My game agent contains the following heuristics.

1. Manhattan or mirror

```
# Manhattan or mirror
# If player 1, pick center. If player two, mirror player 1 if possible
    num_blank_spaces = len(game.get_blank_spaces())
# If first play
if (num_blank_spaces == game.width*game.height):
  if game.get_player_location(player) == (game.height/2, game.width/2):
    return float('inf')

# If second player (assumes an odd, square board)
# Returns the mirrored position of the tuple location
def mirror(location):
   return tuple(x-y for x,y in zip((6,6),location))

if (num_blank_spaces % 2 == 0):
  opponent_location = game.get_player_location(game.get_opponent(player))
    if game.get_player_location(player) == mirror(opponent_location):
       return float('inf')

# If all else fails, uses open space with Manhattan distance
# Manhattan distance between two points
def m_distance(p1, p2):
  components = tuple((abs(a-b) for a,b in zip(p1,p2)))
    return sum(components)

# Sum of reciprocals of m_distance from player to each blank space
# The value of the sum is greater when more blanks are nearby
sum_of_distances = sum(
             [ 1/(m_distance(game.get_player_location(player), blank_space))
                    for blank_space in game.get_blank_spaces() ] )

return float(sum_of_distances)
```

This heuristic was my most consistent and successful. It uses a combination of heuristics to make a play based on player order in the game. Because of its higher consistency, I chose it as my best heuristic.

Trial	Custom performance	`AB_Improved` performance	Median number of wins against MM (in 10 games)	Median number of wins against AB (in 10 games)
1	71.4%	61.4%	8	5
2	70.0%	65.7%	9	5

2. Open space with Manhattan distance

```
# Open space with Manhattan distance
# Manhattan distance between two points
def m_distance(p1, p2):
  components = tuple((abs(a-b) for a,b in zip(p1,p2)))
  return sum(components)

# Sum of reciprocals of m_distance from player to each blank space
# The value of the sum is greater when more blanks are nearby
sum_of_distances = sum(
             [ 1/(m_distance(game.get_player_location(player), blank_space))
                          for blank_space in game.get_blank_spaces() ] )

return float(sum_of_distances)
```

This heuristic aims to maximize available spaces around the player by calculating the sum of the reciprocals of the Manhattan distances to all empty spaces. It performed well, but inconsistently, so I included it as a secondary custom score in my game agent.

Trial	Custom performance	`AB_Improved` performance	Median number of wins against MM (in 10 games)	Median number of wins against AB (in 10 games)
1	72.9%	67.1%	8	7
2	61.4%	60.0%	6	4

3. Defend/attack mobility

```
return float( len(game.get_legal_moves()) -
       2*len(game.get_legal_moves(game.inactive_player)) )
```

This heuristic places value on maintaining available moves and on limiting moves available to the opponent. I tried a variety of weights on the two values, but the results were similar. It failed to outperform `AB_Improved`.

Trial	Custom performance	`AB_Improved` performance	Median number of wins against MM (in 10 games)	Median number of wins against AB (in 10 games)
1	60.0%	60.0%	6	5
2				