# Isolation Agent Heuristic Analysis

## Overview

The *AlphaBetaPlayer* isolation agent was played against 7 different adversarial agents, all with varying levels of sophistication ranging from a random agent, to an *AlphaBetaPlayer* using the `improved_score` heuristic (explained below).

The *AlphaBetaPlayer* using the `improved_score` is considered the state of the art for this experiment, with the goal of developing a heuristic function that would improve on the state of the art.

The tournament was run for three custom heuristic functions described below, and the performance was compared to that of the `improved_score`, named "AB_Improved". Each adversarial agent was played 5 times and the average
score over the 5 matches was used to evaluate the performance.

The opponent naming convention in the results presented is the following: `ID_Heurstic`, where *ID* refers to the *type* of agent and *Heuristic* refers to the function used to evaluate each move.

The types of agents are

- Random: an agent making a random move at each step
- MiniMaxPlayer (MM): An agent which implements the MiniMax game tree search algorithm
- AlphaBetaPlayer (AB): An agent which improves on MM by including Alpha-Beta pruning to the it's game tree search algorithm.

Throughout this document, the term *player* will be loosely used to represent the player we want to win the game, and *opponent* refers to the player *opponent* is trying to beat.

## Heuristic Analysis

## Results:

```
Match #  Opponent   AB_Improved  CanvasCount   Manhattan   AvoidEdges
                    Won | Lost   Won | Lost   Won | Lost   Won | Lost
```

```
1       Random       9  |   1     8  |   2     9  |   1     9  |   1
2       MM_Open      7  |   3     5  |   5     9  |   1     8  |   2
3      MM_Center     9  |   1     8  |   2    10  |   0    10  |   0
4     MM_Improved    9  |   1     7  |   3     7  |   3     7  |   3
5       AB_Open      5  |   5     5  |   5     8  |   2     8  |   2
6      AB_Center     7  |   3     5  |   5     7  |   3     6  |   4
7     AB_Improved    6  |   4     5  |   5     5  |   5     3  |   7
-----------------------------------------------------------------------
        Win Rate:      74.3%        61.4%        78.6%        72.9%
```

## Improved Score

This heuristic was provided to the students through the lecture materials. The heuristic maximizes the number of available moves to the *player*, while minimizing the number of available moves to the *opponent*.

## Custom Heuristic 1: Open Canvas Score

This heuristic evaluates the player position by calculating the number of open spaces in a 2-block wide *canvas* around the player. Below, the player position is labeled as `p` and the canvas is marked with x's.

The idea behind this heuristic is simply to optimize space around the player to increase the probability of having an available L-shaped move in the future.

```
      0   1   2   3   4   5   6
0  |   |   |   |   |   |   |   |
1  |   | x | x | x | x | x |   |
2  |   | x | x | x | x | x |   |
3  |   | x | x | p | x | x |   |
4  |   | x | x | x | x | x |   |
5  |   | x | x | x | x | x |   |
6  |   |   |   |   |   |   |   |
```

## Custom Heuristic 2: Manhattan Distance

## Custom Heuristic 3

# Code

Here are the python implementations of the heuristics described above.

## Improved Score

```
def improved_score(game, player):
    """
    This is the default evaluation function for the isolation agent

    The "Improved" evaluation function discussed in lecture that outputs a
    score equal to the difference in the number of moves available to the
    two players.
    """
    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)
```

## Open Canvas score

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

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

    '''
    Get the number of open squares around the players in a 2 square radius
    '''
    own_y, own_x = game.get_player_location(player)
    opp_y, opp_x = game.get_player_location(game.get_opponent(player))
    blank_spaces = game.get_blank_spaces()

    own_canvas_count = 0
    opp_canvas_count = 0
    for bx, by in blank_spaces:
        if (abs(bx - own_x) == player.n and abs(by - own_y) <= player.n) or (
                        abs(by - own_y) == player.n and abs(bx - own_x) <= player.n
            own_canvas_count += 1

        if (abs(bx - opp_x) == player.n and abs(by - opp_y) <= player.n) or (
                        abs(by - opp_y) == player.n and abs(bx - opp_x) <= player.n
            opp_canvas_count += 1

    return float(own_canvas_count - opp_canvas_count)
```