0 Introduction

This assignment is focused on providing a fast, yet high-performing, agent for the game tic-tac-toe. This was split into 4 weeks, each spent on a different aspect of the agent. The first week began with a random-agent, which was fast but nonoptimal. This, however, did provide a base for comparison for the future agents. The second week was spent making a depth first search algorithm to expand the minimax tree, along with a few time-based optimizations. This agent was optimal but relatively slow. In the third week, alpha-beta pruning was introduced (along with heuristics for move ordering) to improve the speed of the depth first search while maintaining optimality. In the final week, these agents were tested on higher-level boards, and a new agent was implemented to provide some sense of high performance while meeting time constraints.

1 Week 1

1.1 Methods

For the first week, I implemented the random agent in `tic_tac_toe/agents/random_agent.py`. I also implemented a decorator class in `tic_tac_toe/agents/timed_agent.py` for timing and other statistics. Small changes were made in `main.py` and `tic_tac_toe/game.py` to record all the statistics needed for the report.

I have also begun working on the minimax agent, which I believe is fully functional. I may end up restructuring the code here in order to simplify the procedure for alpha-beta pruning in the following weeks.

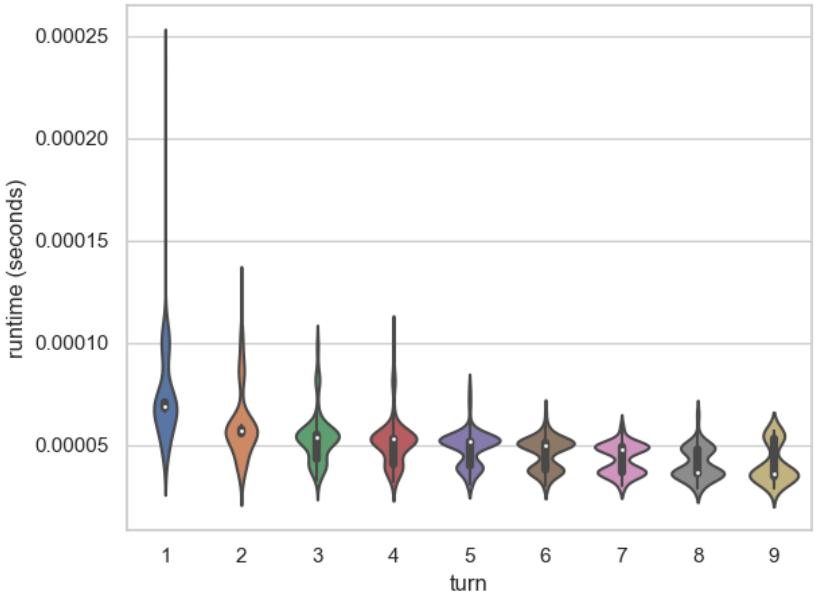
1.2 Results

The following table shows the results of x/o wins and draws from 100 random games.

x	**o**	**draws**
55	31	14

As expected, the first agent (who plays x) generally has an advantage in a completely random game. This is likely due to the fact that the amount of x's is, at any given time, either equal to or 1 higher (a significant porportion in short games) than the amount of o's on the board.

The next plot shows the distribution of runtime per turn for random agents over the same 100 games.



Runtime ranged from about 35 to 65 milliseconds per move. There is also a fairly steady descrease in runtime as the game progresses, which is likely due to the process of the `valid_moves` method. This method constructs a `Move` object for each empty cell on the board, so less constructions will take place as the board fills up. The `random.choice` method itself completes in constant runtime.

2 Week 2

2.1 Methods

For the second week, I added a depth-first minimax agent to the tic-tac-toe program. In its original version, nothing was optimized, and the agent would expand the entire minimax tree. Then, as this program took around 40 seconds per game, three optimizations were successively added upon the program.

The first optimization was eliminating the `deepcopy` method, where the board was originally copied for each expansion. Instead, moves were made in-place on the board, and were undone when that branch had been checked.

On top of this, I began caching board states. Hence, if a node had already been checked and cached, it would not be expanded. This approach recognizes that the configuration of nodes is truly a graph (not a tree), as there are different permutations of moves that lead to the same state. The cache was cleared at the end of each game.

The final optimization was prioritizing early wins. Instead of scoring each leaf node (end-of-game node) simply by whether it was a win/loss/draw (which were scored 1/-1/0), this method would divide the original score by the depth of the leaf node, hence assigning greater priority higher nodes. The intent here is to end the game as early as possible while still attaining the same result (in terms of a win/loss/draw).

