Search assignment

Implementing search algorithms for the Hex game

Reinforcement Learning 2021

Deniz Sen

Universiteit Leiden

26 February 2021
Leiden

1 Introduction

In this paper we examine the potential of different search algorithms when playing a game of Hex. The objective being to create an intelligent computer program, which can evaluates moves and make the optimal selection among them. The first search algorithm we consider is alpha-beta pruning with a Dijkstra shortest path evaluation function. Here we evaluate the strength of the algorithm by letting it play against varying version of itself and an agent with a random decision policy. The specific set-up for these experiments can be seen in Table 2. Second, enhancements such as iterative deepening (ID) to turn regular alpha-beta into an any time algorithm and transposition (TT) tables to avoid evaluating a board configuration, which has already been examined are added as well. Again the strength of these improvements are evaluated by letting it play against other agents. The match-ups and settings for each agent are summarized in Table 3. Lastly, the potential of Monte Carlo Tree Search (MCTS) is examined. Here special attention is given to the trade off between exploitation and exploration by experimenting with a number of different hyper-parameter configurations. Details for the conducted experiments can be seen in Table 4.

As such, we examined the following topics in chronological order: alpha beta search and Dijkstra heuristic evaluation, iterative deepening and transposition tables as enhancements for alpha beta and lastly Monte Carlo Tree Search(MCTS).

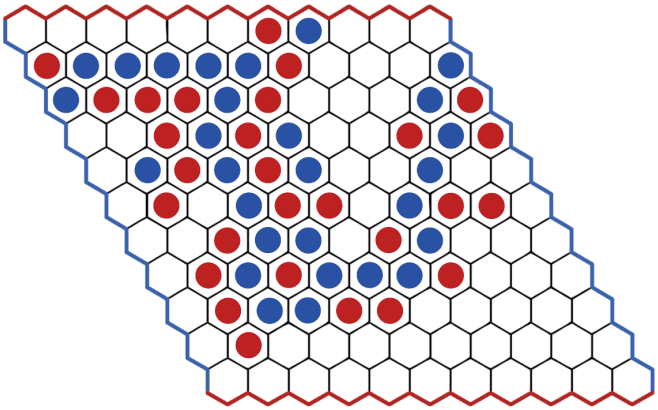


Figure 2: A regular rhombus hex board with red and blue players.

.

1.1 A Game of Hex

Hex is a two player strategy board game played on a diamond or rhombus shaped board made of hexagonal cells. Board size dimensions can vary, but the typical board size is 11×11 as shown in Figure 2. Two opposite sides of the board are labelled "red", and the remaining two sides are labelled "blue". One of the players has a supply of red tiles while the other player has a supply of blue tiles. The players alternate turns to place their tile on any unoccupied space on the board, with the goal of forming an unbroken chain of tiles (of their own colour linking two regions of the same colour). The Hex Theorem states that a game of Hex cannot end in a draw. The only way to keep the opponent from building a winning chain is to build a winning chain first. Therefore there is always a winner and it is a zero-sum game.

The complexity of Hex is not to be underestimated. Similar to Go a single cell can have $p = 3$ possible states; red, blue or unoccupied. A simple strategy to estimate the state space complexity $\log 10(p)$of a board is to consider all possible board configurations p^n:

$$\begin{aligned} \text{Let: } p^n &= a \times 10^b \text{ with } 1 \leq a < 10, b \in \mathbb{N} \\ 10^b &\leq p^n < 10^{b+1} \\ b &\leq n \times \log_{10}(p) < b + 1 \end{aligned} \qquad \text{(Method 1.)}$$

Using the ratio of $\frac{p^n}{n \log 10(p)}$, all possible board combinations can then be expressed as $\frac{p^n}{n \log 10(p)} \times 10^{n \log 10(p)}$. However, since each turn is regarded as an independent event unfeasible board combinations (i.e. entire board black or white) are included. These can be

excluded by considering that the number of spaces taken by either player can at most be greater than one compared to the number of spaces taken by the other player. At no given state, can a player have occupied more than one cell than its adversary. This consecutively results in a more accurate and smaller estimate of possible board combinations, even-though some infeasible position are included (i.e. board configurations where the game has already ended before it could emerge). Taking those positions into account makes the problem enormously more difficult to solve in closed form without brute-forcing it. Therefore, the number of possible board configuration is better described with Method 2.

Given that there are n cells on the board and each player has i tile pieces, there are then

$$\left(\begin{array}{c} n \\ i \end{array} \right) \left(\begin{array}{c} n-i \\ i \end{array} \right)$$

different board configurations. If one player has one more piece than the other, then there are

$$\left(\begin{array}{c} n \\ i \end{array} \right) \left(\begin{array}{c} n-i \\ i-1 \end{array} \right)$$

possible board configurations. Therefore Hex has a total number of

$$\sum_{i=0}^{n/2} \left(\begin{array}{c} n \\ i \end{array} \right) \left(\begin{array}{c} n-i \\ i \end{array} \right) + \sum_{i=1}^{n/2} \left(\begin{array}{c} n \\ i \end{array} \right) \left(\begin{array}{c} n-i \\ i-1 \end{array} \right)$$

possible board configurations, which using Pascals Rule can be reduced to

$$1 + \sum_{i=1}^{n/2} \left(\begin{array}{c} n \\ i \end{array} \right) \left(\begin{array}{c} n-i+1 \\ i \end{array} \right) \qquad \text{(Method 2.)}$$

Method 2 provides a feasible upper bound for the number of possible board combination in closed form. Since possible board combinations are expressed as a sum and as a function of n, as n increases so does the estimate of the upper bound. Possible board combinations are an increasing function of n. Computational cost therefore increases and is the reason why we will primarily focus on experiments using smaller board sizes. That is board size 3×3, 4×4 or 5×5 as shown in Table 1.

