The notion of 'depth' in games: a case study with Quixo

Student: Jau Tung Chan (jc3395)

Advisor: Professor James Glenn

0 Abstract

The notion of 'depth' is an informal concept commonly used within the game designing and playing community, to describe how absorbing or how profound a given game is. In 2017, Lantz et al. proposed a formal conceptual model to define 'depth', based on a strategy ladder graph of plotting "Solution Strength" against "Computational Resources", for a computational learning agent learning to play a given game.

This project applies this strategy ladder model to the board game of Quixo — and in particular, to variants of Quixo played on different board sizes (namely, 5×5, 4×4, and 3×3). Two computational learning agents — a Monte Carlo Tree Search (MCTS) agent and a Q-learning agent — were implemented and investigated.

For this project, the *x*-axis of "Computational Resources" is quantified by the training time of the computational learning agent; while the *y*-axis of "Solution Strength" is quantified by the performance of the computational learning agent, in comparison to the exact optimal solution for Quixo (which was also implemented and computed for this project). Three such quantifications for the *y*-axis were used — via (1) game outcomes, (2) move accuracy, and (3) game length.

The main result of this project is that the strategy ladder model *can* be used to reveal differences in 'depth' between the different sizes of Quixo — specifically, 'depth' in Quixo is positively correlated with board size, as intuitively expected. However, some quantifications for the *y*-axis reveal this phenomenon more strongly than others — namely, quantification via game outcomes yielded the most significant results. Alongside this main result, other notable features of Quixo — that Quixo is a 'mistake-punishing' game, and that Quixo is difficult to learn based only on easily human-identifiable features — were incidentally revealed by this project's investigation.

1 Introduction

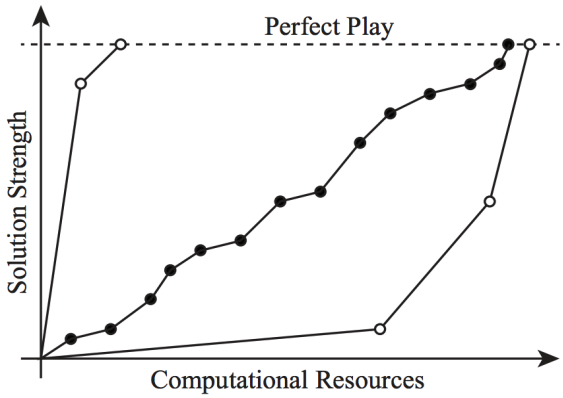
1.1 The Notion of 'Depth' in Games & Lantz et al.'s Strategy Ladder Model

The notion of 'depth' in games has long been referenced within the game designing and playing community, to refer to an intuitive notion of how absorbing or how profound a given game is (Lantz et al., 2017). In their paper, Lantz et al. describe how some games like tic-tac-toe are intuitively extremely 'shallow', in comparison to the canonical 'deep' games like chess or Go, which both have a large community of dedicated players continuously being able to improve themselves at these games, even after hundreds of years. Lantz et al. qualitatively define this notion of 'depth' in games as the game's *"capacity to provide a lifetime of study, learning, and improvement"*.

Despite this qualitative definition, this notion of 'depth' is still almost always used informally and/or qualitatively among the game designing and playing community; and there is currently no

widely-accepted formal quantitative definition of 'depth'. Lantz et al. discuss a couple of potential formal quantitative definitions of 'depth' that can be defined based on its strategic game tree — such as the branching factor, the game strategic length, and the state space — before coming to the conclusion that adversarial constructions of intuitively 'shallow' games can be used to defeat all of these formal quantitative definitions.

This leads to the main contribution and conclusion of Lantz et al.'s paper: a formal conceptual model that can be used to define 'depth'. This conceptual model is based on the idea that a greater number of distinct steps on the 'skill chain' of learning the game, corresponds to more opportunities to learn and improve at the game, which gives rise to the intuitive notion of 'depth' — where the 'skill chain' of learning the game results from players improving at the game through repeated playing and/or studying of the game. This conceptual model can thus be summarized by the following strategy ladder graph (from Lantz et al.):



Each strategy ladder is constructed for one particular game (hence, three example games are represented with the three ladders above), and each dot on the ladder represents a *"complete, fully-defined algorithmic strategy"* that corresponds to the best strategy that can be implemented to play this game, subject to some constraint on computational resources. The constraint on computational resources is plotted on the *x*-axis, while the solution strength of the resulting strategy is plotted on the *y*-axis. The positive slope of the ladder can then intuitively be interpreted as a player getting better at the game, through spending more time playing and/or studying the game.

Specifically, the relevance of this ladder to 'depth', according to Lantz et al., is the number of dots on the ladder from 0 on the *x*-axis (i.e. a player with no computational resources, or correspondingly, a first-time player of the game), to when the player attains perfect play (or asymptotically doing so). Lantz et al.'s point is that more dots corresponds to more 'depth'. To illustrate this, the game represented by the left ladder yields itself quickly to perfect play (i.e. it is too easy), while the game represented by the right ladder requires a lot of computational resources to even be played decently (i.e. it is too hard). Both of these games have few dots on their ladders, and so both of these games are not 'deep', in that they both do not allow players to improve at the game through repeated playing and/or studying. In contrast, the game represented by the middle ladder allows a player to reach various intermediate stages of incremental improvement as they repeatedly play and/or study the game, and is hence

'deeper' than the other two games. In this way, according to Lantz et al., the 'depth' of a game can be quantitatively defined or measured by the number of dots on, or the shape of, its strategy ladder.

1.2 Quixo

Quixo is a two-player turn-based board game produced by Gigamic. Quixo is played on a 5×5 grid of tiles, where each tile is either empty, or contains the symbol of one of the two players — X or O; and the grid starts off with all empty tiles at the start of the game. The players take turns to make moves on this 5×5 grid of tiles, and the objective of the game is to be the first player to form 5-in-a-row of their symbol. The game rules also dictate that, if 5-in-a-rows of both symbols are created simultaneously with a single move, then the player making this move loses (i.e. the opponent wins). Even though this is not in the official game rules, for consistency, and without loss of generality, this paper will always assume that player X makes the first move (i.e. on the initial board with all empty tiles).

The interesting thing about Quixo is the mechanism used to make a move, which is more complex than simply placing an X or an O in an empty tile of the grid (as is the case in tic-tac-toe). For Quixo, a move consists of two parts: (i) the active player removes a tile that is either empty or already contains their symbol from the border of the grid, and (ii) that player then transforms that tile to contain their symbol, and subsequently slides that tile back into the grid by pushing one row or column of tiles to fill the gap created by the tile removal in (i). For further clarification on this, a PDF of the official rules can be found here, the official Gigamic website for Quixo is here, a video tutorial of Quixo can be found here, and an example photo of the physical Quixo game board is as follows:

From a computational intelligence perspective, there are several notable features about Quixo. Firstly, Quixo is a combinatorial game (i.e. it is a two-player turn-based deterministic game where both players have perfect information about the game tree). So, every board state can be classified into one of 3 types — first-player-win, first-player-lose, or draw. Secondly, Quixo's game tree can contain cycles (i.e. the same game state could recur after several moves by both players). Thirdly, and as a result of the second feature, Quixo is therefore not a finite game (i.e. a single game can take arbitrarily many turns, without either player winning).

1.3 Project Goals

The primary goal of this project is to investigate the 'depth' of Quixo, via the strategy ladder model proposed by Lantz et al. Furthermore, since the rules of Quixo are easily adaptable to variants that use either a smaller or a larger board (e.g. 3×3, 4×4, 6×6), and assuming (reasonably) that variants using

larger boards are intuitively 'deeper' than variants using smaller boards — in other words, assuming that 'depth' is positively correlated with board size — the secondary goal of this project is to explore whether these differences in 'depth' between variants of Quixo can be revealed by this strategy ladder model.

To more concretely specify the goals of this project, it is necessary to quantify the two axes of the strategy ladder graph. For this project, the *x*-axis of "Computational Resources" will be quantified by the amount of time given to a computational learning agent for learning the game; while the *y*-axis of "Solution Strength" will be measured by the performance of this computational learning agent, in comparison to the exact optimal solution for the game (i.e. the perfect player). The specific quantification of this notion of performance will be further elaborated on in Section 3.

At this point, it is worth noting that the game of Quixo is well-suited for this 'depth' investigation because it has a game tree that is small enough for the exact optimal solution to be computable, while still being large enough for this exact optimal solution to be non-trivial.

1.4 Paper Outline

Section 2 covers the implementation details for this project. Section 2.1 focuses on the Quixo game engine, Section 2.2 focuses on the computation of the exact optimal solution of Quixo, while Sections 2.3 and 2.4 focus on two computational learning agents — a Monte Carlo Tree Search (MCTS) agent and a Q-learning agent — for learning Quixo.