2.2 Results

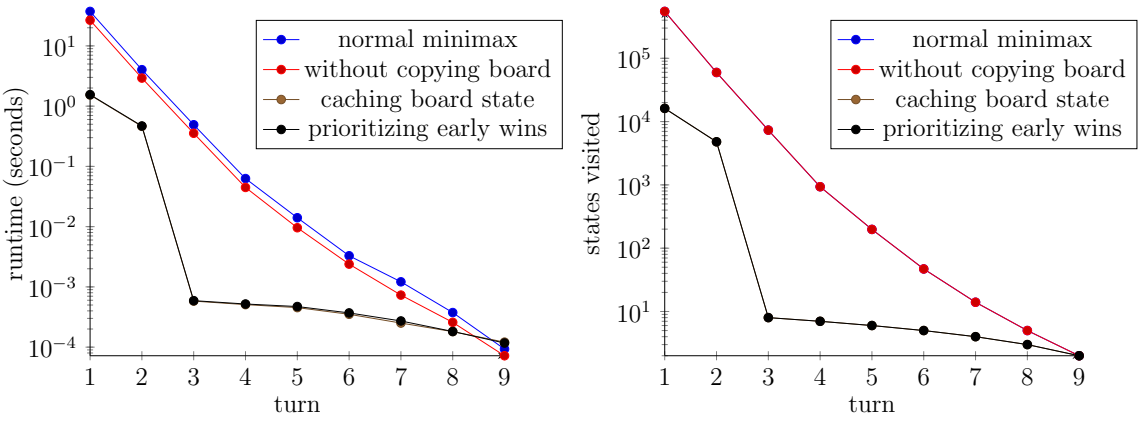
2.2.1 Minimax vs. Minimax

With two minimax agents, no randomness was involved, and hence the game was completely deterministic, ending in configuration below (a draw).

x	x	o
o	o	x
x	o	x

It should be noted that, at the first move, there were multiple moves scored `0` in minimax, but the first (top left) was chosen. This even occured when spots were chosen by priority of depth, as all draws will have depth 9 (a draw is always on a full board).

The two log-scaled graphs below show the results, over 10 games in each of the successive optimizations, of average runtime and states visited. Note that each successive optimization was applied on top of the previous.



In the first graph, we see a general exponential trend: as there are more open squares (at earlier moves), the time increases exponentially (reflecting that minimax trees are exponential in terms of the branching factor). Eliminating board copies slightly reduced the time, in proportion to how many moves were checked in the turn. Caching states reduced the time even more, and drastically after the first two turns (after which the rest of the game would already be cached). These later moves (3-9) are linear, as they depend merely on how many top-level moves (pre-cachced states) they need to check. Although it's impossible to tell with just two points, moves 1-2 are likely less-than exponential, as extrapolating along a linear form on the scaled graph would intersect the more-costly methods. That is, if the cache had been reset at each move, the graph would reflect this less-than exponential form (which makes sense, as branches would be removed from the tree). The last optimization actually slightly increased the time (a negligable amount), as taking depth into account adds constant time at each node.

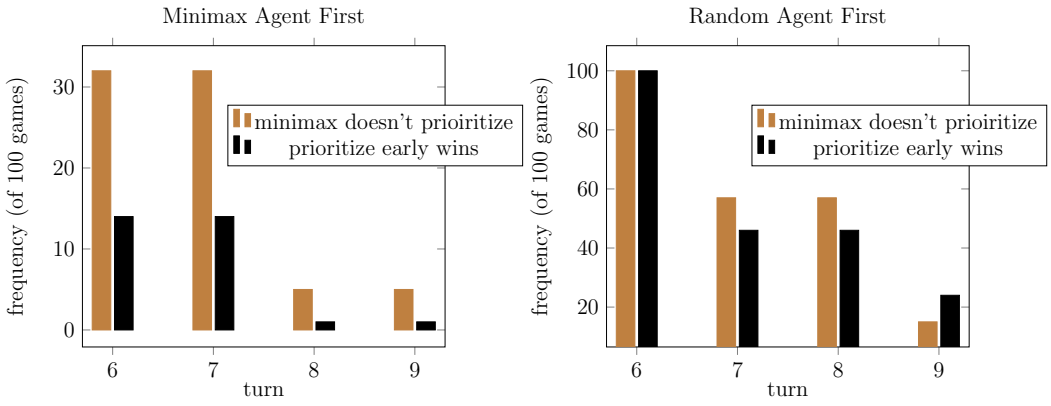
The same exponential trend is shown in the states-visited graph for the original algorithms, as well as the sub-exponential trend for the cached methods. The only optimization that made a difference in terms of states, as expected, was caching the states.