Board Game Setup

Board Size	State Space Complexity (as log10)	Board Combinations
3x3	4	1.96×10^4
4x4	7	4.3×10^7
5x5	11	8.47×10^{11}

Table 1: The table summarizes the state space complexity and the number of possible combinations as estimated by the second method above.

2 Methodology

2.1 Implementation

In a game of Hex, the game tree, where a parent node represents the boards current state and its children represent future board states can be illustrated as shown in Figure 3. Invoking all possible moves can then be regarded as the transition from the state given by the parent to the state given by the child.

As such the searcher looks through the children of the parent and evaluates each future state as a potential move to take. To define such an intelligent computer program a number of helper functions, an evaluation function and the searcher itself need to be created.

2.1.1 Helper functions

For helper functions we define a function *getMoveList()*, which collects possible moves, *makeMove()*, which makes a move on the board at a specified position and returns the resulting board state and *unMakeMove()*, which reverts a specified move and returns the previous board state.

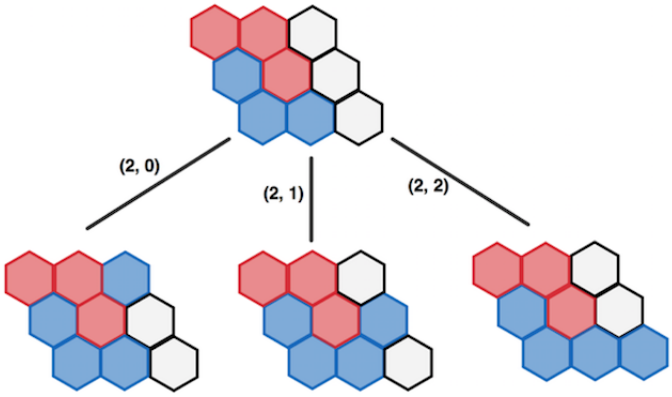


Figure 3: Game tree representation of a Game of Hex on a 3×3 board.

.

2.1.2 Evaluation function

Possible moves are evaluated using an adapted version of the Dijkstra shortest path algorithm, where we consider both agents and its respective opponent. That is, instead of having each player only evaluate its own shortest distance given a possible move, the adversary is also considered. This is achieved by evaluating the board state using the difference between the shortest paths of the agent to its adversary. As a result, the agent will search for moves that increase the length of the shortest path for its opponent and decrease the length of the shortest path for itself. We discarded the idea of using a random number generator for an alpha beta agent which utilizes a random decision policy since possible actions can be collected using our helper function *getMoveList()* from which we then make a random selection. This achieves the same purpose without having to initialize alpha beta itself and is more efficient.

2.1.3 Iterative deepening

Iterative deepening turns alpha beta into an anytime algorithm. A time budget is defined and given to the searcher, which allows the search function to iteratively examine deeper nodes to find a better move for each turn until the time budged is spend. If the time limit has been reached, the agent uses a previously saved action as a fail safe.

2.1.4 Transposition tables

Transposition tables allow the alpha beta searcher to check whether a certain board has already been evaluated previously. If a previous evaluation of the given board state already exists, then it does not need to be evaluated again. The evaluation function does not need to be called as the return values are stored in the transposition table for this call and are readily available for re-call. It avoids having to evaluating the same board arrangements multiple times. In that sense transposition tables are similar to memory or experience of a real player. Instead of making a new experience the agent can rely the same experience it made previously. We save depth as key and the best move and board state as values in a dictionary and define two functions *lookup()* and *save()*; *lookup()* looks for a transposition state and *save()* stores a board state.

2.1.5 Monte Carlo Tree Search

Monte Carlo Tree search (MCTS) tries to accumulate value estimates to guide an agents towards highly rewarding actions trajectories in a search tree. As such, MCTS pays more attention to nodes that are more promising and avoids searching through many nodes. MCTS combines three techniques: (1) adaptive exploration of the game tree, (2) sampling and (3) averaging for selection. As such unlike the alpha beta, which is a mini-max searcher, MCTS is not fixed-width fixed-depth search and does not use a heuristic evaluation but sampling for evaluation. Since Hex is similar to Go, where moves can have hidden long-term effects. In such games version of mini-max like alpha-beta, which are width-depth fixed, moves which have long-term strategic benefits for winning are not easily found. To define such a searcher, we define a function for the Upper Confidence Bound applied to

Trees Selection *ucts( )* to select a child. A move is then chosen in each node of the game tree for which the expression

$$\frac{w_i}{n_i} + C\sqrt{\frac{\ln N_i}{n_i}} \tag{1}$$

has the highest value. The index i denotes the i-th move, the parameter w_i stands for the number of wins a node; n_i stands for the number of simulations for a node, N_i stands for the total number of simulations and C is the exploration parameter often equal to $\sqrt{2}$. In the expression the first component of the formula above corresponds to exploitation, which means that it is high for moves with high average win ratio. The second component corresponds to exploration, which means it is high for moves with few simulations. Since the second term is a multiple of C, if C is equal to zero the agent will not explore and if C is large exploration occurs more in favor of nodes with less simulations or visits. To keep track of the number of wins w_i and the number of simulations for a node n_i a function which updates these node parameters *updateNodeParams()* is defined.