Then, Section 3 presents the various results of experimental investigations, including several strategy ladder graphs, which use different quantifications of the performance of the computational learning agent for the *y*-axis. Finally, Section 4 summarizes and concludes these results with respect to the project goals, including a discussion of possible future research directions.

2 Implementation Details

The codebase for this project is publicly available here on GitHub, where more detailed usage information is provided in the README. Note that most of the implementation details for the Quixo game engine, as well as for the computation of the exact optimal solution of Quixo, are adapted from Tanaka et al.'s paper, "Quixo is Solved" (2020).

2.1 Game Engine

2.1.1 Tanaka et al.'s Representation of Quixo

As is done in Tanaka et al.'s paper, a given state of the Quixo board is stored as an unsigned 64-bit number (`typedef`-ed as `state_t` in this project's codebase), where the most significant 32 bits are used to store the positions of the X tiles on the board, and the least significant 32 bits are used to store the positions of the O tiles on the board. Specifically, within each of these blocks of 32 bits, the least significant *n* bits of that block are used to denote the presence of an X tile (or respectively an O tile) at each of the *n* positions on the board, in row-major order — where *n* is 25 for 5×5 Quixo, 16 for 4×4 Quixo, and so forth. The primary advantage of this representation of states is that making a move according to Quixo rules — or reversing the effect of a move (which is useful for computing the exact optimal solution, as will be elaborated on later) — becomes very efficient, involving only 1 bit-shifting

operation and 5 bitwise and/or operations, with constants that can be precomputed (more details provided in Tanaka et al.'s paper). Unfortunately, however, the notable disadvantage of this representation is that rotational and reflection symmetries of the board are not easily computable or utilized at all using this implementation.

A given move on the Quixo board is stored as an unsigned 16-bit number (`typedef`-ed as `move_t` in this project's codebase), encoding all of the direction, row, and column of the move, whether the move involved picking up an already-marked tile (i.e. either an X or an O) or an empty tile, as well as whether the move was a normal move or a reverse move.

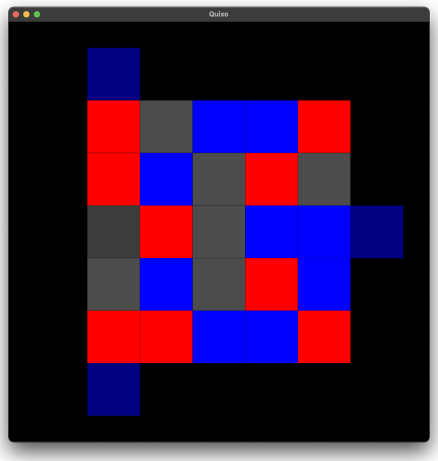
These two design choices allow many functional operations on the Quixo board (e.g. `makeMove()`, `swapPlayers()`) to be implemented with minimal memory allocations, thus reducing memory overhead and increasing the efficiency of the program. These functions are implemented in the `GameStateHandler` class in this project's codebase.

2.1.2 This Project's Implementation of the Game Engine

Based on the representation of Quixo as described above, the game engine used to handle players X and O taking turns to make moves (i.e. playing against each other) is implemented in the `GamePlayHandler` class in this project's codebase. Arbitrary player strategies can be defined and implemented by creating subclasses of the generic `Player` class, with the key `virtual` function to be implemented being `selectMove()` — which is given a specific board state as an argument.

2.1.3 This Project's Implementation of a Graphical User Interface

In addition to the raw game engine described above, a graphical user interface is also implemented in the `GraphicsHandler` class in this project's codebase. This uses the OpenGL framework, and the resulting board looks like the following (where red represents X tiles and blue represents O tiles):



This has three main benefits. Firstly, it allows Quixo to be played interactively by a human player (implemented as the `InteractivePlayer` subclass of the generic `Player` class), whether against another human player, or against one of the implemented agents that will be subsequently described in Sections 2.2-2.4 (here is a video demo of this). Secondly, and relatedly, it allows for a visualization of the

gameplay between two implemented agents (here is a video demo of this). Thirdly, and lastly, it allows for an interactive specification of a particular state of the board (via clicking), which can be used to visualize how implemented agents play when starting from that state of the board, or to evaluate the classification of that state of the board based on the exact optimal solution, as will be subsequently described in Section 2.2.

2.2 Optimal Agent

2.2.1 Tanaka et al.'s Classification of Board States

Again, the implementation of the computation of the exact optimal solution of Quixo is adapted from Tanaka et al.'s paper. Specifically, the goal of this computation is to classify all possible board states of Quixo into first-player-win, first-player-lose, or draw states — henceforth known as win, lose, or draw states for brevity.

The basic premise of this computation is a brute-force backward induction algorithm, combined with value iteration to handle potential cycles in the game tree. Explicitly, this backward induction involves starting by classifying terminal states (i.e. states that have 5-in-a-row of either or both symbols), and then iteratively classifying the parent states after their children states have known classifications:

- having at least one lose child state implies that the parent state is a win state; and
- having all win children states implies that the parent state is a lose state.

Because of potential cycles in the game tree, this must be done iteratively until no new classifications can be made, and then all remaining states with unknown classifications are classified as draw states.

Tanaka et al. describe and implement several optimizations to increase the speed of this brute-force computation:

1. *They partition the set of all possible states into classes, based on the number of X and O tiles in each state.*

   Noting that the parent-child relation between states forces an analogous relation between classes due to Quixo rules, the value iteration can be conducted sequentially on each class rather than on the entire set of all possible states at once. This speeds up the computation since less working memory is needed to store the classifications of all states currently being worked on.

2. *They apply the 'parent link' optimization.*

   As soon as any state gets classified as a lose state, all of its parent states are immediately classified as win states, without having to wait for the value iteration. With this, the value iteration step then only needs to check for states that have all win children states, and does not need to check for lose child states.

3. *They introduce the (first-player-)win-or-draw (temporary) classification for states.*

   Similar to the previous optimization, as soon as any state gets classified as a draw, then all of its parent states are immediately classified as win-or-draw. In combination with the 'parent link' optimization, this improves computation speed as well, since the value iteration step now only

needs to iterate over board states with unknown classifications, and can skip over the known win-or-draw states (noting that the potential re-classification of states from win-or-draw states to win states will still be correctly done via the 'parent link' optimization).

4. *They parallelize the value iteration process within each class.*

   This, of course, requires appropriate mutexes on the data structure storing the classifications of all states currently being worked on. They also use up to 32 threads in total.

For more details on any of these optimizations, Tanaka et al.'s original paper provides further elaborations and pseudo-codes.

2.2.2 This Project's Classification of Board States

This project's codebase implements the brute-force backward induction algorithm, together with all of Tanaka et al.'s optimizations described above, within the `computeAll()` function of the `OptComputer` class. Some noteworthy points about this project's implementation are noted in the following two subsections.

2.2.2.1 Data Structures & Memory Usage

In this project's codebase, the classification of each state into win, lose, draw, or win-or-draw states is stored using (the minimum possible memory usage of) 2 bits (`typedef`-ed as `result_t`). To further minimize memory usage, every 4 classifications is packed into a single 8-bit unsigned integer (`typedef`-ed as `result4_t`), with the appropriate bit-shifts. At runtime, the data structure used to store the classifications of all states currently being worked on is therefore a `std::vector<result4_t>`; while for permanent storage, these 8-bit unsigned integers are written sequentially as `char`'s, as raw bytes, into data files (specifically, data/optimal/*.dat). These functionalities are implemented in the `DataHandler` class.

For a sense of scale, it is worth noting that, even with this minimum possible memory usage, the permanent storage required for the classifications of all states of 5×5 Quixo is $3^{25} \times 2$ bits $\approx$ 200 GB, where 3^{25} is the total number of board states of 5×5 Quixo. For 6×6 Quixo, the number is correspondingly $3^{36} \times 2$ bits $\approx$ 38,000 TB, which makes such a computation unfeasible on most commodity hardware. For the same reason, the permanent data files for 5×5 Quixo are also not uploaded as part of this project's codebase on the linked GitHub.

2.2.2.2 Parallelization Technique

Two subtleties of the parallelization technique implemented for this project (using OpenMP) are also worth noting:

1. *The first involves the granularity of mutexes used.*

   To prevent race conditions based on the prescribed parallelization technique, these mutexes need to restrict access to the `std::vector<result4_t>` that is used to store the classifications of all states currently being worked on.

It is too fine-grained (i.e. leads to too much overhead) to have a lock for each `result4_t` in the `std::vector`, because there can be up to ≈6×10^9 `result4_t`'s in the `std::vector` for the biggest class of states (namely, the class of states on 5×5 Quixo with 8 tiles each of X's and O's).

