Some comments on the efforts taken to improve the Isolation game AI

Adverserial search:

Both minimax and alphabeta players were implemented furthermore additional effort was made to improve the performance of alphabeta search (see AlphabetaPlayer1. Also note that it does not pass the interface test due to modifications required to isolation.py) by ordering the moves based on the results of the shallower iteration deepening searches. I used two strategies:

a) ordering all the moves based on the results of the evaluation of previous level during iteration deepening

b) only making sure that the best move from the previous evalutaion is considered first

Aproach a) was 55-70% faster on average, while approach b) 65-80% faster than the base case

I feel that a major reason for the slower performance of the full ordering is due to pruning in alpha-beta search which does not guarantee that the ordering of the nonoptimal nodes is corrrect. Also the estimates of moves utility varied quite a bit when going from shallower to deeper search, all the way to depth of 10, meaning that the move ordering was far from perfect

In addition to move ordering I also saved the result of the previous searches in a map structure this allowed me to reuse some of the results of the previous searches during Iteration Deepening Search. For example, let's assume that the search at turn 1 was able to go to level 9 depth before timeout. Than at the next level the results of up to level 7 are available right away and we can start the iterative deepening search from level 7. This simple caching technique allowed to gain anouther 20% in speed up.

Finally I made some modifications in isolation.py

The first modification was to write a small function reverse_move(previos_loc, current_loc) that allowed to reverse the effect of a given move, given that the location at which the player stood on a previous move was saved. This allowed to use apply_move and reverse move when going down the alphabeta search instead of copying the entire position, winning both speed and memory. This modification sped up the search ~10 percent

Second modification was to write a series of function to check for different type of symmetries in a position and is described in more details in the appendix

Heuristic Analysis:

Improving heuristic function

I spend quite a lot of time trying to find a way to improve the heuristic functions used for evaluating the position value. To do it I ran millions of test where the different version of heuristic functions played against each other. However I had a hard time finding any practical functions that would improve the performance of simple mobility measures (number of own moves available or improved_score). The overall conclusion was that mobility score was a surprizingly effective and fast position evaluation score. It turned out that it was way better to use this simple score , but go down deeper down the evaluation tree, than use more advanced heuristics that took more time to evaluate. Going 2 additional plys down improved winning percetages 30-60 percent with advantage slowly levelling off.

Here is a short summary of different approaches I tried of improving the heuristics:

The first approch that I tried was incrementally varying coeffiecient c in the expression N_own_moves - c*N_opponent moves, from .1 to 5, but surprizingly it did not seem to affect the test results in any significant way.

The second approach I tried was to refine the estimates of number of moves available after a few moves from a given position, however these evaluation functions were more difficult to evaluate and had very modest effect on accuracy of the position value estimates, and in the end the additional computational effort required for more advanced functions did not warrant its use. It seemed that it was much better to keep the evaluation function very simple, but evaluate more positions at deeper levels. I also tried using the mobility score, but slightly biasing the move choices towards center or edges, however both approaches had negative effects

In the end I settled for the following set of heuristic functions that seemed to have a slim advantage over the improved score heuristic

Converted mobility values

The last approched that I tried was to convert the raw mobility scores used in improved_heuristic to some modified version of the score:

a) linear_mobility_conversion

converts simple mobility score in the following manner

f(1) = 1, f(2) = 1+1/2, f(7) = 1+ 1/2+...+1/7, where argument of f is number of moves available

now the improved heurstics is f(own_moves) - f (opp_moves). See the 3rd spreadsheet for the full table

