Adversarial Game Playing Agent

Synopsis

In this project, we will experiment with adversarial search techniques by building an agent to play knights Isolation. Unlike the examples in lecture where the players control tokens that move like chess queens, this version of Isolation gives each agent control over a single token that moves in L-shaped movements--like a knight in chess.

Experiments

This report will focus on 2 experiments:

- Option 1: Develop a custom heuristic
- Option 3: Build an agent using advanced search techniques

Experiment – Option 1

First lets establish a baseline using the heuristic #my_moves - #opponent_moves. The solution developed is based on the Minimax algorithm using Alpha-Beta Pruning as an optimisation technique. During this experiment iterative deepening will be disabled and a fixed search depth of will be used. Table 1. Illustrates the baseline against an opponent using Minimax at a fixed search depth of 3. For completeness I have also added test run data against the Random and Greedy (Minimax, D=1) opponents. The matches were each playing 50 rounds with the fair flag enabled resulting in 200 games.

Player Heuristic	**Wins against Random**	**Wins against Greedy**	**Wins against Minimax (D=3)**
Baseline (#my_moves - #opponent_moves)	98.5%	81.5%	64.5%

Table 1. Results of running custom Alpha-Beta (fixed D=5) against the three opponents over 200 games each.

The evaluations functions chosen for this experiment are listed in Table 2. along the baseline for comparison. In the table *weight* is an integer number greater than or equal to 1, *game_progress* is a value between 0 and 1 and measures how many squares have been used: *state.ply_count/board_size*

Name	**Evaluations function**	**Comments**
Baseline	#my_moves - #opponent_moves	Maximize the number of available player moves minus the number of opponent moves
Weighted Attack	#my_moves - weight*#opponent_moves	Favors minimizing the opponent's available moves
Weighted Defend	weight*#my_moves - #opponent_moves	Favors maximizing my own available moves
Weighted Attack then Defend	game_progress <= x then "Weighted Attack" else "Weighted Defend"	game_progress = state.ply_count/board_size
Weighted Defend then Attack	game_progress <= x then "Weighted Defend" else "Weighted Attack"	
Avoid Borders	'Baseline' + (border_distance == 0 then -10, if border_distance == 1 then -5)	Where border_distance is the distance to the closest border

Table 2. Evaluation functions

The results of the evaluation functions are listed in Table 3. Again the opponent was Minimax (fixed D=3) and the matches were each playing 50 rounds with the fair flag enabled, resulting in 200 games.

Player Heuristic	**Weight**	**Game progress ratio**	**Nodes expanded**	**Total Time per game (sec)**	**Wins against Minimax (D=3)**
Baseline			12703	10.13	64.5%
Weighted Attack	2		13905	9.86	62.5%
Weighted Attack	3		14271	9.77	70.0%
Weighted Defend	2		13631	10.04	63.0%
Weighted Defend	3		13899	9.96	66.0%
Weighted Attack then Defend	2	0.5	13586	9.80	64.0%
Weighted Attack then Defend	2	0.3	13905	10.17	70.5%
Weighted Attack then Defend	3	0.5	14078	10.07	63.5%
Weighted Attack then Defend	3	0.3	14158	9.93	68.0%
Weighted Defend then Attack	2	0.5	13538	9.81	66.0%
Weighted Defend then Attack	2	0.3	13631	9.75	60.0%
Weighted Defend then Attack	3	0.5	13779	9.77	66.0%
Weighted Defend then Attack	3	0.3	13970	9.68	59%
Avoid Borders			12673	9.94	65%

Table 3. Results of running custom Alpha-Beta (fixed D=5) against Minimax opponent (fixed D=3) over 200 games each

Ignoring the significant uncertainty in reported win rates we conclude the following trends based on experiments performed:

- Attack preferred to Defend
- If combined, Attack first and then Defend
  - Attack first then Defend but don't wait to the very end
- If we initially Defend the Attack at the very end of the game

New custom heuristic: Aggressive Attack then Aggressive Defend:

```
game_progress <= x then
        #my_moves – weight*#opponent_moves*(1-game_progress)
else
        weight*#my_moves*(1-game_progress) - #opponent_moves
```

Player Heuristic	**Weight**	**Game progress ratio**	**Nodes expanded**	**Total Time per game (sec)**	**Wins against Minimax (D=3)**
Aggressive Attack Aggressive Defend	3	0.3	14430	10.06	72.5%

Table 4. Results of running custom Alpha-Beta (fixed D=5) against Minimax opponent (fixed D=3) over 200 games each

So the aggressive strategy layout with our latest custom heuristic seems to have paid of as the best result so far.