However, at the same time, it is too coarse-grained (i.e. leads to too much waiting between threads) to simply have a single lock for the entire `std::vector`.

To achieve a balance between these two extremes, the number of locks to use per `std::vector<result4_t>` is implemented to be controllable by the command line parameter `-L`. Empirically, choosing this number to be about L = 1,000 results in the best performance.

2. *The second involves the granularity of each individual 'unit task' that each thread is allocated to execute sequentially (in this project's codebase, 'unit task' refers to OpenMP's explicitly defined `task` directive).*

   The largest sensible size of this 'unit task' is to partition the set of all states currently being worked on, by the number of threads, and then setting a 'unit task' to be the value iteration work that needs to be done on one of these partitions — i.e. every thread executes exactly one 'unit task'. This, however, can potentially lead to poor load balance.

   On the other extreme, the smallest sensible size of this 'unit task' is the value iteration work that needs to be done on a single state — i.e. every thread executes a very large number of 'unit tasks'. Even though load balancing is much better since OpenMP handles the scheduling of `task`'s to threads based on runtime execution speeds, this falls prey to the other problem of high task-management overhead by OpenMP, since many `task`'s are being created and destroyed during value iteration.

   Again, to achieve a balance between these two extremes, this project instead implements a partitioning of the set of all states currently being worked on, by B times of the number of threads, and sets a 'unit task' to be the value iteration work that needs to be done on one of these partitions — i.e. every thread executes ≈B number of 'unit tasks'. Empirically, choosing this number to be about B = 10 results in the best performance, and this is hard-coded as the `BLOCK_TASK_FACTOR` macro of the `OptComputer` class.

2.2.3 Tanaka et al.'s Derivation of the Optimal Agent's Strategy

After obtaining the classification of all possible board states of Quixo into win, lose, or draw states, a straightforward optimal strategy described by Tanaka et al. is as follows:

- From a win state, choose any move that leads to a lose child state, uniformly at random;
- From a draw state, choose any move that leads to a draw child state, uniformly at random; and
- From a lose state, choose any move (they all lead to a win child state), uniformly at random.

However, Tanaka et al. then point out that an intuitively 'more optimal' strategy is for the optimal agent to play to win in the minimum number of moves possible, or to play to lose in the maximum number of moves possible (as opposed to choosing moves uniformly at random among all 'acceptable' moves).

This requires a slight modification to (and a re-execution of) the backward induction algorithm, to increase the specificity of the classification of all possible states, to also include a number of steps — which is the number of moves to win (or respectively lose) for any win state (or respectively any lose state). Specifically, this number of steps is:

- 0 for terminal states;
- 1 plus the minimum number of steps among lose children states, for a win state; and
- 1 plus the maximum number of steps among win children states, for a lose state.

2.2.4 This Project's Derivation of the Optimal Agent's Strategy

Following Tanaka et al.'s paper, this project's codebase implements both the straightforward optimal strategy, as well as the number-of-steps-aware 'more optimal' version of the strategy. These can be invoked as the player `opt`, and the player `opt-steps`, respectively (these are possible arguments for the command line parameters `-X` and `-O`, which set the strategies of players X and O respectively). These are implemented in the `OptimalPlayer` subclass of the generic `Player` class.

It is worth noting that the `opt-steps` strategy is only computed for 2×2 through 4×4 Quixo, even though the `opt` strategy is computed for 2×2 through 5×5 Quixo. This is because the memory usage requirement of storing a number of steps for each possible state in 5×5 Quixo, assuming that this number of steps is stored as a 8-bit unsigned integer, using a similar calculation as in Section 2.2.2.1, amounts to $3^{25} \times 8$ bits $\approx$ 800 GB, which is unfeasible for the commodity hardware used for this project.

2.3 Monte Carlo Tree Search (MCTS) Agent

MCTS is a well-known heuristic search algorithm that can be used to implement computational learning agents for games. Various readily-available literature covers the algorithm in more detail, including Browne et al.'s paper (2012).

The basic premise of this algorithm (as it is implemented in the `MCTSPlayer` subclass of the generic `Player` class in this project's codebase) is as follows. Given a particular board state of Quixo (i.e. for each invocation of `selectMove()`), the MCTS agent runs *n* iterations of simulated games, starting from the given state, exploring and learning about the game tree in the process of the simulations, and then, after all iterations have been completed, chooses the best move based on the learnt game tree. The number of iterations of simulated games, *n*, is a user-specified parameter.

Specifically, each iteration of a simulated game — `runIter()` — involves four steps:

1. *Selection (making moves within the explored part of the game tree).*

   If the current state is a terminal state, jump directly to the back-propagation step. Otherwise, consider all the children states of the current state.

If all of these children states have been explored at least once before, then select the child state that seems the most promising. This child state replaces the current state (i.e. the algorithm 'moves to' the child state), and this step is recursively invoked.

Otherwise, if at least one of these children states have not yet been explored, jump to the expansion step.

The selection of the child state that seems the most promising is done via choosing the child state i that maximizes the value of the following Upper Confidence Bound (UCB) formula, which balances exploitation and exploration in the game tree:

$$\left(1 - \frac{w_i}{n_i}\right) + \sqrt{2\left(\frac{ln(N)}{n_i}\right)}$$

where w_i is the number of past iterations that have passed through state i and ended in a win, n_i is the number of past iterations that have passed through state i, and N is the number of past iterations that have passed through the current state. It is worth noting that the $(1 - \bullet)$ part of the formula is necessary, since the win-rate from the child state is a negative reward for the current state (due to the swap in the active player between moves).

2. *Expansion (choosing to add one new explored state to the game tree)*.

   Uniformly randomly, choose one of the children states that have not yet been explored, to be expanded, 'move to' that child state, and jump to the playout step.

3. *Playout (making moves from the new explored state, into unexplored parts of the game tree)*.

   Again, if the current state is a terminal state, jump directly to the back-propagation step. Otherwise, make uniformly random moves and recursively invoke this step (i.e. make random moves until a terminal state is reached).

4. *Back-propagation (updating the game tree with knowledge from this iteration)*.

   Upon reaching a terminal state, check which player won, and update cached values of w_i and n_i for all states i that were passed through, from the initial given state to the current terminal state, swapping the active player between moves accordingly.

Finally, to fully specify this project's implemented MCTS algorithm, the best move selection at the end of the algorithm is done as follows. The MCTS agent chooses the move that leads to the child state i that minimizes the value of $\frac{w_i}{n_i}$ (again, using a minimization instead of a maximization due to the swap in the active player between moves).

This MCTS agent can be invoked as the player `mcts*` (again, as a possible argument for the command line parameters `-X` and/or `-O`), where the `*` specifies the number of iterations *n* that the MCTS agent runs for, upon each invocation of `selectMove()`.

2.4 Q-learning Agent (with Feature Approximators)

Q-learning is another well-known reinforcement learning algorithm that can be used to implement computational learning agents for games. Similarly, various readily-available literature covers the algorithm in more detail, including Watkins & Dayan's paper (1992).

The basic premise of this algorithm (as it is implemented in the `QLearningPlayer` subclass of the generic `Player` class in this project's codebase) is as follows. Prior to receiving any particular board state of Quixo (i.e. prior to any invocation of `selectMove()`), the Q-learning agent's initialization routine involves running *n* iterations of simulated games, each starting from a different uniformly randomly chosen non-terminal state. The key difference between this Q-learning agent, compared to the MCTS agent described earlier, is that the Q-learning agent does not learn about the specific game tree during these simulated games — rather, it attempts to generalize about the quality of each move in relation to the presence of various features of states. Then, after this one-time initialization for a game, upon receiving any particular state during that game (i.e. for each invocation of `selectMove()` during that game, which is invoked for every move in the game), the Q-learning agent then chooses the best move based on this learnt relation between the quality of moves and features. Like the MCTS agent, the number of iterations of initialization simulated games, *n*, is a user-specified parameter.

Formally, this notion of the quality of moves can be interpreted as a function from a given state and a given move to a quality value:

$$Q : S \times A \rightarrow \mathbb{R}$$

where S denotes the set of all possible states, A denotes the set of all possible moves, and the quality value is a real number. Denoting the function to obtain a vector of feature weights from a given state $s \in S$ by $f(\cdot)$ (this is implemented as `getFeatures()` in this project's codebase), this function Q can be expressed as:

$$\begin{bmatrix} Q(s,a_1) \\ Q(s,a_2) \\ \vdots \\ Q(s,a_k) \end{bmatrix} = Wf(s)$$

where $A = \{a_1, a_2, ..., a_k\}$ is the set of all possible moves, and W is a learnt weight matrix that is matrix-multiplied with the vector $f(s)$.