2.2 Experiments

In order to evaluate the regular alpha beta implementation, we defined a match-up, where we let the agent play against variants of itself (see Table 2). Number of alpha beta cut-offs, speed and Elo ranking are examined across 3 different board sizes (3×3. 4×4, 5×5).

For iterative deepening and transposition table, we re-run the same experiments as before with iterative deepening (time limit of 2s) and transposition tables enabled (see Table 3). The speed of iterative deepening with and without transposition tables is measured separately from these experiments. If the enhancements work well, transposition tables should decrease the speed of iterative deepening. When we refer to iterative deepening with transposition tables with depth 3 or 4, we mean the depth the agent was initialized not fixed at.

Lastly, an adaptive sampling method MCTS is examined. Here we compare MCTS's performance to the strongest searcher from the previous experiments (IDTT with depth 4). In addition, two experiment with the hyper-parameters C, which controls the exploration/-exploitation trade off and N the number of simulations are conducted. The aim being to explore, how low and high settings of each affect the agents performance against IDTT with depth 4. Specific details on the agent match-up can be seen in Table 4.

Alpha Beta Player Matchups

Match:	Player 1 policy:	Player 2 policy:
1	random	alpha beta depth = 3
2	random	alpha beta depth = 4
3	alpha beta depth = 3	alpha beta depth = 4

Table 2: Match-up table of the experiments run with regular alpha beta.

Enhancements Alpha Beta Player Matchups

Match:	Player 1 policy:	Player 2 policy:
1	random	IDTT depth = 3
2	random	IDTT depth = 4
3	IDTT depth = 3	IDTT depth = 4

Table 3: Match-up table of the experiments run with alpha beta enhancements.

MCTS Player Matchups

Match:	Player 1 policy:	Player 2 policy:
1	IDDT depth $d = 4$	MCTS $N = 2500$, C = $\sqrt{2}$
2	IDDT depth $d = 4$	MCTS $N = 1$, C = $\sqrt{2}$
3	IDDT depth $d = 4$	MCTS $N = 2500$, C = 1000

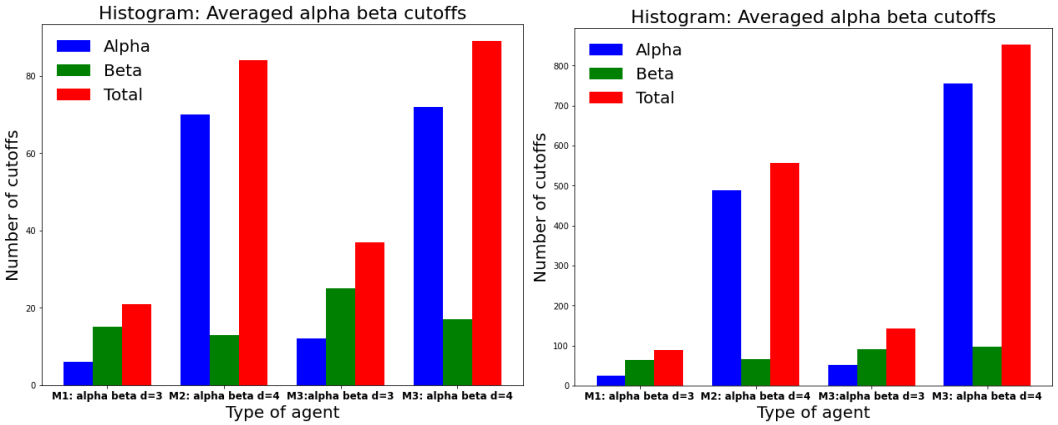
Table 4: Match-up table of the experiments run with monte carlo tree search.

3 Results

3.1 Alpha Beta

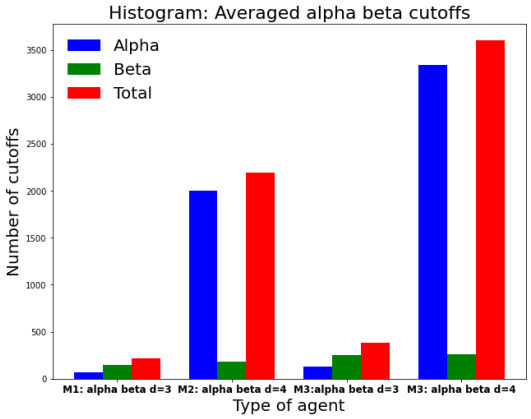
3.1.1 Cutoffs and Speed

The number of cut-offs for each matchup M are shown in Figure 4. From panel 4a to 4c we see that as board size increases so do the number of cut-offs. The same holds for depth. As depth increases, so do the searched number of nodes, resulting in more cut-offs. This is to be expected, since the deeper we travel down a tree the more branches occur, resulting in more cut-offs. The speed of the algorithm exhibits similar behaviour as shown in Figures 5a to 6a. As state space increases so does computation time. In comparison to depth 3 alpha beta at depth 4 has a higher computational load as it searchers through more nodes. However as the game progresses the effect of depth on computational load is mitigated, since the number of unoccupied positions decreases with each turn.



