AI Nanodegree Project 2: Build a Game-Playing Agent

Heuristic Analysis

In this document I describe and evaluate the 3 evaluation functions created for the isolation game agent.

custom_score

The evaluation function I settled with was decided as a result of (manually) observing the tournament results, with it regularly outperforming the others. The evaluation function is a set of evaluators of strategies defined (mostly) by trial and error.

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

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

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

   if len(own_moves) != 0 and len(opp_moves) == 0:
      return float("inf")
   elif len(own_moves) == 0 and len(opp_moves) != 0:
      return float("-inf")
   elif len(own_moves) == 0 and len(opp_moves) == 0:
      return -10
```

The first block is shared by all the evaluators and essentially assess if a game is won or tied; if won then obviously reward the agent otherwise return a negative score.

Each heuristic is given a weight to allow easy adjustment of the importance (neglected here for brevity).

The first heuristic is to incentivize the agent to stay closer to the middle; the logic here was being in the middle would increase the mobility of our player.

```
  euclidean_distance = math.sqrt( (center_pos[0] - current_pos[0])**2 + (center_pos[1] - current_pos[1])**2 )
  max_distance = math.sqrt( (center_pos[0])**2 + (center_pos[1])**2 )
  score += player_center_weight * (1.0 - euclidean_distance/max_distance)
```

Next is incentivizing moves were the agent has more moves than the opponent (mobility), inspired by the heuristic presented in the lectures but using a ratio rather than the absolute difference.

```
  score += weight_moves_difference_weight * float(len(own_moves))/float(len(opp_moves))
```

The last set of heuristics was to incentives the agent to force the opponent toward the edge or corner (reducing the opponent's mobility). Similar thinking to keeping the agent in the centre, being stuck at the edge would lead to fewer options.

```
  for opp_move in opp_moves:
      if is_in_corner(opp_move):
          opp_in_corner_count += 1
      elif is_on_edge(opp_move):
          opp_on_egde_count += 1

  score += opp_edge_weight * opp_on_egde_count
  score += opp_corner_weight * opp_in_corner_count
```

custom_score_2

This evaluation was an experiment with being offensive or defensive i.e. keeping as close as possible to the agent or moving far away. The final version was incentivising the agent being 3 positions away from the opponent (taking one of its positions - this is somewhat captured in a comparison between the legal moves remaining but puts higher emphasis in moving the agent towards the opponent ie a type of chasing).

The above heuristic is complementary to the absolute difference in legal moves remaining.

```
  score = float(len(own_moves) - len(opp_moves))
  distance = abs(opp_current_pos[0] - player_current_pos[0]) + abs(opp_current_pos[1] - player_current_pos[1])
  distance_from_target = 1.0 - abs(distance-target_distance)
  score += distance_from_target * 2.0
```

custom_score_3

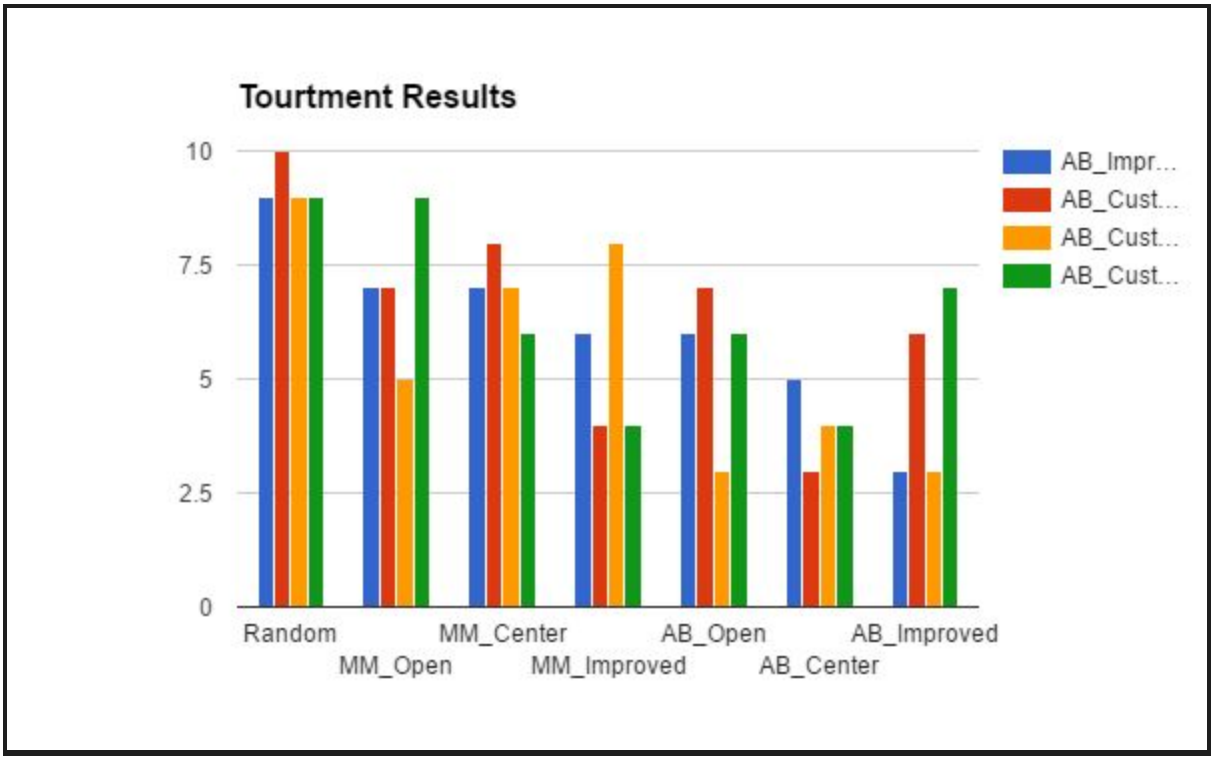
Similar to the evaluator presented in class but uses squares the difference to put more emphasis on larger differences between the player and opponent with the intention of having a more ‘aggressive’ agent.

```
  if len(own_moves) >= len(opp_moves):
      return (len(own_moves) / len(opp_moves)) ** 2.0
  else:
      return -(len(opp_moves) / len(own_moves)) ** 2.0
```

Analysis

The following figure shows the results of a tournament.

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	9	1
2	MM_Open	7	3	7	3	5	5	9	1
3	MM_Center	7	3	8	2	7	3	6	4
4	MM_Improved	6	4	4	6	8	2	4	6
5	AB_Open	6	4	7	3	3	7	6	4
6	AB_Center	5	5	3	7	4	6	4	6
7	AB_Improved	3	7	6	4	3	7	7	3
	Win rate	61.43%		64.29%		55.71%		64.29%	



In terms of complexity; each shared similar degrees of complexity, using O(1) functions with the cost being in the number of evaluations performed.

Overall, **custom_score** performed best compared to the other heuristics but not significantly. All shared a heuristic that assessed mobility which appeared to be the major contributing factor to the results. This, or any, heuristic would become less relevant as the possible depth became smaller as the contributing factor would be whether the agent lost or not.

Suggested improvements

Restricting the opponent's mobility or increase opportunity for mobility appears to be the strategy of the game (very curious to see what the other students came up with). Therefore having the previous move and state and looking ahead one move would allow for a more effective strategy but require *memory* and further *searching*, with the later increasing the compute time.
Another idea I was interested in was building a playbook using reinforcement learning but decided against this because it was unclear how to provide sufficient competition to ensure the agent would learn the best moves, maybe a crowd sourced version might be an option.