b) quadratic_mobility_conversion

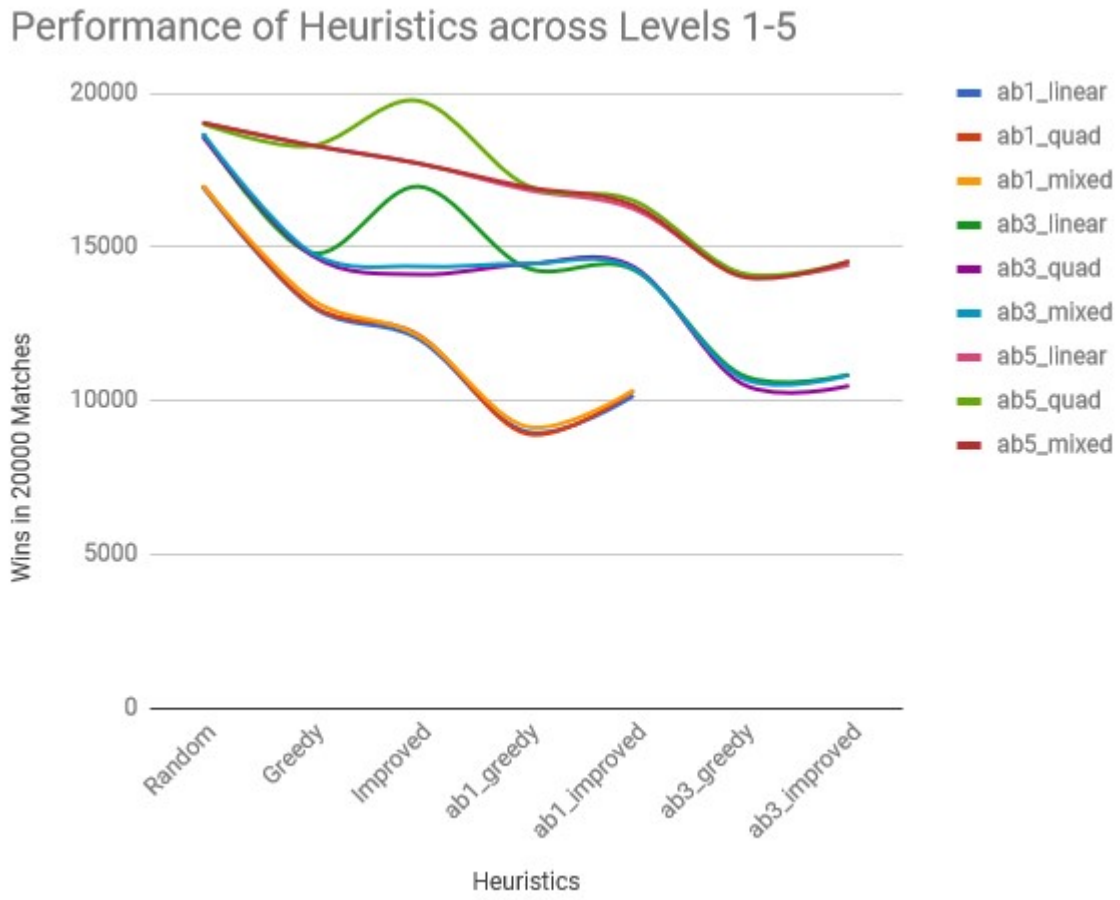
converts simple mobility score in the following manner

f(1) = 1, f(2) = 1+ 1/4, f(3) = 1 +1/4 +1/9,etc See the spreadsheet for the full table

c) mixed_mobility_conversion
f_mixed(own_moves, opp_moves) =
= (f_linear(own_moves) - f_linear(opp_moves))* ((f_quadratic(own_moves)- f_quadratic(opp_moves))
The tables are also available in the attached spreadsheets
To speed up the caluculations a table giving results of all possible scores is calculated beforehand. The new linear, quuadratic, and mixed_mobility conversion scores give a modest iprovement of 1-3% over the original improved_score evaluation function. The idea is that these new functions distinuguish between the situations of 8 moves against 7 and 2 moves against 1 giving bigger advantage to the situation of 2 moves against one. In general one can't say that this strategy always work, yet it seems to give some slight advantage on average, without much additional computational effort.

Heuristic Performance Results

The tables with all the tournament results are available in the first part of the attached spreadsheets. The summary and 4 charts are also given in the Summary part of the spreadsheet. The most interesting results can be seen in chart 4.



ab1_linear - means alpha-beta player that only goes to depth of 1 and uses linear_score_conversion heuristic.
It shows performance of several alpha-beta players performing in tournaments. The results from 9 different agents result in just 3 narrow bands. There is very little difference between performance of different agents (linear, quad, mixed) when using the same depth of search, but when the depth level of the search changes the performance of the three agents changes dramatically.

Charts 1-3 (see spreadsheet) also show that at different level of search the relative strength of 5 heuristics (open_score, improved_score, linear_score, quad_score, mixed_score) changes. The only consisten result is that open_score heuristic is clearly inferior to the other 4. Yet, when going to level 3 of the search the best heuristics are

linear, and mixed winning 51.6 and 51.4 percent of the matches with improved_score heuristics winning only 47.3 percent. When using depth of search 5, the best heuristic becomes improved_score winning 51.5 percent, mixed_score winning 50.6 and linear_score 50.4 percent. At level 7 the performance of improved_score drops to 48.8 percent, and linear and mixed win 52.2 and 52.1 percent respectively. Plese note that despite the relatively small differences in performance the difference is consistent and significant for each level of search. The variance for 20000 wins is around 70 wins or .35%

Final recommendation

The best heuristic seems to be mixed conversion score, it gives a slight advantage over the improved_score heuristic. Mixed_score heuristics requires more computation for getting its value than the improved_score heuristic, but since the values are only calculated once and then cached the difference in performance for value calculation is insignificant. There is another source of slower performance for the mixed_value heuristics. Alpha-beta algorithm uses pruning and it is easy to show that the amount of pruning for improved_score >= amount of pruning for mixed score. However, I could not observe that the mixed score would slow the search down compared to improved score. Both algorithms can go to level 9-11 during search in the beginning of games.