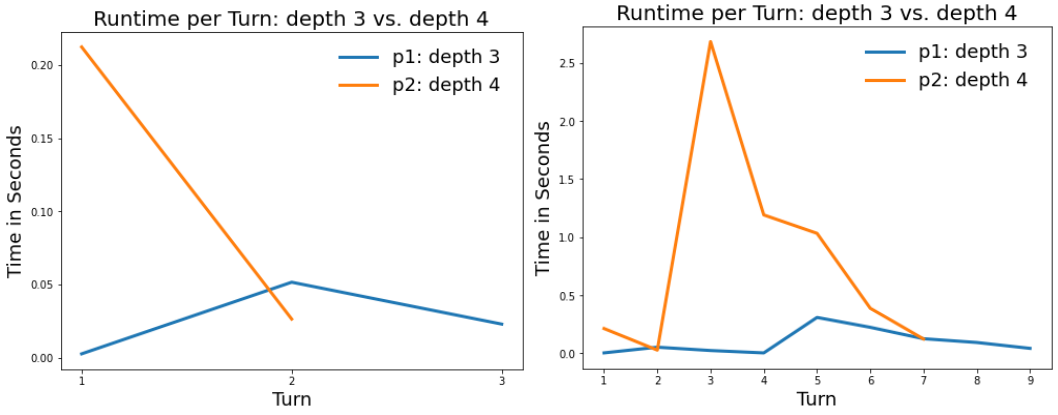
(a) Board size of 3×3.

(b) Board size of 4×4.



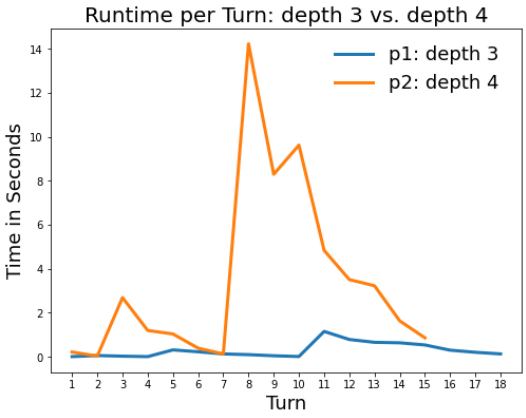
(c) Board size of 5×5.

Figure 4: Number of cutoffs for varying board sizes averaged over 25 games.



(a) Board size of 3×3

(b) Board size of 4×4



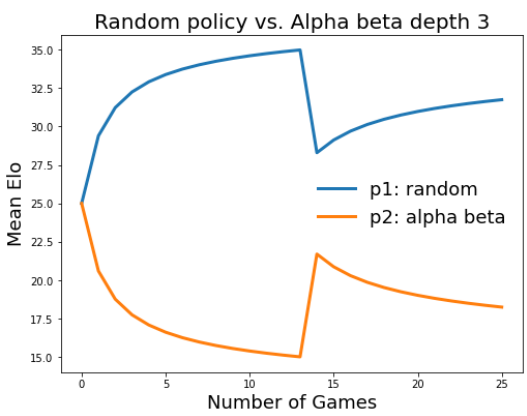
(a) Board size of 5×5

Figure 6: Run-time per turn for varying board sizes.

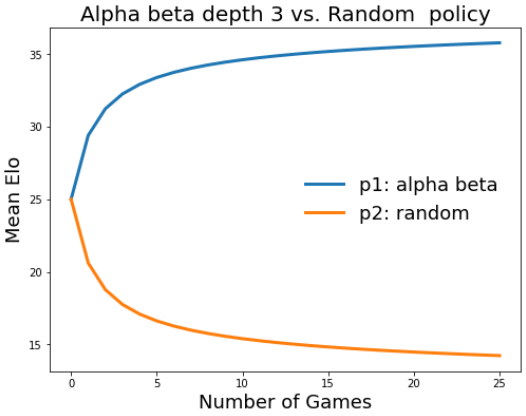
3.1.2 Elo Rating

Elo ratings from the match-ups in Table 2 are depicted in Figure 7, 8 and 9. Reverse player order are included on the right panels.

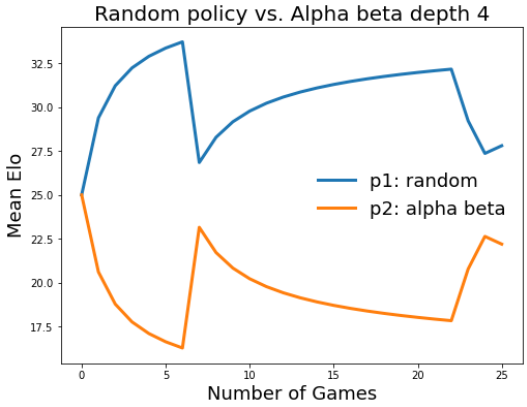
For a board size of 3×3, when the starting player is random as shown in 7a, 7c and the starting player advantage occurs (pie rule not applied), the random decision policy is able to compete with alpha beta. A random policy wins the majority of the games. However when alpha beta starts first as shown in Figure 7b and 7d this effect is reversed. The effect of this order dependence is further illustrated in 7e and 7f. Both agents use the starting advantage to win the game. As a result the difference between alpha beta with depth 3 and 4 on a board size of 3×3 is not visible. This this is to be expected as both players search until the full depth of the board is reached and can come up with similar moves. Therefore order matters. Convergence occurs after approximately 15 games, but may not settle in as some match-ups (i.e Figure 7a, Figure 7c and Figure 7d).