Q&A: Option1 - Advanced Heuristic

What features of the game does your heuristic incorporate, and why do you think those features matter in evaluating states during the search?

So the features covered are:

- Liberties for our agent
- Liberties for our opponent
- Game progress, that is, fields available to move into
- Borders

The purpose of the evaluation function is to try an estimate our strength at a given point in time. So the liberties for our agent gives us a good strong indicator if a move should be performed – the higher the better. Likewise the liberties of our opponent is a strong indicator – the less options our opponent have the better. Game progress is interesting because it given us options to dynamically change our behaviour, an example being: 'Aggressive Attack then Aggressive Defend'. Finally borders, borders are strongly correlated with liberties but can gives us a sense of direction, that is, direction out of future trouble (towards centre).

Regarding the 'Aggressive Attack then Aggressive Defend' heuristic result: It makes sense that as there are fewer moves that our agent can perform as the game progresses, we focus more on finding those available spaces than what the opponent does.

Analyze the search depth your agent achieves using your custom heuristic. Does search speed matter more or less than accuracy to the performance of your heuristic?

Before answering search depth let's turn iterative deepening on and run it against the Baseline and the Aggressive Attack heuristics with weight: 3 and game progress: 0.3 . The results are illustrated in Table 5.

Player Heuristic	**Depth interval**	**Plies (avg, min, max)**	**Nodes expanded (avg, min, max)**	**Algo exec time per game (sec)**	**Depths (avg, min max): (median, mode)**	**Wins against Minimax (D=3)**
Baseline*	Fixed 5	67,-,-	12703,-,-	0.51	5,-,- : -,-	64.5%
Agg. Attack Agg. Defend*	Fixed 5	68,-,-	14491,-,-	0.48	5,-,- : -,-	72.5%
Baseline	1-9	68, 49, 86	71702, 59546, 92364	2.39	7.48, 5, 9 : 7, 9	75.5%
Agg. Attack Agg. Defend*	1-9	68, 47, 82	69391, 51876, 86829	2.34	7.38, 4, 9 : 7, 9	82.5%
Baseline	1-20	67, 48, 85	89586, 58195, 123144	2.93	10.05, 5, 20 : 7, 6	82.0%
Agg. Attack Agg. Defend*	1-20	67, 44, 83	87384, 52895, 111657	2.89	9.48, 5, 20 : 7, 6	83.5%

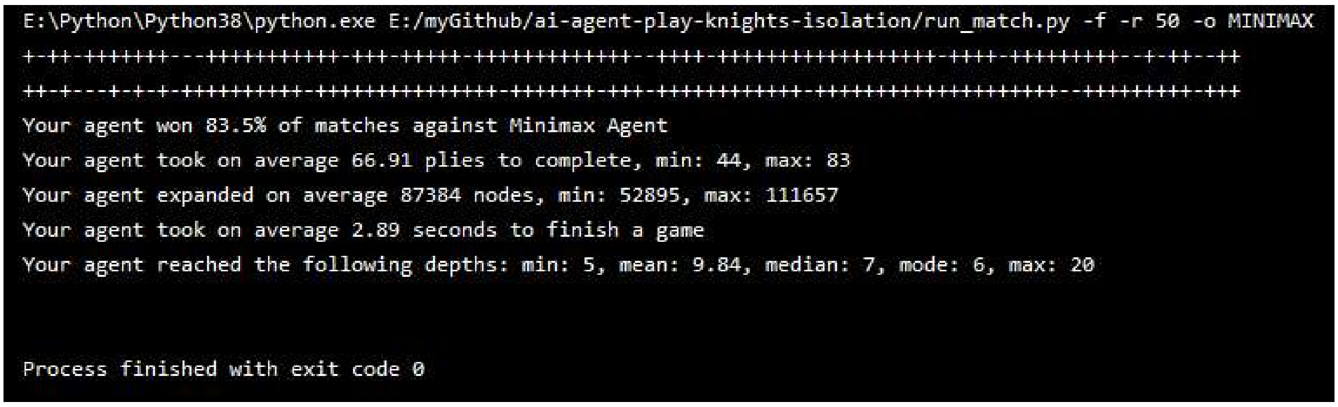
Table 5. Results of iterative deepening for selected heuristics.

*No iterative deepening, fixed depth 5, added for reference

Table 5. top rows illustrate that more accurate heuristics like the Aggressive Attack heuristic can result in big performance gains of up to 8% for our data (ignoring statistical uncertainty). On the other hand high search speed means small simple heuristics that can be executed fast resulting in deeper tree searches which is equivalent to more nodes expanded. This results in better data to make next move decisions upon. The importance of going deep is illustrated by iterative deepening where going from D=5 to D=20 results in performance improvements of 7.5% and 11% for Baseline and Aggressive Attack heuristics respectively.

