Heuristics Overview

While exploring the new to me Isolation game, I tried various heuristics as evaluation functions. Since I am quite familiar with Chess and Computer Chess, although evaluation functions are important in Chess they are not so important as standalone functions. Their most important role is guiding search, initiating extensions or pruning

For evaluation the influence of the various evaluations functions I tested, I used the provided tournament.py I have to note that there is a serious problem with how the tournament works. It provides a total amount of time for the engine to play the game and does zero time management. This will result, in long games eventually to a loss by time by some engine, since there is zero time management code

Heuristics Reference

1. **Open move score**: Number of legal moves available for the player. This is also used in the tournament.py by AB_Open and MM_Open. It seems competitive and on par with the AB_Improved
2. **Mobility**: The difference between number of own legal moves - opponent legal moves. This is also used in the tournament.py by the AB_Improved agent Its a simple and effective heuristic and its hard to do something better with simple tricks
3. **Blocking**: This simple heuristic gives a higher score to positions where both players have squares they can both go. It counts the common squares and give that as a score for the position
4. **Distance**: This returns the distance between the two players in the board
5. **Combined_1**: A linear combination of (1), (2) and (3). Tried various combinations and there is no statistical evidence that any combination clearly gives some improvement
6. **Combined_2**: This heuristic gives (number of legal moves)^2 - number of opponent legal moves, when the board is more than half full (that means Its more important to have open moves when the board is full) and (number of legal moves) - (number of opponent legal move)^2 when the board is not , which means I care more about restricting the opponent in the early stages. This gives a slight improvement over AB_Improved and it can be more fine tuned
7. **Combined_3**: Expanding on the previous idea this returns:

```
  if moves_to_board >= 0.8:  # closer to endgame (most squares are occupied
      return float(len(own_moves) * 4 - len(opponent_moves))
  elif 0.4 <= moves_to_board < 0.8:
      return float(len(own_moves) * 2 - len(opponent_moves))
  else:
      return float(len(own_moves) - len(opponent_moves) * 2)
 which does not change much compare to the previous heuristic
```

8. **Combined_4**: This was just a crazy idea as a shot in the dark. In short it calculates the following:

```
 score = float(own_moves ** 2 / (1 + opp_moves)) + float(own_moves / (1 +
opp_moves ** 2))
```

You can see the results of those heuristics in the following tournament tables.

Tournament Tables for some of the heuristics used

table 1:

Match	Opponent	AB_Improved		Combined_1		Combined_2		Combined_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	37	3	36	4	36	4	35	5
2	MM_Open	29	11	34	6	35	5	31	9
3	MM_Center	36	4	31	9	34	6	36	4
4	MM_Improved	31	9	29	11	30	10	26	14
5	AB_Open	19	21	19	21	16	24	22	18
6	AB_Center	29	11	23	17	23	17	24	16
7	AB_Improved	20	20	22	18	17	23	15	25
	Win Rate	71.8%		69.3%		68.2%		67.5%	

table 2:

Match	Opponent	AB_Improved		Combined_1		Combined_2		Combined_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	AB_Improved	199	201	206	194	201	199	201	199
	Win Rate	49.8%		51.5%		50.2%		50.2%	

table 3:

Match	Opponent	AB_Improved		Distance		Combined_4		Common Moves	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	37	3	36	4	39	1	33	7
2	MM_Open	29	11	28	12	32	8	27	13
3	MM_Center	36	4	34	6	36	4	32	8

4	MM_Improved	27	13	29	11	26	14	22	18
5	AB_Open	23	17	22	18	17	23	15	25
6	AB_Center	20	20	20	20	19	21	20	20
7	AB_Improved	16	24	21	19	20	20	18	18
	Win Rate	67.1%		67.9%		67.5%		59.8%	

Summary

First of all the statistical sample is definitely not significant to draw conclusions about the effectiveness of various heuristics What can be observed is that search is more important than heuristics. We can see that just changing the search method from classical minimax to alpha beta pruning , using the same evaluation function the agent works a lot better.

A big role plays how computationally "heavy" is the evaluation function. A computationally heavy function takes more time to compute and thus explores fewer nodes per sec. Efficient search is more important than fine tuned evaluation function