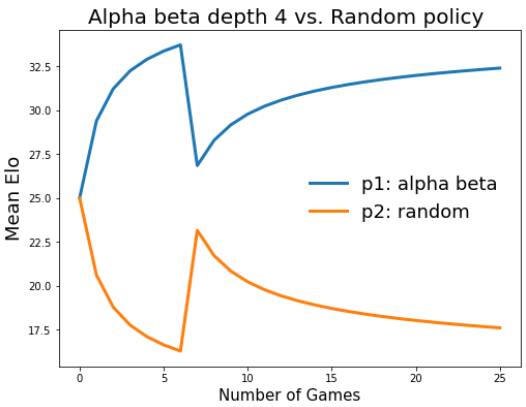
(a) Random policy starts first.



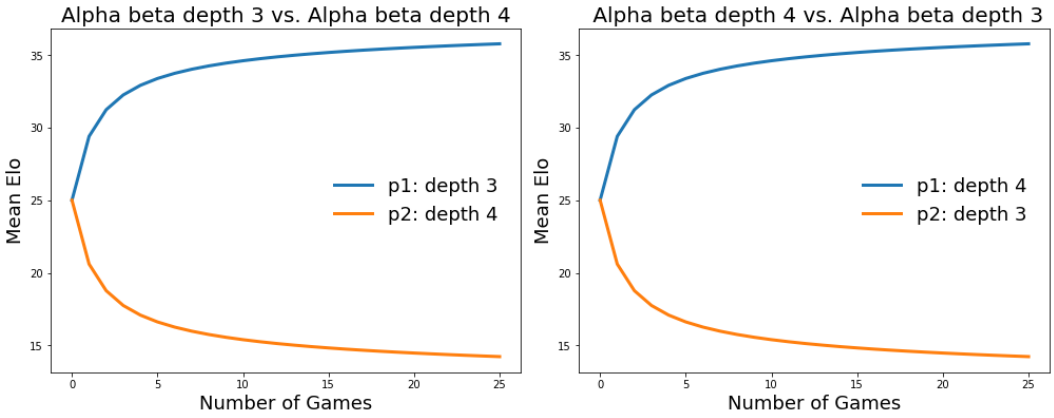
(b) Alpha beta starts first.



(c) Random policy starts first.



(d) Alpha beta starts first.

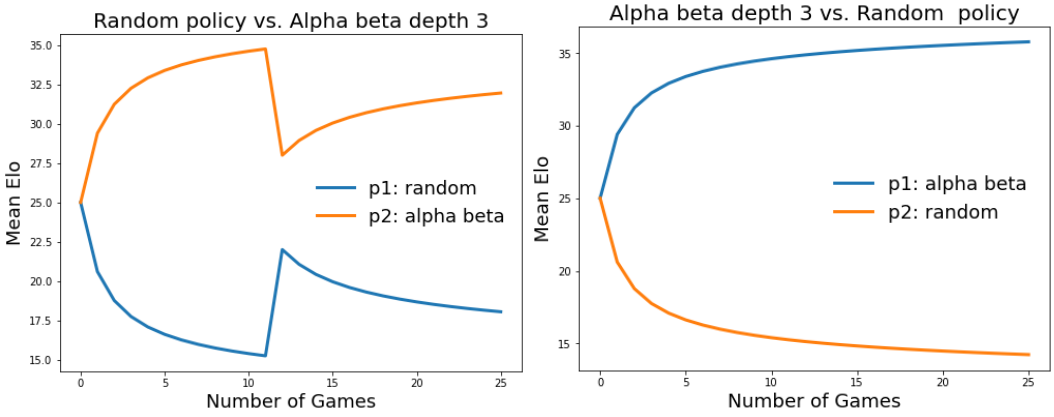


(e) Alpha beta with depth = 3 starts first.

(f) Alpha beta with depth = 4 starts first.

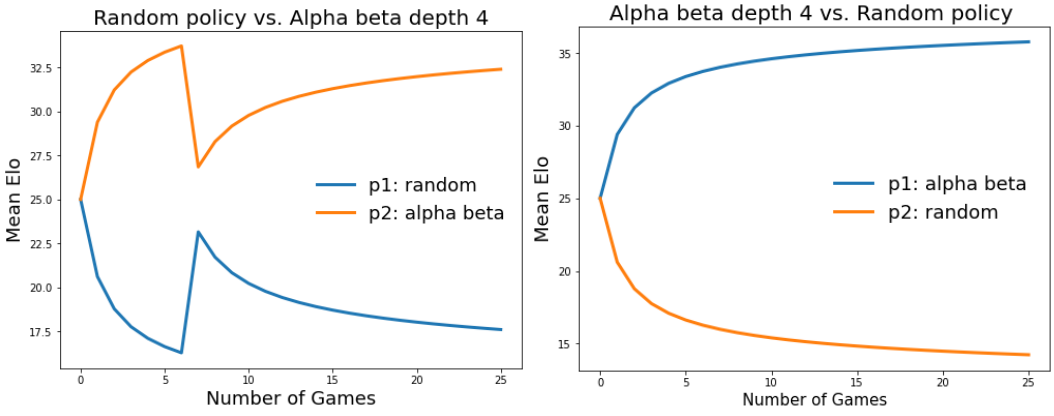
Figure 7: Elo Ranking of regular player match-up on the left and reverse player match-up on the right on a board size of 3×3.

