Udacity Artificial Intelligence Nanodegree

Advanced Game Playing

Author: Ke Zhang

Submission Date: 2017-06-24 (Revision 2)

Analysis of the Heuristic Evaluation Functions for Isolation Game

Heuristic Functions Analysis

In this analysis 4 heuristic functions are described and evaluated. In the first three functions, each of them implements a different strategy. And the last one tries to reuse the best strategy from the former three functions, but improves it further by optimizing the effectiveness of the alpha-beta pruning algorithm.

Variable and Method Descriptions

The following utility methods and variables are used in this document. Most of them are shared between the heuristic functions:

- Variables:
  - player: the current active player
  - opponent: the opponent player
  - own_score: score of the active player
  - opp_score: score of the opponent player
  - board_size: size of the board (default: 7*7=49)
- Methods:
  - is_winner(): check if the current player has won the game
  - is_loser(): check if the current player has lost the game
  - chase_opponent_moves(): increases the score of the opponent progressively
  - is_next_player(): next active bonus player gets a small bonus (e.g. to solve the parity case when both have the same score)
  - calculate_available_moves(): scoring function based on the number of available moves
  - calculate_central_tendency(): scoring function based on location of the player to the center
  - calculate_connected_area(): finding and calculating the number of connected free cells starting from the player location

Heuristic 1 (H1):

- Function Name: custom_score_2()
- Relevant scoring methods:
  - is_winner() and is_loser()
  - calculate_available_moves()
  - chase_opponent_moves()
- Strategy as Pseudocode:

```
own_score = calculate_available_moves(player)
opp_score = calculate_available_moves(opponent)
opp_score = chase_opponent_moves(opp_score, game.move_count)
return own_score - opp_score
```

- Description:

> The characteristic of this heuristic is that it's very simple and fast, so that it enables a deeper search compared to the other

> strategies. It uses only the number of available moves to calculate the scores and then increases the score of the opponent player scored by a progressive increasing factor to introduce a more aggressive play into the game.

Heuristic 2 (H2):

- Method Name: custom_score_3()
- Relevant scoring functions:
  - is_winner() and is_loser()
  - calculate_central_tendency()
  - calculate_available_moves()
  - chase_opponent_moves()
- Strategy as Pseudocode:

```
if opening:
   own_score = calculate_central_tendency(player)
   opp_score = calculate_central_tendency(opponent)
else:
   own_score = calculate_available_moves(player)
   opp_score = calculate_available_moves(opponent)
   opp_score = chase_opponent_moves(opp_score, game.move_count)
return own_score - opp_score
```

- Description:

> This heuristic uses different algorithms for different game stages. For the beginning of the game it assumes that the board is roughly empty and 'good' moves tend to be in the board center. The heuristic sums up the center score, which is calculated by the squared distance of the player position to the center and the number of moves available to provide an approximation of the goodness of the board at the opening game. In the main game, the heuristic applies the progressive opponent chasing strategy like in the previous heuristic. And again it tries to go fast and deep by counting the number of available moves of each player.

Heuristic 3 (H3):

- Method Name: custom_score()
- Relevant scoring functions:
  - is_winner() and is_loser()
  - calculate_central_tendency()
  - calculate_available_moves()
  - chase_opponent_moves()
  - adjust_next_player_advantage()
- Strategy as Pseudocode:

```
if opening:
   own_score = calculate_central_tendency(player)
   opp_score = calculate_central_tendency(opponent)
if midgame:
   own_score = calculate_available_moves(player)
   opp_score = calculate_available_moves(opponent)
   opp_score = chase_opponent_moves(opp_score, game.move_count)
if endgame:
   own_score = calculate_connected_area(player)
   opp_score = calculate_connected_area(opponent)
   own_score, _ = next_player_advantage(game, player, own_score, opp_score)
```

```
    adjust_next_player_advantage(player)
    return own_score - opp_score
```

- Description:

> This more advanced heuristic extends an additional logic to handle the end game and some parity cases when both players have the same score. In the end game, instead of counting available moves it chooses to calculate the connected area available around the player. That should be a better estimate because to the end game there would be less and less available moves (and so the maximum depth), so the time left can be used for more sophisticated algorithms like this one. And in the event there is a parity, the heuristic ensures that the next active player always wins in that case.

