Adversarial Game Playing Agent for knights isolation

Isolation

Minimax agents and variations

Minimax is an algorithm for 'zero sum games', where at each step of the game, the agent tries to minimize the worst possible loss. In minimax, all the nodes of the tree is searched, thus it is not a very effective algorithm to use in practice. Instead, there are more advanced variations that limit how many nodes are searched, such as alpha beta pruning, principal variation search, MTDF, etc.

In this project, I added iterative deepening (ie, the agent progressive search to greater depths in order to make use of all the available time and not run into a time limit) and also experimented with the alpha-beta agent with transposition table, and also node sorting. I also implemented a version of the alpha beta agent in iterative deepening where instead of starting from -infinty to infinity for each search, the iterative deepening algorithm saves the values of alpha and beta to reduce the search window.

The transposition table stores the state and the best alpha/beta value it takes to reach that state. This is because in the alpha beta search, it doesn't matter so much how the state is reached, but rather what is the 'best' value for that state. By saving the 'best' values it has obtained in the search so far, the nodes agent will rerun a search on is much less and the agent should be more efficient.

Node sorting -- in certain experiments, I also order the nodes at each depth depending on the scoring at at each depth. Since alpha beta pruning picks the 'best' nodes in terms of the heuristic score, by node sorting, it should help in making a more efficient agent.

Heuristic function

I decided to implement a custom heuristic function based on the baseline `availabe_my_moves - available_opponent_moves` heuristic. I added in an extra param to minimize the manhattan distance for my agent while maximizing the manhattan distance to the adversary-- this is implemented by adding in the `manhattan_opponent - manhattan_own` to the base heuristic. Since the probability of having more available moves is greater the closer the player is to the board center, you would want to minimize your distance from it while maximising your opponent's.

In addition, I also expect the depth at which the cost function is evaluated will have an effect-- a higher depth means the further in the future the agent searches. In one experiment, I multipled my custom heuristic score by `x ^ depth` where x is varied. I expected that x would have a optimal value just below 1 (so deeper depths are less important) for the agent to win. Surprisingly, it is the other way round, with the agent winning more for x > 1. (and afterwards the win rates seems to more or less saturate)

Scoring	Agent	games	time limit	win rate
my_moves - opp_moves	alpha-beta	100	200ms	75%

Scoring	Agent	games	time limit	win rate
my_moves - opp_moves + manhattan_opponent - manhattan_own	alpha-beta	100	200ms	87%
(my_moves - opp_moves + manhattan_opponent - manhattan_own) * 5.0 ^ depth	alpha-beta	100	200ms	90%
(my_moves - opp_moves + manhattan_opponent - manhattan_own) * 2.5 ^ depth	alpha-beta	100	200ms	77%
(my_moves - opp_moves + manhattan_opponent - manhattan_own) * 0.85 ^ depth	alpha-beta	100	200ms	84%
(my_moves - opp_moves + manhattan_opponent - manhattan_own) * 0.5 ^ depth	alpha-beta	100	200ms	85%
(my_moves - opp_moves + manhattan_opponent - manhattan_own) * 0.25 ^ depth	alpha-beta	100	200ms	77%

Efficiency As a measure of how efficient the agent is at searching, I investigated the maximum depth at different play counts by the different variations of the alphabeta agent -- both with and without node ordering, and with and without the transposition table.

The time limit used is 200ms, 100 games for the below (ply count, naxdepths are slightly approximated, since in my experiments the maxdepth is shared by different ply counts):

Agent	ply_count	maxdepth	win rate
transposition table + node sorting	70	500	77%
transposition table	60	750	82%
node sorting	70	500	80%
no transposition table , no node sorting	70	500	79%

While there is no significant difference in win rates between the different schemes, the tranposition table definitely helped the agent in searching to deeper depths as it does not need to re-search a large number of nodes.

In addition, I also investigated the number of alpha/beta cutoffs using the combination of node sorting/transposition table The basic alpha beta alorithm is more efficient than just minimax since it prunes branches that are definitely not going to have a higher/less value than the current highest/lowest node in that depth. The more branch pruning that occur, the more efficient the search is. Over 500 games each, the results are :

Type	cutoffs
sorting + transposition table	55626
no sorting + transposition table	20905
no sorting, no transposition table	5108

We can see that the sorting and the transposition table defintely made a significant difference in the amount of pruning. The win rates of all three combinations are more or less the same-- around 80% over the basic minimax agent. We can see that the extra search efficiency did not come at a cost to the move quality.

Implementation

There is one full implementation of the agent with tranposition table + node sorting in my_custom_player.py Most of the other experiments around adding sorting etc are done in the sample_players.py file