2.2.2 Minimax vs. Random

When playing against a random agent, the minimax agent never loses, as the worst case scenario (seen in minimax vs. minimax games) for any optimal agent is a draw. Below is a chart of the winning percantages (from 100 games each) of minimax agent when playing as either the first or second player. Surprisingly, prioritizing earlier wins seems to increase the chance of a draw, which is likely an effect of the game not being deterministic (i.e., when the expected outcome is a draw, prioritizing early wins a few branches down happens to disfavor branches with more winning chances). As expected, the minimax agent going second reduces its likelihood of winning, as it has less moves and hence less control of the board.

	minimax first	**random first**
no prioirty	99%	85%
prioritizing early wins	99%	76%

The following graphs depicts the effect of prioritizing earlier wins. Games tend to be shorter, whether the minimax agent goes first or second, when it prioritizes earlier wins. Again, games last longer in general when going second, due to the fact that the 'o' player has less control of the board.



3 Week 3

3.1 Methods

For the third week, I worked on alpha-beta pruning for the minimax algorithm. Once this was completed, I updated the board hash to store a `hash` variable, which updates with each move, rather than rehashing completely each time. When using the alpha-beta algorithm, only some states could be cached (i.e., those without pruned descendants), in order to avoid copying faulty scores across the tree to a state where pruning should not be applied.

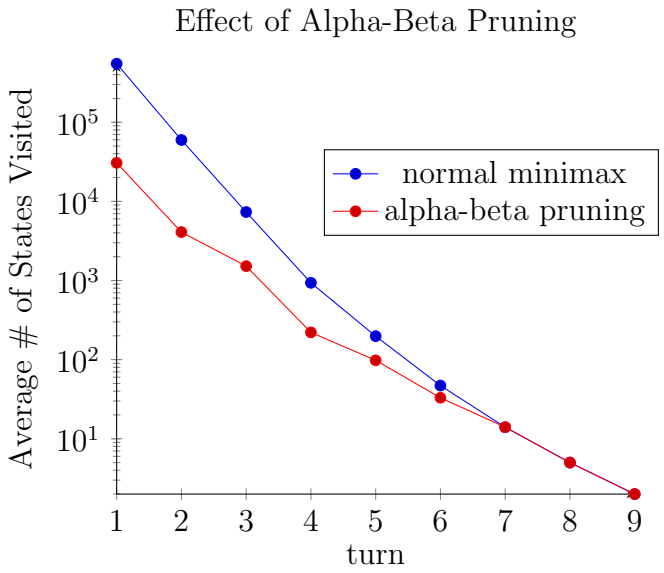
I also added an evaluation function, by which moves would be ordered. This works separately from the result function (leaf nodes), but does begin by checking if the game is in a terminal state and returning $-\infty$, 0, or ∞ when appropriate. Otherwise, all game lines are considered, and if the opponent has the opportunity to win in the next move, $-\infty$ is returned. If the current player has a fork (two opportunities to win, so the opponent has no chance), ∞ is returned. Otherwise, for each line that the player or opponent has control

over, the number of squares taken in that line is added/subtracted (respectively) from the score. This score is then returned as the evaluation.

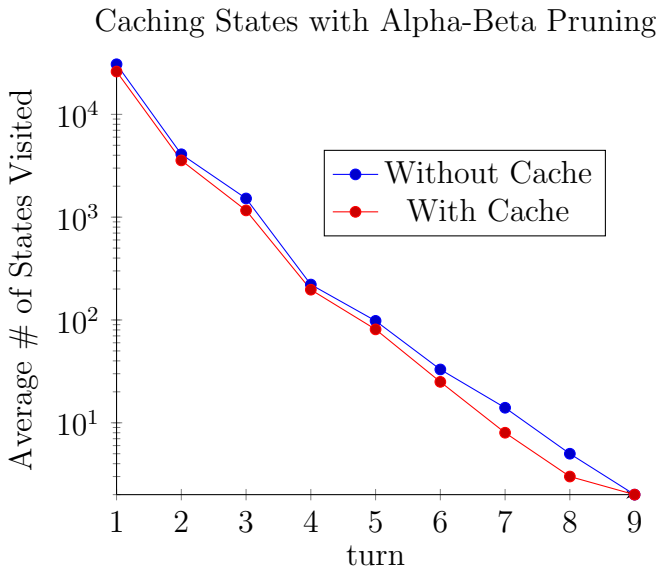
As shown by the following results, more improvements may need to be made to the evaluation function. This can be achieved through further testing and experimentation.