For a board size of 4×4 (see Figure 8), the trend changes. Even-though the agent utilizing a random policy starts first, alpha beta with depth 3 wins the majority of the games (see Figure 8a). The same occurs for when the random agent plays against alpha beta with depth 4 (see Figure 8c). As such a big difference to the previous board size is that alpha beta at depth 3 and 4 improve compared to a board size of 3×3. The random agents can however still manage to win sometimes. Therefore, the correlation between starting order and winning is still visible, even-though to a lesser degree. When alpha beta starts against the random policy as in Figure 8b and Figure 8d it wins more often. Furthermore, alpha beta with depth 3 is still able to win against alpha beta with depth 4 (see Figure 8e and Figure 8f). This shows that using alpha beta with depth 3 is enough to find the best move, even if the board is not searched until its full depth. Compared to a board size of 3×3, mean Elo ratings begin to settle in more slowly as both ratings for player 1 and player 2 move away from each other. Converge however does not improve by a lot.



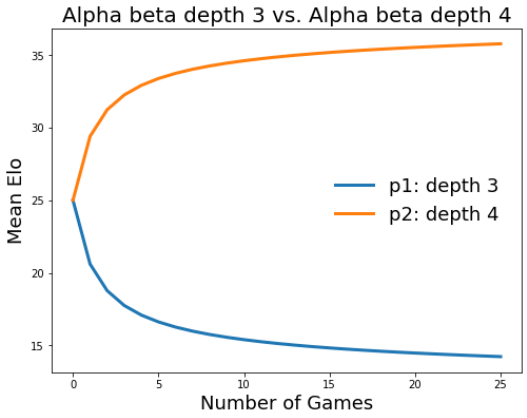
(a) Random policy starts first.

(b) Alpha beta starts first.

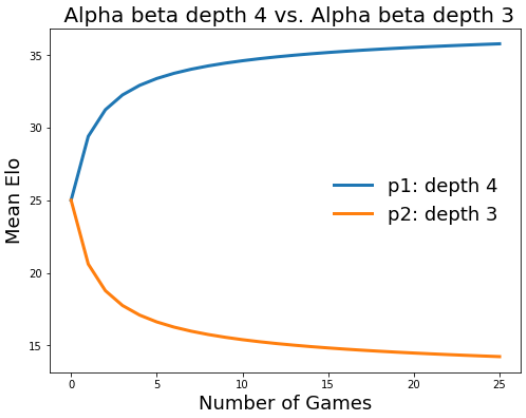


(c) Random policy starts first.

(d) Alpha beta starts first.



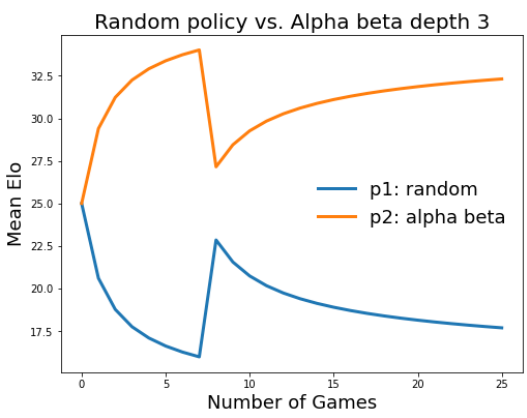
(e) Alpha beta with depth = 3 starts first.



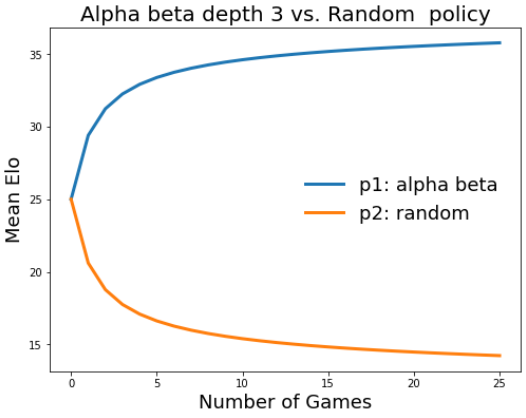
(f) Alpha beta with depth = 4 starts first.

Figure 8: Elo Ranking of regular player match-up on the left and reverse player match-up on the right on a board size of 4×4.

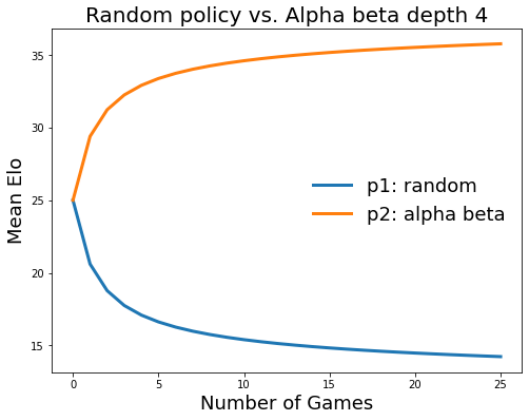
The experiments on a board size of 5×5 reveals that alpha beta with depth 3 and depth 4 converges faster than on previous boards as shown in Figure 9a, Figure 9c, Figure 9b and Figure 9d. Convergence occurs at approximately 10 games as ratings between both players move away from each other. The effect of starting order dependence only occurs when alpha beta with depth 3 and alpha beta with depth 4 play against each other (see Figure 9e). The effect holds in reverse order as shown in Figure 9f). Therefore, a depth of 3 is still enough to find the best moves, since alpha beta with depth 3 can still win against alpha beta with depth 4, even though both players cannot search until the full depth of the board.