Competition Agent with improved alpha-beta algorithm (PvP)

- Class Name: CustomPlayer
- Description:

> The custom player implementation is used for the PvP tournaments. It uses the same strategy as the third heuristic function, since it showed a better results in the simulation games. The difference is that the custom player utilizes a more advanced alpha-beta pruning algorithm. It evaluates every child states and then reorders them before the next expansion. In the tournament results it reduces the amount of nodes drastically and improves the test result tremendously as you can see in the next chapter.

Comparison of the different heuristics

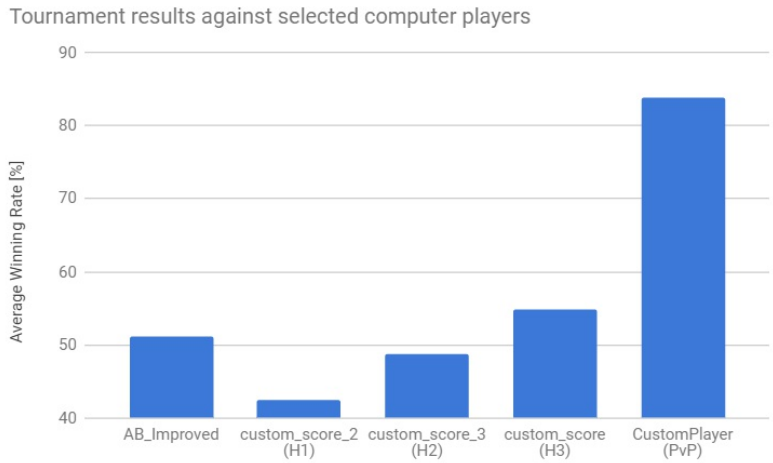
Performance Analysis

The best way to compare the performance of the different heuristics is to let them play against each other multiple time and then evaluate the game results.

Tournament Results:

#	Opponent	AB_Improved	custom_score (H3)	custom_score_2 (H1)	custom_score_3 (H2)	CustomPlayer (PvP)
1	Random	8-2	8-2	5-5	9-1	9-1
2	MM_Open	5-5	7-3	3-7	6-4	8-2
3	MM_Center	8-2	6-4	7-3	3-7	9-1
4	MM_Improved	5-5	4-6	2-8	3-7	9-1
5	AB_Open	4-6	6-4	6-4	5-5	9-1
6	AB_Center	6-4	4-6	6-4	3-7	8-2
7	AB_Improved	4-6	7-3	5-5	9-1	10-0
8	CustomPlayer	1-9	2-8	0-10	1-9	5-5
		51.2%	55.0%	42.5%	48.8%	83.8%

The graph below shows the average winning rate of the different heuristic implementations:



We can conclude from the tournament results that the custom player among all heuristics shows an amazing average winning rate of approximately 84%, while other test players had rather comparable results of around 50% winning rate. Among the other three heuristics the third one (H3) performed best with the overall winning rate of 55%. The first two heuristics had even worse results than the standard improved alpha-beta player (AB_Improved). Another surprise is that the standard minimax computer players won even more often than the alpha-beta algorithms which theoretically should outperform them because of the pruning effect. It could be caused by the test environment on my small-dimensioned laptop.

Recommendation

Last but not least we can conclude that the alpha-beta algorithm using a sophisticated evaluation function and having enough time and space to prune has drastically improved the overall result and won against the AB_Imporved in 10 games with 10 to 0. The success of the CustomPlayer implementation is based on the smart cut of the game into three stages. During the first two stages it attaches great importance to a simpler but faster heuristic function to allow a deeper search in the game tree. And in the last stage it ensures a better estimate with a more complex heuristic that could more likey lead to a victory in the game. Therefore we clearly recommend the last custom player implementation for further PvP competitions.