3.2 Results

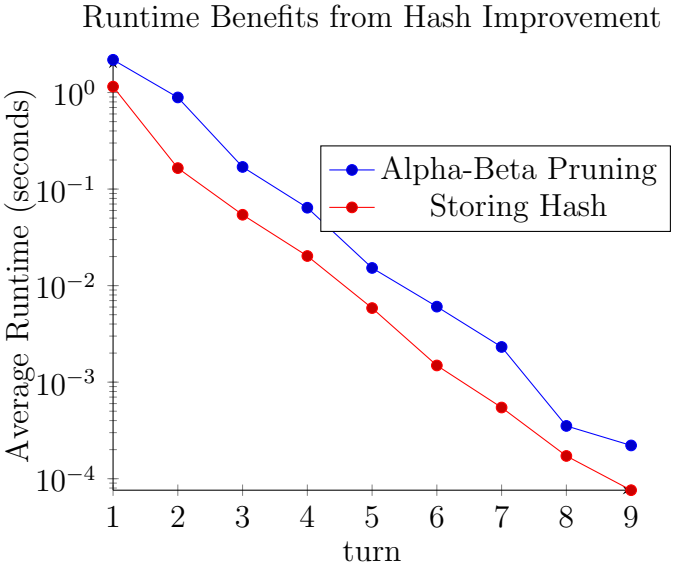
The following graph shows the result of applying pruning (without cache). This is most effective on the first move, where the pruned method visits only about 5% as many states as the original minimax algorithm.



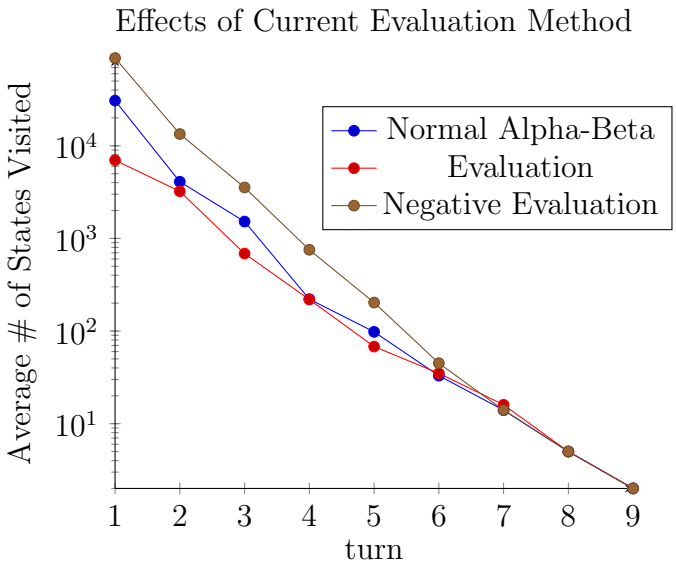
Caching did eliminate a few thousand states, but did not make a large difference due to the fact that not many states were cached. Note that there are still some uncached states visited after move 2, due to the fact that not all states were cached (those with pruned descendants were not).



The next graph highlights that the new hashing method (storing the hash rather than rehashing) has a consistent runtime improvment.



The following graph shows a comparision of states visited from a normal alpha-beta search, and two searches where moves are ordered by the current evaluation, in both a normal/positive and a reverse/negative fashion. All of these were done without a cache. There is a fairly consistent improvement caused by move ordering. Although the scale under-exaggerates it, move ordering visits just 25% as many states as the default order.