Specifically, in this project's implementation, W is initialized to be an all-zero matrix perturbed randomly by some small noise value ($\approx 10^{-6}$). Then, each iteration of a simulated game — `runIter()` — involves:

1. *Playout (making epsilon-greedy moves from the current state, until a terminal state)*.

   From the current (non-terminal) state, repeatedly make moves according to the epsilon-greedy strategy that (usually) maximizes the quality value of the move chosen, until a terminal state is reached.

For this project's implementation, based on preliminary empirical trials, epsilon is chosen to be 0.01.

2. *Back-propagation (updating W with knowledge from this iteration)*.

   Upon reaching a terminal state, check which player won, and conduct a value iteration update of W for each pair of parent-child states passed through (denoted respectively by s_0 and s_1, with the move taken between them denoted by a), backward from the last pair to the first. Specifically, this applies the following update:

$$W_a \leftarrow W_a + \alpha \times \left\{ \gamma \left(- max_a \left[(Wf(s_1))_a \right] \right) - (Wf(s_0))_a \right\} \times f(s_0)$$

   where W_a is the row (vector) of the matrix corresponding to action a, and analogously $(Wf(s_0))_a$ and $(Wf(s_1))_a$ are action a's entries of the quality value vectors $Wf(s_0)$ and $Wf(s_1)$ of states s_0 and s_1, respectively — i.e. $(Wf(s_0))_a = Q(s_0, a)$ and $(Wf(s_1))_a = Q(s_1, a)$. Hence, $max_a \left[(Wf(s_1))_a \right]$ is an estimate of the best quality value for the next player who is choosing a move a from state s_1 (this estimate is replaced by ± 1 when s_1 is a terminal state, with the sign depending on which player won). Then, $- max_a \left[(Wf(s_1))_a \right]$ is an estimate of the future quality value for the current player (with the negation again due to the swap in the active player between moves). The last three weighting factors in this equation are the learning rate α, the temporal discounting rate γ, and the vector of feature weights of the parent state $f(s_0)$.

   For this project's implementation, again based on preliminary empirical trials, α is chosen to start at 0.5 and to decrease by 0.05% after every iteration, while γ is chosen to be 0.99999.

This Q-learning agent can be invoked as the player `q-learn*` (again, as a possible argument for the command line parameters `-X` and/or `-O`), where the `*` specifies the number of initialization iterations *n* that the Q-learning agent runs for.

2.4.1 Specific Features Chosen for This Project's Q-learning Agent

For this project, the vector of feature weights of a given board state of Quixo (implemented in `getFeatures()`) consists of the following feature weights:

- A constant bias term (to weight certain moves as having higher quality values than other moves, regardless of the state that it is played from).
- The total number of X and O tiles on the board (i.e. two separate features), each normalized by the total number of tiles on the board.
- The total number of X and O tiles on each row, column and diagonal of the board (again, with separate features for X and O tiles), each normalized by the total number of tiles on one side of the board.

which cover most — if not all — of the easily humanly-identifiable features on a Quixo board. This totals to $4l + 7$ feature weights for each board state, where l is the number of tiles on one side of the board.

3 Results

3.1 Computation Time for the Optimal Solution's Classification of Board States

It is worth noting, for a sense of scale, the computation time for the brute-force backward induction algorithm, as described above in Section 2.2.2.

The full execution of this algorithm for 5×5 Quixo, even with all of the optimizations described previously, and even running on 16 threads, still takes approximately 2 weeks to complete on one node of Yale's Zoo machines. For a sense of the algorithm's runtime asymptotics, the corresponding execution times for 4×4 Quixo and 3×3 Quixo are approximately 20 seconds and less than 1 second respectively.

For comparison, Tanaka et al.'s implementation took approximately 2 weeks to complete with 1 thread, and approximately 32 hours to complete with 32 threads, using *"an Ubuntu 18.04LTS server equipped with 32GB of RAM and powered by a 16-core Intel Core i9-9960X CPU"*. This indicates that there might be some room for improvement from an efficiency standpoint in this project's codebase, but the current implementation suffices for the purposes of this project.

3.2 Basic Analysis & Verification of the Optimal Solution's Classification of Board States

Before proceeding with experiments for producing the strategy ladders, it is worthwhile to compare some aggregate statistics of the optimal solution's classification of board states, against Tanaka et al.'s results, to lend some credibility to the correctness of this project's implementation (and to verify Tanaka et al.'s results).

3.2.1 Main Result for 5×5 Quixo (and the Corresponding Results for 4×4 & 3×3 Quixo)

Tanaka et al. first note that their main result is that 5×5 Quixo is a draw game, i.e. the board state with all empty tiles is classified as a draw state. For comparison, they also note that 4×4 Quixo and 3×3 Quixo are first-player-win games.

This project's implementation yields all three of the same results, by using the following commands (hitting Enter after each command is run to confirm the initial board state selection):

```
$ ./bin/quixo -p opt-check -l 5 -g 800
$ ./bin/quixo -p opt-check -l 4 -g 800
$ ./bin/quixo -p opt-check -l 3 -g 800
```

3.2.2 Aggregate Number of Win, Lose, or Draw States for 5×5 Quixo

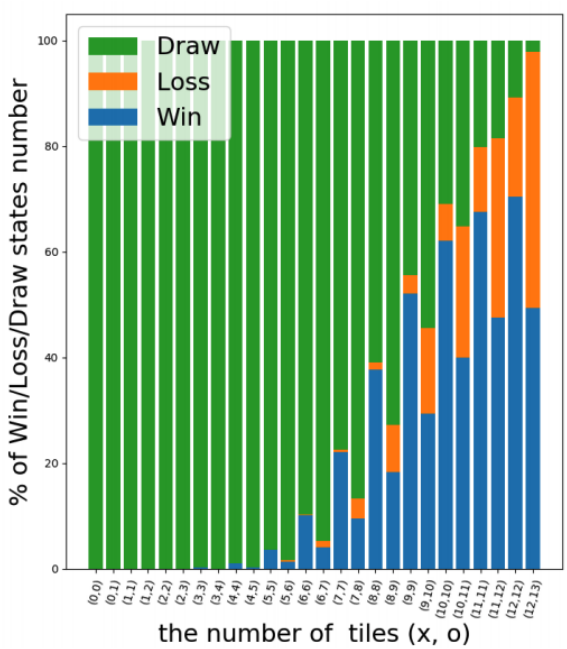
Tanaka et al. next present their aggregate number of win, lose, or draw states for 5×5 Quixo as:

- 441,815,157,309 win states;
- 279,746,227,956 lose states; and
- 125,727,224,178 draw states.

which this project's implementation verifies, by using the following command:

```
$ ./bin/quixo -p opt-analyze -l 5
```

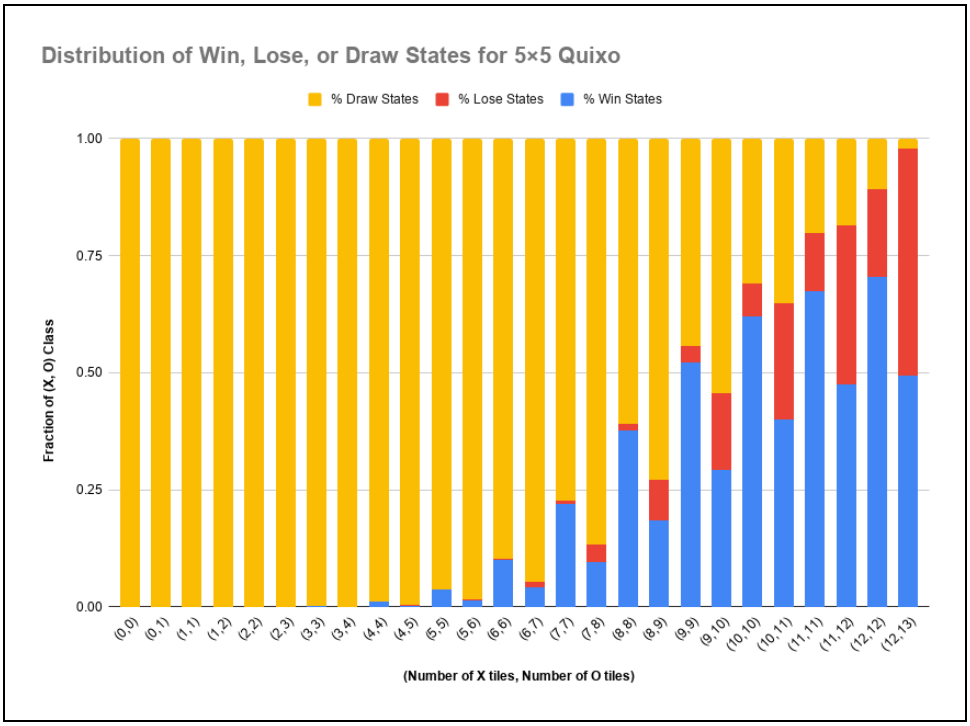
Tanaka et al. further present the following graphical breakdown of the fraction of the number of win, lose, or draw states for 5×5 Quixo for some selected classes of states:



This project's implementation, by using the following command:

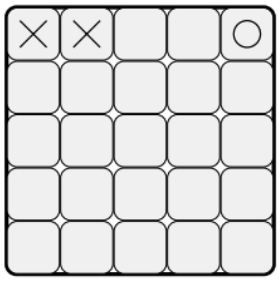
```
$ ./bin/quixo -p opt-analyze-adj -l 5
```

also visually verifies this by producing the following graph:

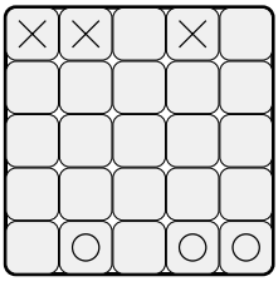


3.2.3 Specific Win, Lose, or Draw States for 5×5 Quixo

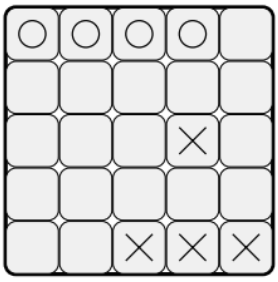
Tanaka et al. next present several specific win, lose, or draw states for 5×5 Quixo as:



(a) Player X can win. This state is one of those with the smallest numbers of tiles such that the outcome is not draw.



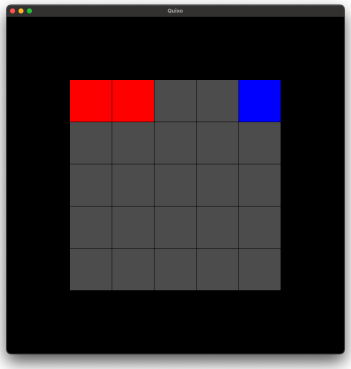
(b) Player X can win. This state is one of those with the smallest numbers of tiles such that the outcome is not draw and both players have chosen empty tiles only.

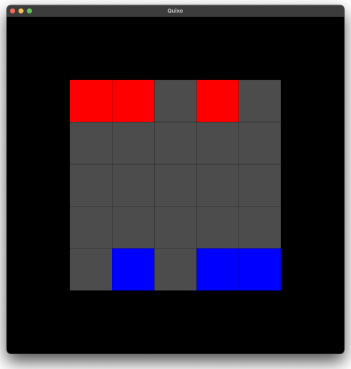


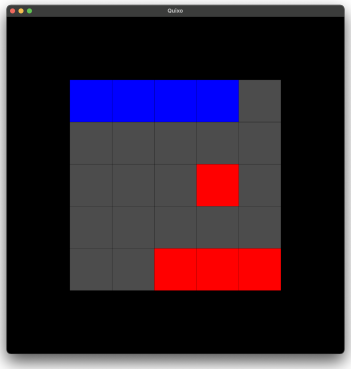
(c) Player X loses. The number of Xs and Os tiles is the same but the active player loses.

which this project's implementation verifies by using the following command (with appropriate clicking to set the board state after each time the command is run):

```
$ ./bin/quixo -p opt-check -l 5 -g 800
```

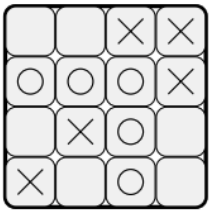




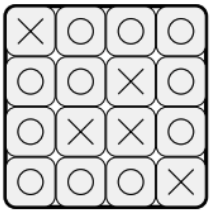


3.2.4 Specific Win, Lose, or Draw States (with Number of Steps) for 4×4 Quixo

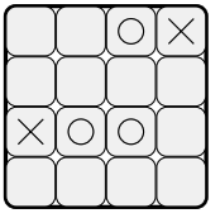
Tanaka et al. next present several specific win, lose, or draw states for 4×4 Quixo as:



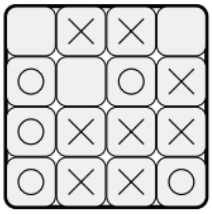
(a) Unreachable state. No previous state.



(b) Player X loses in 1 step.



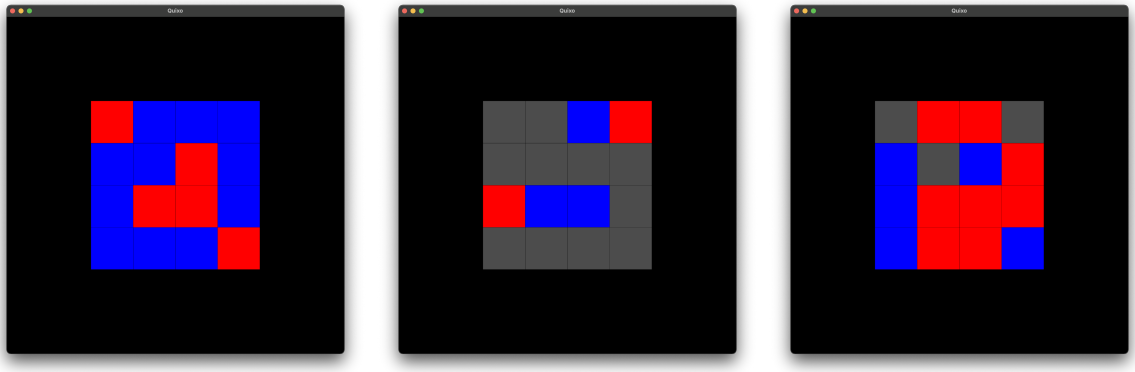
(c) Player X loses in 22 steps.



(d) Draw state. O can come back to this state (or a symmetric one) with the next O step even if X plays optimally.

which this project's implementation verifies, including the number of steps, by using the following command (again with appropriate clicking to set the board state after each time the command is run):

```
$ ./bin/quixo -p opt-check -l 4 -g 800
```



3.2.5 Aggregate Number of Win, Lose, or Draw States (with Number of Steps) for 4×4 Quixo

Tanaka et al. finally present their aggregate number of win, lose, or draw states, with number of steps, for 4×4 Quixo as:

steps	Win states	Loss states	steps	Win states	Loss states
0	4,697,505	4,530,779	12	0	182,954
1	15,277,446	528	13	100,374	0
2	0	3,775,611	14	0	66,280
3	2,419,938	0	15	29,314	0
4	0	2,970,384	16	0	18,014
5	1,740,992	0	17	6,656	0
6	0	1,982,339	18	0	4,084
7	1,214,497	0	19	1,012	0
8	0	1,034,097	20	0	520
9	658,834	0	21	57	0
10	0	438,138	22	0	8
11	287,864	0	23	0	0
			total	26,434,489	15,003,736

which this project's implementation verifies, including the number of steps, by using the following command:

```
$ ./bin/quixo -p opt-analyze-steps -l 4
```

3.3 MCTS Agent Strategy Ladder: Quantifying Solution Strength via (1) Game Outcomes

Circling back to this project's primary goal of investigating the 'depth' of Quixo via the strategy ladder model proposed by Lantz et al., first, several strategy ladder graphs for the MCTS agent will be presented, before moving on to the strategy ladder graphs for the Q-learning agent.

Even with the exact optimal solution of Quixo computed, there are still several ways to quantify the performance of the MCTS agent, in comparison to this exact optimal solution, to form the *y*-axis of the strategy ladder graph (i.e. "Solution Strength").