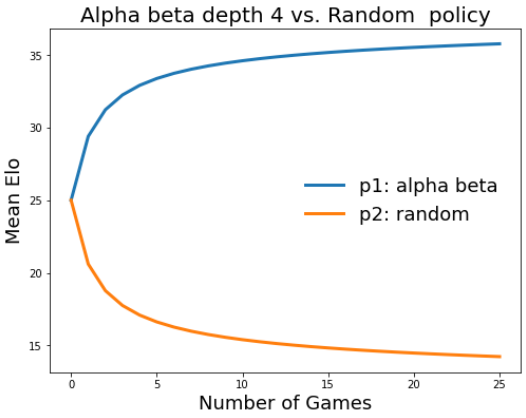
(a) Random policy starts first.



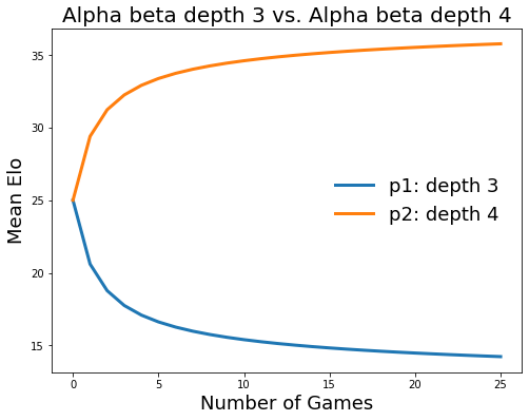
(b) Alpha beta starts first.



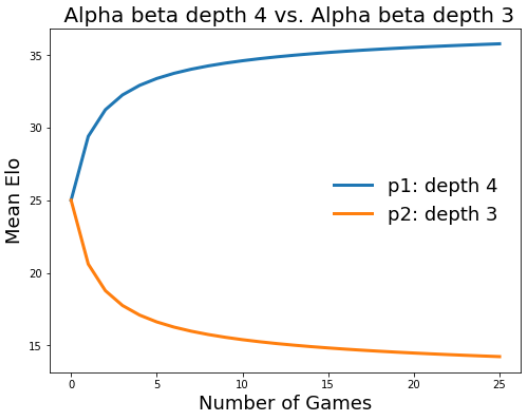
(c) Random policy starts first.



(d) Alpha beta starts first.



(e) Alpha beta with depth = 3 starts first.

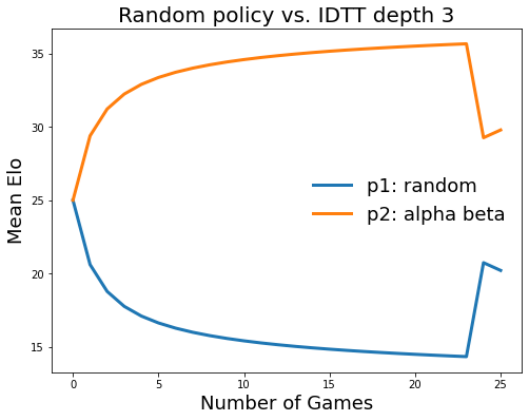


(f) Alpha beta with depth = 4 starts first.

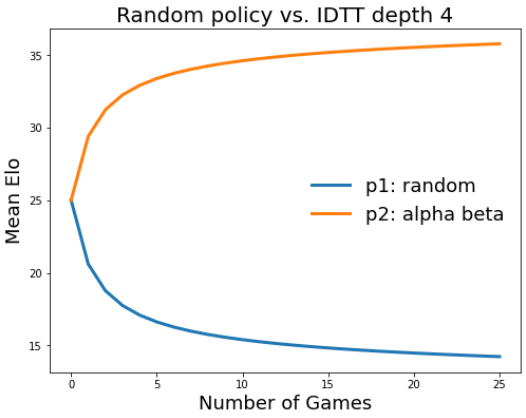
Figure 9: Elo Ranking of regular player match-up on the left and reverse player match-up on the right on a board size of 5×5.

3.2 Iterative Deepening and Transposition Tables

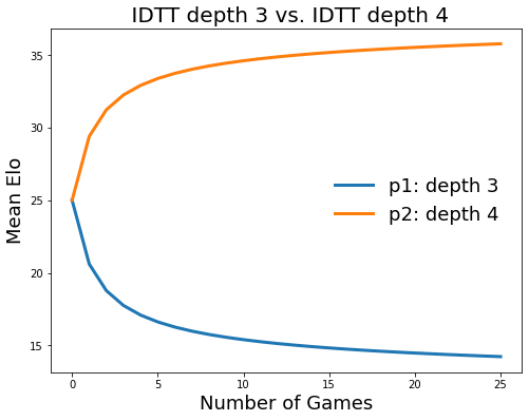
Mean Elo ratings of the experiments regarding IDTT on a board size of 5×5 are shown in Figure 10. The Elo ranking of the experiments using iterative deepening and transposition tables do not differ from the Elo ranking of a regular alpha beta without enhancements as in Figure 9. It would seem that given the present board size iterative deepened searches do not yield a performance boost, but could potentially when size increases.