This evaluation results in the following board state, with the first move occuring in the middle cell (with the highest heuristic):

o	**x**	**x**
x	**x**	**o**
o	**o**	**x**

The reverse evaluation ends in this state, with the first move being in the middle of the top row (tied for lowest heuristic value):

o	**x**	**o**
x	**o**	**x**
x	**o**	**x**

4 Week 4

4.1 Methods

For the last week, I worked primarily on extending the previous algorithms to $N \times N$ boards. Beforehand, I worked on improving the cache for alpha-beta pruning. For every board state that is encountered, a score is stored along with a flag: 0 for exact, -1 for lower bound, and 1 for upper bound. If the score is not more than the alpha value used to get there, there may have been pruning at some point that did not allow a lower score to appear (and an upper bound flag is used). Similarly, if the score is at least as large as beta, it is treated as a lower bound. Then, when searching the cache, only those flagged as exact (0) are used. If the cached value is an upper bound, beta is updated to the upper bound if it's lower. Similarly, for a lower bound, alpha is updated to the bound if it's higher. Hence, if alpha becomes at least the value of beta, the node can immediately be pruned (either the score is at least beta or at most alpha, in which cases the node will end up being pruned).[1]

[1]Based on an algorithm in Chapter 2 of *Memory versus Search in Games* by Dennis M. Breuker.

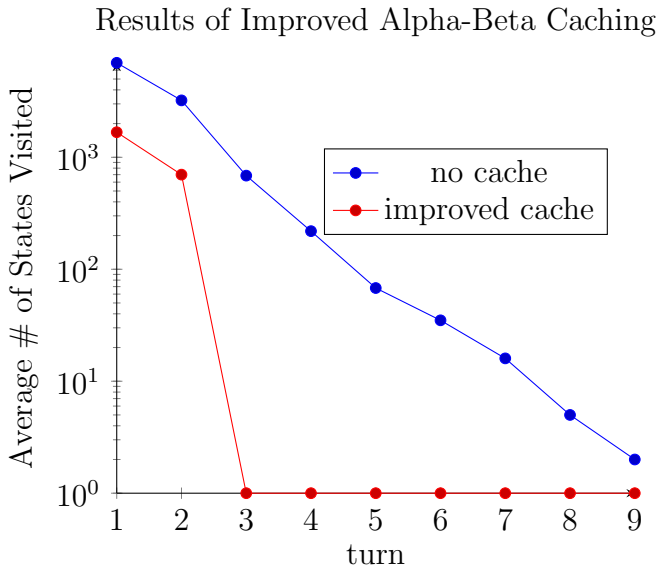
For the first experiment, I tested both the cache-based DFS algorithm against the above alpha-beta (cached) algorithm, ordering moves by heuristic calculations, against higher level boards. The heuristic/evaluation method was updated to work for $N \times N$ boards, including those with winning sequences of length $k < N$. This was done by considering, within each line, connected subsequences of cells of length k. For instance, when $N = 4$ and $k = 3$, the first and last three cells would be considered in every line of length four.

I then worked on an iterative deepening approach, where an additional cache was used to keep track of move history. This made use of three caches. The `cached_leaf_nodes` dictionary kept track of the leaf nodes (where nodes at `max_depth` were considered leaves) using the alpha-beta method above. This was cleared at each iteration. The `last_iter_cache` stored the most recently found score for each board (even if pruned, for simplicity) and was used for move ordering. If move ordering failed to find a historical store in this cache, the evaluation/heuristic method was used (with its own cache to speed up reevaluations).

A .25 second timer was added to this method, so the algorithm would work down as many levels until time ran out. This worked up through 9×9 boards, but 10×10 and higher boards could not complete even one level of evaluations in time. This could be improved by creating a beam-size of, for example, 81 moves, but going up to 9×9 worked for our purposes.

4.2 Results

The following chart shoes the results shows a comparison of the new caching method and no caching for a move-ordering based alpha-beta pruning algorithm. For the first move, the cached version visits under a quarter of the states as the non-cached version. Also, after each agent has gone through the tree once, the score can immediately be returned, as the deterministic game (2 optimal agents) does not reach any states that were originally pruned.