The fact is they are both very important and should both be optimized.

Finally our best result (so far):



```
E:\Python\Python38\python.exe E:/myGithub/ai-agent-play-knights-isolation/run_match.py -f -r 50 -o MINIMAX
+-++-+++++++---+++++++++++-+++-+++++-+++++++++++++--++++-+++++++++++++++++-++++-+++++++++--+-++--++
++-+---+-+-+-+++++++++++-++++++++++++++-+++++++-+++-++++++++++++-+++++++++++++++++++--++++++++++-+++
Your agent won 83.5% of matches against Minimax Agent
Your agent took on average 66.91 plies to complete, min: 44, max: 83
Your agent expanded on average 87384 nodes, min: 52895, max: 111657
Your agent took on average 2.89 seconds to finish a game
Your agent reached the following depths: min: 5, mean: 9.84, median: 7, mode: 6, max: 20


Process finished with exit code 0
```

Experiment – Option 3

As an additional advanced search technique I have chosen Monte Carlo Tree Search. To achieve good balance between exploration and exploitation I am using the UCB1 selection function. When applied to MCTS, the combined algorithm is named UCT (Upper Confidence Bound 1 applied to trees), however going forwards I will refer to the whole solution as MCTS.

Table 6. demonstrates the performance of Alpha-Beta with iterative deepening using 'Agg. Attack Agg. Defend' heuristics compared to MCTS. The MCTS reported here uses an exploration parameter of 2 and a max roll-out moves loop of 90. 90 is picked as we never see max plies greater than mid 80ish. Both experiments are with an agent timeout of 150 milliseconds. The matches were each playing 50 rounds with the fair flag enabled resulting in 200 games.

Function	**Depth interval**	**Plies** (avg, min, max)	**Nodes expanded** (avg, min, max)	**Algo exec time per game** (sec)	**Depths** (avg, min max): (median, mode)	**Wins against Minimax** (D=3)
Alpha-Beta iter. deepening	1-100	68,47,83	100676,62765 ,128475	3.21	21.67,5,100 : 7,6	84.0%
MCTS exp_param=2 max_roll-out_ moves = 90		71,24,87	93358,32631, 112454	4.36	6.70,1,23 : 6,5	78.5%

Table 6. Results of Alpha-Beta with iterative deepening using 'Agg. Attack Agg. Defend' heuristics compared to MCTS. Agent timeout of 150 milliseconds

Finally in our last experiment we explore the impact of time on the performance of the algorithms. This experiment uses exactly the same settings as in previous experiment with one exception and that is an agent timeout of 500 milliseconds. Table 7. captures the results.

Function	**Depth interval**	**Plies** (avg, min, max)	**Nodes expanded** (avg, min, max)	**Algo exec time per game** (sec)	**Depths** (avg, min max): (median, mode)	**Wins against Minimax** (D=3)
Alpha-Beta iter. deepening	1-100	67,46,86	306048, 188312, 399477	10.14	25.85,6,100 : 9,7	83.0%
MCTS exp_param=2 max_roll-out_ moves = 90		71,53,87	314238, 231926, 380533	14.78	8.51,1,28 : 7,6	81.5%

Table 7. Results of Alpha-Beta with iterative deepening using 'Agg. Attack Agg. Defend' heuristics compared to MCTS. Agent timeout of 500 milliseconds

Q&A: Option3 - Advanced Search Techniques

Choose a baseline search algorithm for comparison (for example, alpha-beta search with iterative deepening, etc.). How much performance difference does your agent show compared to the baseline?

The results of the baseline versus out MCTS implementation are shown in Table 6 and 7. I have used the fair flag to mitigate differences caused by opening position. I have also gone deeper in the iterative deepening loop to achieve comparable executions times for the two solutions.

Why do you think the technique you chose was more (or less) effective than the baseline?

The experiments shows that Alpha-Beta with iterative deepening and custom heuristics are performing slightly better than this MCTS implementation for short agent timeouts. In general with Alpha-Beta using iterative deepening we don't see much improvement once a certain depth (D=9) or compute time has been reached. On the contrary MCTS benefits with more compute time and is on par with baseline when we reach about 500 ms as demonstrated in Table 7.

One possible explanation is that Minimax will always have good information for the next move, by full calculation or heuristics, down to a certain depth. However MCTS selects the node that has the best chance of winning and expand from there for a number of iterations but due to its random nature if the samples are too small it might miss the winning move. So increasing the sample size increases its performance and as sample size is directly related to time - the more time the better it performs.