Finally below are some considerations on the symmetry. Inside the isolation.py I implemented isSymmetrical function that relatively efficiently checks for several types of symmetry: symmetry about the center point, horizontal and vertical center lines, and diagonals in boards that have equal height and width.

Symmetry

Determining whether there is a symmetry in the position is extremely important, because it is easy to show that
if one of the players was able to take the position symmetrical to it's opponent, the game can be won following
a very simple strategy. For example on any board of size nxm where both n and m are even player 2 will always
win by responding with a move that reflects the position of player 1. Say the board size is 4 by 4. Then in
response to any move by player 1, player 2 can take the position that reflects player1 move with respect to the
center. In response to move b2 by player 1 player 2 goes to c3. In this position player 1 has 4 moves:
a4, c4, d3, d1.
Player 2 job is to maintain the symmetry. Response to a4 : d1, to c4 : b1, to d3: a2, to d1 : a4. Each responce
maintains the symmetry around the center. It is guaranteed that player 1 would run out of moves first.
The center symmetry strategy works for player 2 on any boards if both dimensions are even, or one dimension is
even and another is odd. Furthermore it works as a special case for the board of sizes 3x3, nx1 and 1xn.
If the size of the board is nxm where both n and m are odd, there exist one square in the center of the board that
the player 2 can't reflect. However if the center square was previously occupied and the position is symmetrical
then player 1 can use the strategy to win the game.
There are 3 more symmetry strategies that exist:
1) symmetry about horizontal center line
2) symmetry about vertical center line
3) If the board is of size nxn symmetry around diagonals

symmetry about horizontal center line:
It works similar to symmetry around center. When the width of the board is even then the center line divides
the board along the line that goes between the two center fields. Given the same example of 4 by 4 board and
player 1 moving to b2 the new correspondense square for player 2 is c2 instead of c3, the response to a4: d4, to c4: b:4, to d3:a3, to d1:a1, and again player2 wins in the end by maintaining

symmetry each turn.
When the board width is an odd number the center line goes through the center column. To create conditions for
maintaining symmetry all the squares on the center line must be umavailable. Player 1 wins if he can achieve
symmetry after all the squares on the center line are filled or unreachable

Symmetry about vertical line:
See notes about symmetry about horizontal line

Symmetry about the diagonal.
The symmetry about the diagonal works in manner similar to symmetry around center lines, but occurs only
when the board is square. The symmetry can only be achieved if all the squares on the diagonal in question are
unreachable. If the board height is odd the symmetry benefits player1, if the height is even player 2 benefit

Symmetry in 7x7 board
While the symmetries around the center and vertical lines as well as major diagonals were discussed above, the positions necessary for these symmetries to occure are not achieavable in practice for 7x7 board.
The only symmetry of practical significance is the one around the center.

*But even the symmetry around the center is quite rare. While running 10000 tournament tests with alpha-beta players going to the depth of 7 with one evaluating center symmetry and another one not the symmetry was only encountered 547 times in total. Given that there are ~ 6^7*15 position evaluated each game the effect of symmetry evaluation was rather small. Out of the 20000 total games played the player with the symmetry evaluation won 10003.*

Several special starting positions for 7x7 board (not included in the program)

1) d4
... b3 (or b5, c2, c6, e2,e6,f3,f5)
2) f5 or (f3,e6,e2,c6,c2,b5,b3)
and player 1 takes symmetrical position and wins

1) c2 (or any of the other squares reachable from d4)
... d4
2) any move from c2
... e6
and even though the player 2 is not in a truly symmetrical position, but rather lags by 1 move from true symmetry. The player 2 will be able to win by going to the square that is symmetrical to the one player 1 held on previous turn. It is easy to show that player 1 would not be able to block player 2 moves and is guaranteed to run out of moves first. Player 2 wins

1) a1 (or any of the corners)
... d4
2) b3 (c2 is not better)
... f5 (e6 for c2)

Now the position is not fully symmetrical as one of the corners is empty, while another one is filled, but player 1 will lose if he ever attempts stepping into assymetric corner, so player 2 proceeds to a win by taking symmetrical position with respect to player 1. Player 2 wins

I am attaching the results of tournament evaluations as a separate spreadsheet.