3.3.1 Experimental Setup

The first and most intuitive way to quantify the performance of the MCTS agent is via game outcomes. Specifically, this experiment involves playing the MCTS agent against the optimal agent 200 times in total (100 times with the MCTS agent as player X, and 100 times with the MCTS agent as player O). Since Quixo is not a finite game, as noted earlier, a turn limit of 200 turns is imposed for each game, and the game is concluded as a draw if the turn limit is reached without either player winning (where a turn is defined as one move by each player, analogous to chess). The 'strength score' of the MCTS agent for each experimental run is then defined as:

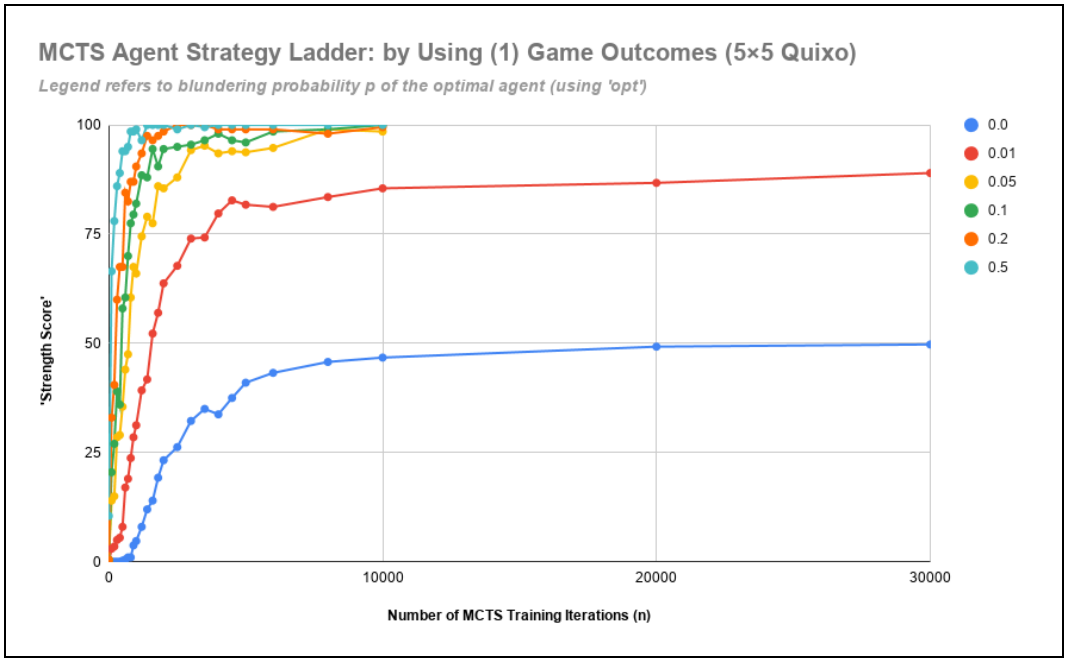
$$\frac{1}{2}\left(w + \frac{d}{2}\right)$$

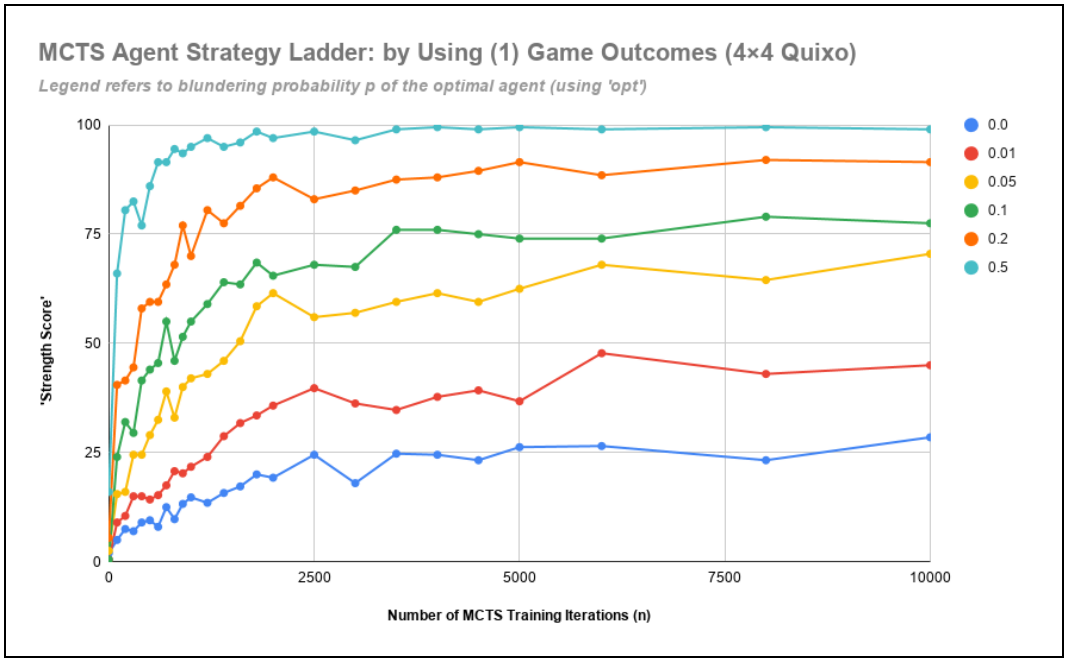
where w is the total number of wins by the MCTS agent and d is the total number of draws. Note that this normalizes the 'strength score' to be out of a 100 in total, and that a 'strength score' of 50 means that the MCTS agent is performing equally well compared to the optimal agent.

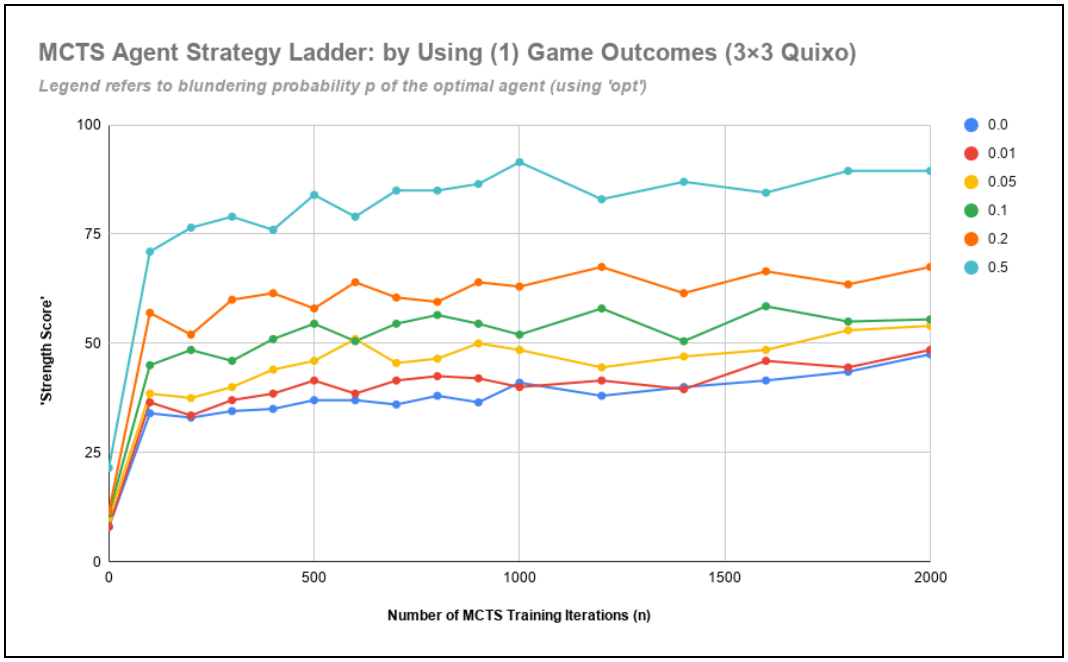
Crucially, because 5×5 Quixo is a draw game, the MCTS agent can never win against the optimal agent (whether playing as player X or player O) — at best, the MCTS agent can force a draw. Therefore, for a more illuminating strategy ladder, variants of this experiment were conducted in which the optimal agent occasionally 'blunders' by playing a uniformly random move (instead of the optimal move), with some fixed probability *p* of blundering for each experimental run.

3.3.2 Results Quantified by Number of Training Iterations, Individually for 5×5, 4×4, and 3×3 Quixo

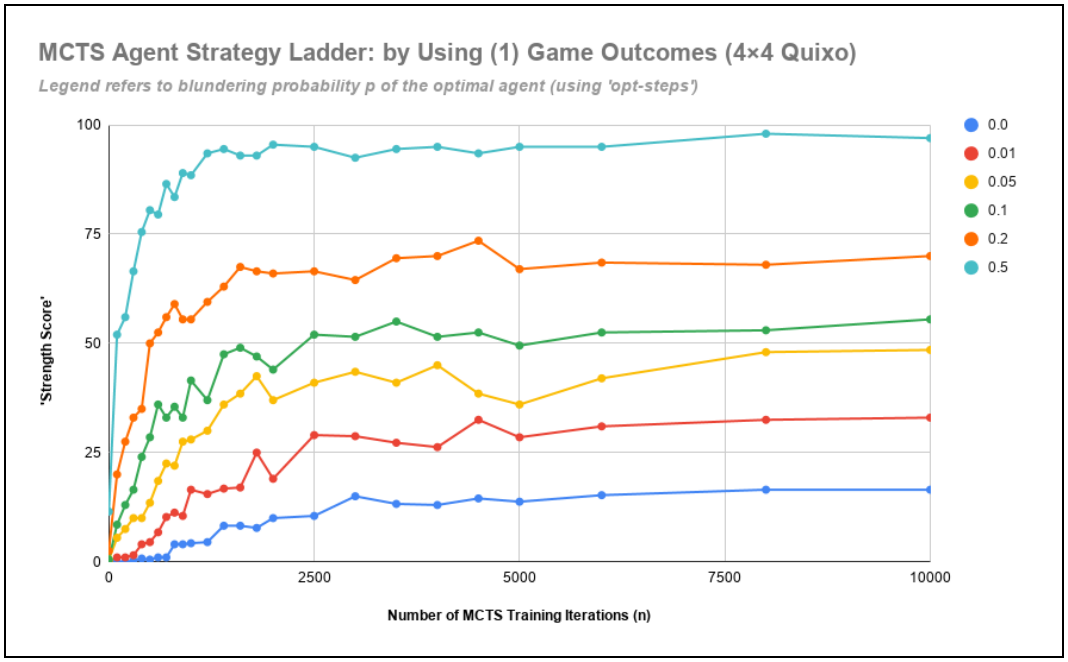
Using this quantification of the performance of the MCTS agent, the strategy ladder graphs when playing against the optimal agent using the straightforward optimal strategy (i.e. `opt`), for 5×5, 4×4, and 3×3 Quixo, are as follows:

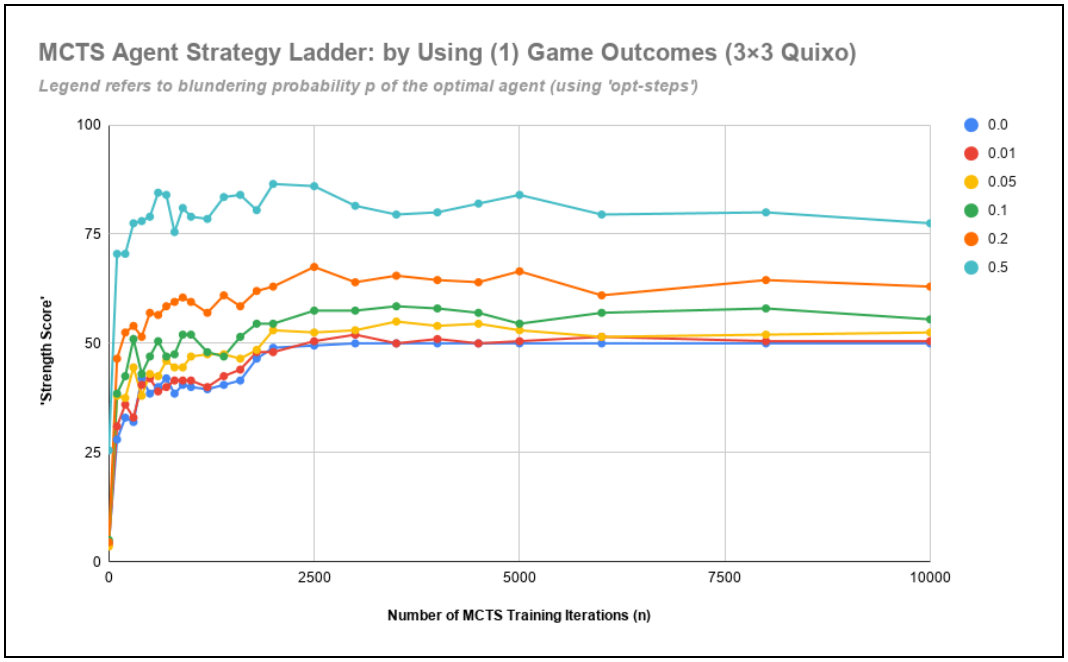






while the corresponding strategy ladder graphs when playing against the optimal agent using the number-of-steps-aware version of the optimal strategy (i.e. `opt-steps`), for 4×4 and 3×3 Quixo, are as follows:





From all of these graphs, it is possible to observe the 'skill chain' of the MCTS agent learning to play the game better with an increasing number of training iterations, and eventually reaching a horizontal asymptote (i.e. of solution strength), for each of the above strategy ladder graphs.

In addition, these graphs fulfill the basic sanity check that the MCTS agent attains increasing values of its 'strength score' as the optimal agent blunders more often (i.e. with increasing p).

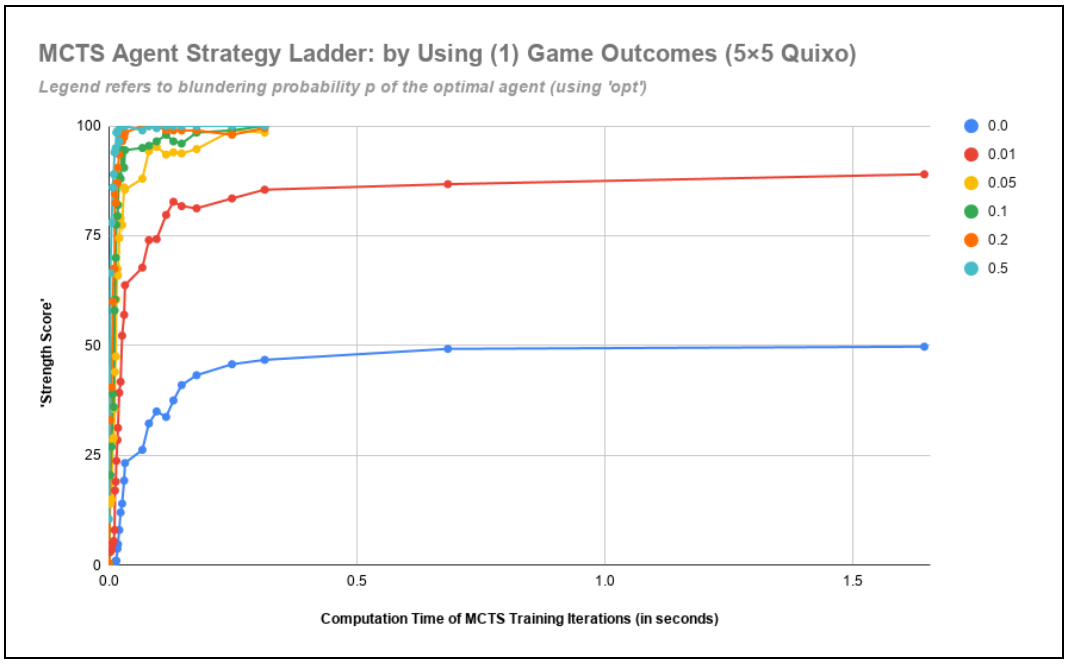
It is worth mentioning that the noise in these graphs (i.e. the kinks) reveal that 200 games per data point might not be a large enough sample size to smooth out statistical random errors; nevertheless,

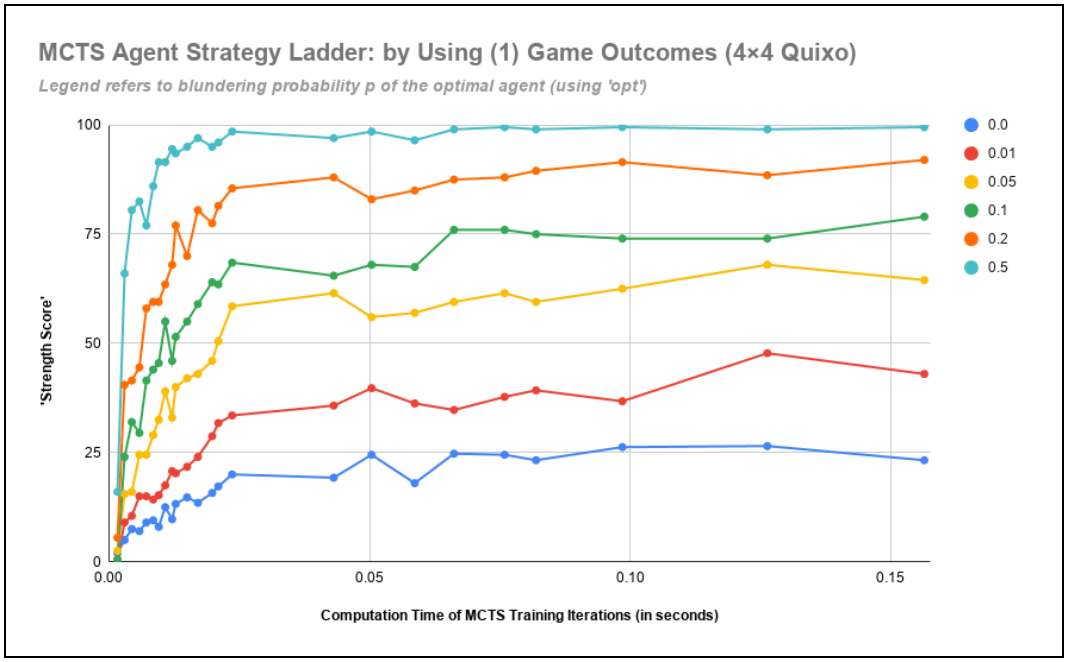
considering that the experiments for all the data points on these five graphs, in total, already took about 5 days to run to completion (using a single thread), the number of games per data point was not increased any further — especially since revealing the shape of the graph suffices for the purposes of this project.

