc
    if blank_spaces == None or blank_spaces == []:
        return 0
    valid_moves = [(r + dr, c + dc) for dr, dc in directions
                   if (r + dr, c + dc) in blank_spaces]
    if len(valid_moves) == 0 or valid_moves == [None] or valid_moves == None or
valid_moves[0] == None:
        return 0

    blank_spaces_left_group = []
    for i in range(len(valid_moves)):
        blank_spaces_copy = blank_spaces
        new_blanks = blank_spaces_copy.remove(valid_moves[i])
        blank_spaces_left_group.append(new_blanks)

    if len(valid_moves) == 1:
        return _longest_path(game, blank_spaces_left_group, valid_moves[0]) + 1
    return max([_longest_path(game, blank_spaces_left, new_loc)
                for blank_spaces_left, new_loc in zip(blank_spaces_left_group,
valid_moves)])+1
```

Heuristic 5 and 6 are transformed from Heuristic 2 and 3 in similar way.

Results and Analysis

Heuristic 4, 5, and 6 are AB_Custom, AB_Custom_2, and AB_Custom_3 respectively. As before, the third heuristic (Heuristic 6) gives the best performance since it has the most information. However, these are worse than previous Heuristic 1, 2, 3. The reason for this must be that the longest path function takes too long to compute, which adversely affects the depth that iterative deepening could search into.

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

 Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
           Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random      30  |   0     29  |   1    30  |   0    29  |   1
    2       MM_Open     23  |   7     23  |   7    21  |   9    23  |   7
    3      MM_Center    25  |   5     27  |   3    30  |   0    29  |   1
    4     MM_Improved   18  |  12     21  |   9    21  |   9    25  |   5
    5       AB_Open     17  |  13     16  |  14    14  |  16    15  |  15
    6      AB_Center    17  |  13     14  |  16    16  |  14    15  |  15
    7     AB_Improved   11  |  19     14  |  16    13  |  17    17  |  13
--------------------------------------------------------------------------
           Win Rate:      67.1%        68.6%        69.0%        72.9%
```

Conclusion

After experimenting 6 different heuristics, we found that Heuristic 3 gives the best performance. It outperformed the simple difference-in-moves heuristic by combining the distance to center and the number of nearby blanks together within a reasonable computation time.