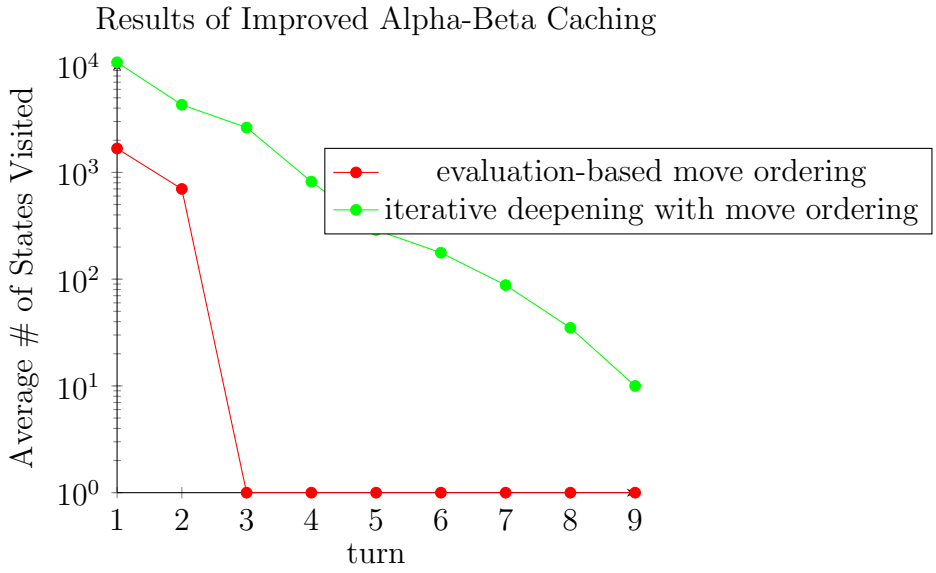


The following chart shows how many states are visited on the first move when we increase both N (board size) and k (winning sequence length), for both normal dfs/minimax, and an

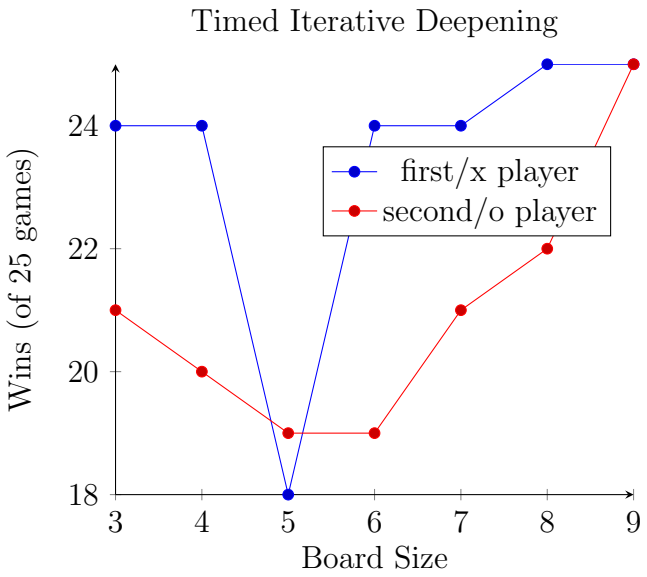
alpha-beta algorithm with move ordering. The alpha-beta agent seems to stay within a more controlled range when increasing N, whereas dfs stays more controlled than alpha-beta when increasing k. However, limited data is shown due to memory constraints. When $k < N$, at least for $k = 3$ the first agent wins when playing optimally, hence decreasing the depth of the board.

$(\mathbf{N}, \mathbf{k})$	$(3, 3)$	$(4, 3)$	$(4, 4)$	$(5, 3)$
alpha-beta	1675	25589	898150	696412
dfs/minimax	16168	23453345	51562425	

The next graph shows a the number of states visited with iterative deepening compared to the normal algorithm (with heuristic based move ordering, same as shown in red on 2 figures up). Note that iterative deepening performs worse, as the agent must perform algorithm level-by-level, resetting each time. Iterative deepening is thus, at least for small tic-tac-toe games, not ideal if we wish to expand the entire minimax tree. Part of this is due to the fact that the heuristic works well for move ordering.



Placing a .25 second time limit on the agent fixed this issue, while maintaining some level of optimality through heuristics. The following graph shows the number of wins of 25 games for each board size (where $k = N$) against a random agent, for cases where the iterative deepening agent goes first and second.



On the blue line, this is likely due to it being the midpoint between when the timed agent can expand the majority of the tree (as in smaller boards) and when the random agent plays poorly enough to win regardless (as in the larger boards).

In reality, the higher level games are easier to win against a random player, as there is more board space for the random agent to "miss" blocking the iterative-deepening agent. Against any agent with some understanding of strategy, it would be extremely easy to block any win, and most games would end in draws. However, especially at the lower level boards, the proportion of wins seems to show that the agent can play somewhat optimally, even without fully expanding the minimax tree. It is also notable that the random algorithm never won, even when going first.