(a) Random policy starts first.



(b) Alpha beta starts first.



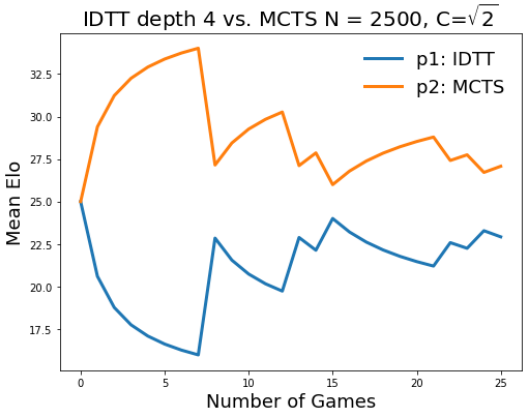
(c) Random policy starts first.

Figure 10: IDTT experiments on a board size of 5×5.

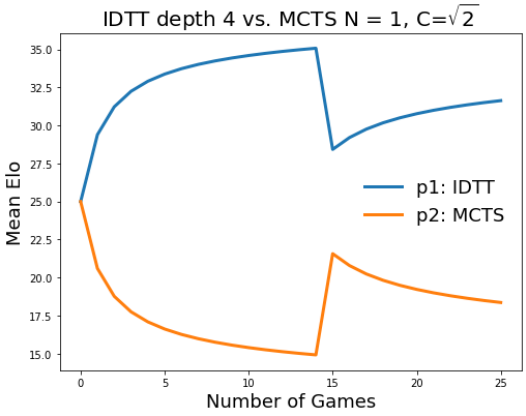
3.3 Monte Carlo Tree Search

Mean Elo ranking for the MCTS are shown in Figure 11. With 2500 simulations and $C = \sqrt{2}$ MCTS outperforms the strongest searcher IDDT with depth 4 from the previous

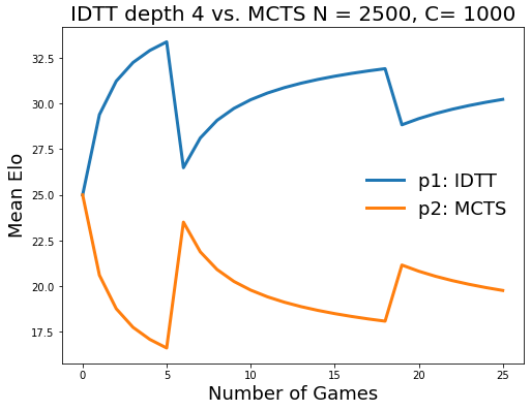
experiments (see Figure 11a). IDTT is however still able to win a game from time to time. When the number of simulations N are lowered to $N = 1$ MCTS performs worse than IDTT with depth 4 (see Figure 11b). The performance of this match-up is then similar to a random agent playing against a regular alpha beta with depth 3 on a board size of 3×3 as shown in Figure 7a. Degeneracies also occur when the exploitation/exploration parameter $C = 1000$ as visible in Figure 11c. This reflects a relatively high value for this parameter and MCTS again performance worse than IDTT with depth 4.



(a) Random policy starts first.



(b) Alpha beta starts first.



(c) Random policy starts first.

Figure 11: MCTS Experiments on a board size of 5×5.

4 Discussion

On all board sizes, the mean Elo ratings converge to a value of approximately 35.0 for both alpha beta with depth 3 and 4. As for a random decision policy values converge around a mean Elo of 15.0. As board sizes increases, a random decision policy cannot keep up with the complexity of the board and begins to loose more frequently. A similar phenomena is visible for alpha beta player with depth 4 against depth 3. Depth 3 cannot keep up with the increasing complexity and starts to lose more often. This is surprising since up until a board size of 4×4 alpha beta with depth 3 could keep up with depth 4, even-though it did not search the entire depth. Furthermore, as board dimensions increase the correlation between starting player and winning a match decreases. This form of order dependence disappears almost entirely on a board size of 5×5 and should completely disappear with an even larger size. As for the alpha beta enhancements, performance is left unaffected, however computational time can be reduced if a transposition table is used to store previously encounter scenarios. It comes as a surprise, that iterative deepening did not yield noticeable performance boost. We believe that this is likely due to two reasons; 1) the time limit constrain of 2 seconds did not allow for much improvements and 2) the relatively small board sizes did not allow for a much deeper search. Our second argument is supported, since a regular alpha beta at a shallower depth 3 could win against a more sophisticated depth 4 player, even if the depth 3 player did not have the entire depth of the game tree available.

With respect to the MCTS searcher, performance is severely affected by the hyper parameter settings of C and N. In general as the number of simulations N increases so does

performance given that C is chosen wisely. In the experiments we utilize a common value $\sqrt{2}$, which works reasonably well. However when drastically changed to a much larger value, performance of the MCTS regardless the number of simulations chosen degrades. The MCTS agent then behaves similar to a random agent. This is to be expected since large values of C can be regarded as a form of curiosity parameter. Large values increase the agents curiosity and it will explore its actions space with the hope of finding a better solution (i.e. node in a branch). As such the agents explores the actions space. If a small value is used, the agent explores other branches and nodes less and is said to exploit its environment. We would advise this parameter to be chosen empirically using a grid-search.