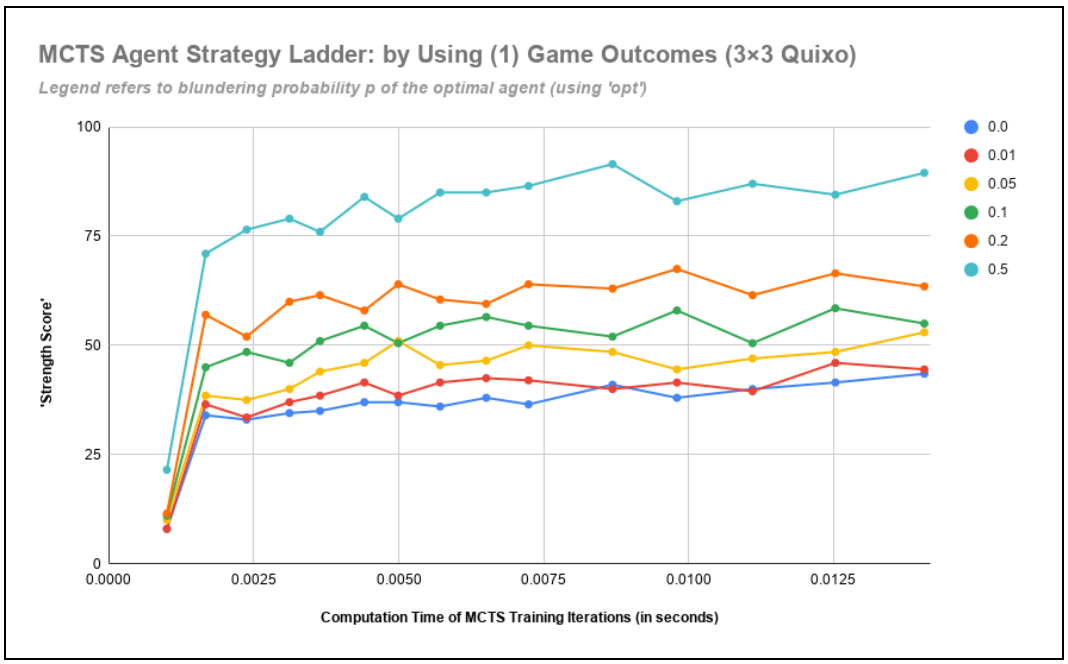
3.3.3 Results Quantified by Computation Time, Individually for 5×5, 4×4, and 3×3 Quixo

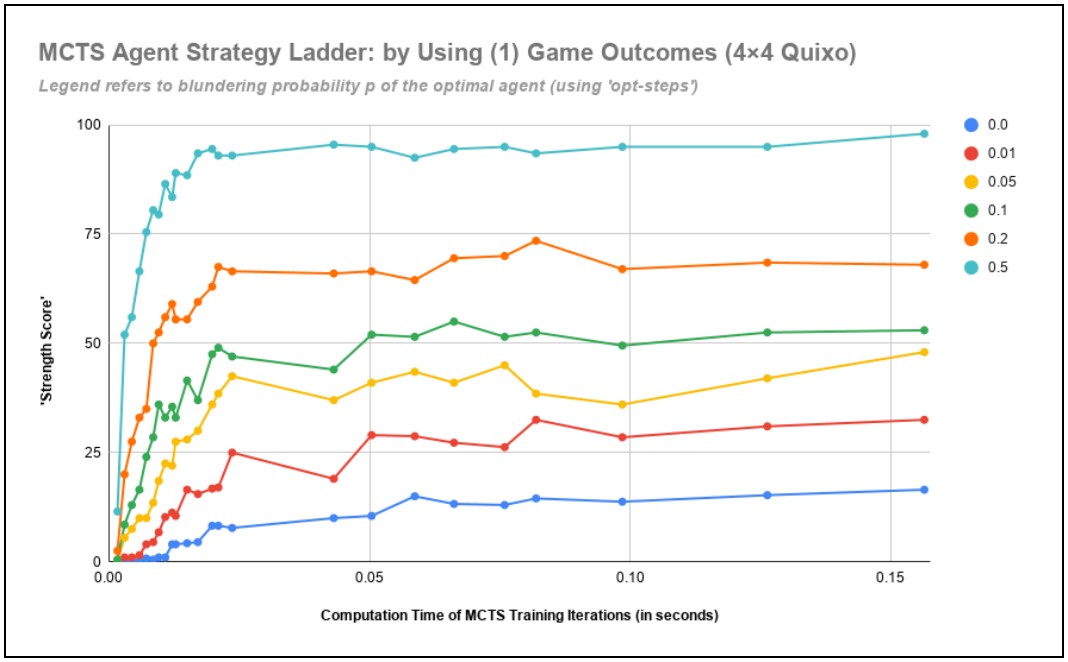
Note, however, that these *x*-axis quantifications of the number of MCTS training iterations do not directly reflect the originally intended *x*-axis of "Computational Resources" — i.e. the amount of time given to the MCTS agent for learning the game — since a single training iteration on 5×5 Quixo takes longer than a single training iteration on 3×3 Quixo (simply because each simulated game lasts for more turns).

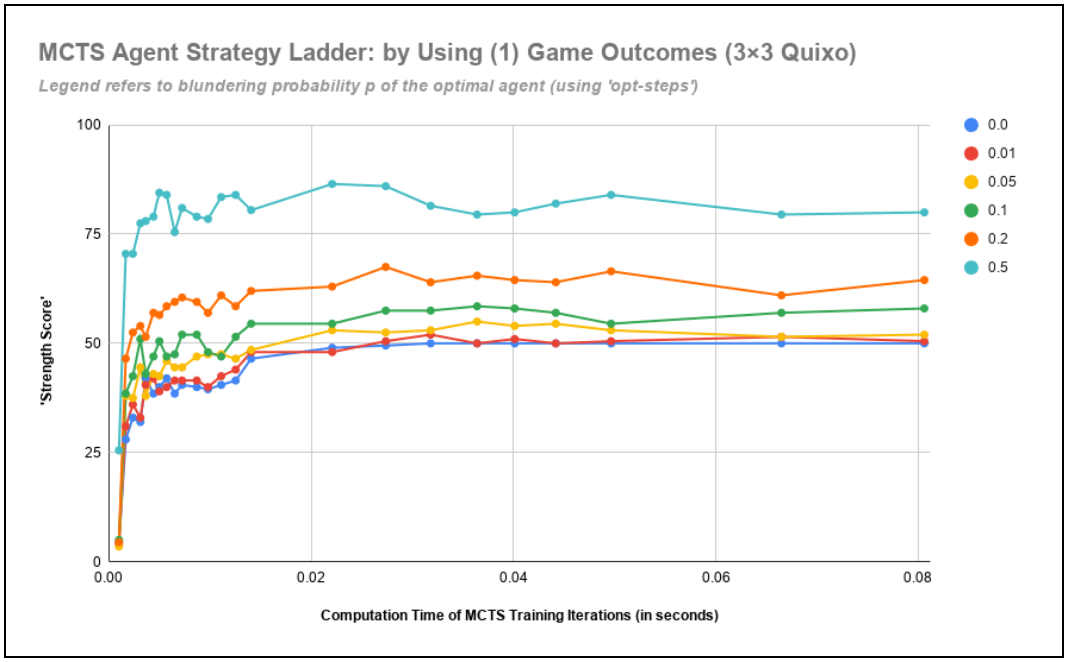
Correcting for this, the following five strategy ladder graphs (in the same order) are obtained instead (note the difference in the *x*-axis):




MCTS Agent Strategy Ladder: by Using (1) Game Outcomes (4×4 Quixo)
Legend refers to blundering probability p of the optimal agent (using 'opt')
'Strength Score'
100
75
50
25
0
0.00
0.05
0.10
0.15
Computation Time of MCTS Training Iterations (in seconds)
0.0
0.01
0.05
0.1
0.2
0.5


MCTS Agent Strategy Ladder: by Using (1) Game Outcomes (3×3 Quixo)
Legend refers to blundering probability p of the optimal agent (using 'opt')
'Strength Score'
100
75
50
25
0
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
Computation Time of MCTS Training Iterations (in seconds)
0.0
0.01
0.05
0.1
0.2
0.5



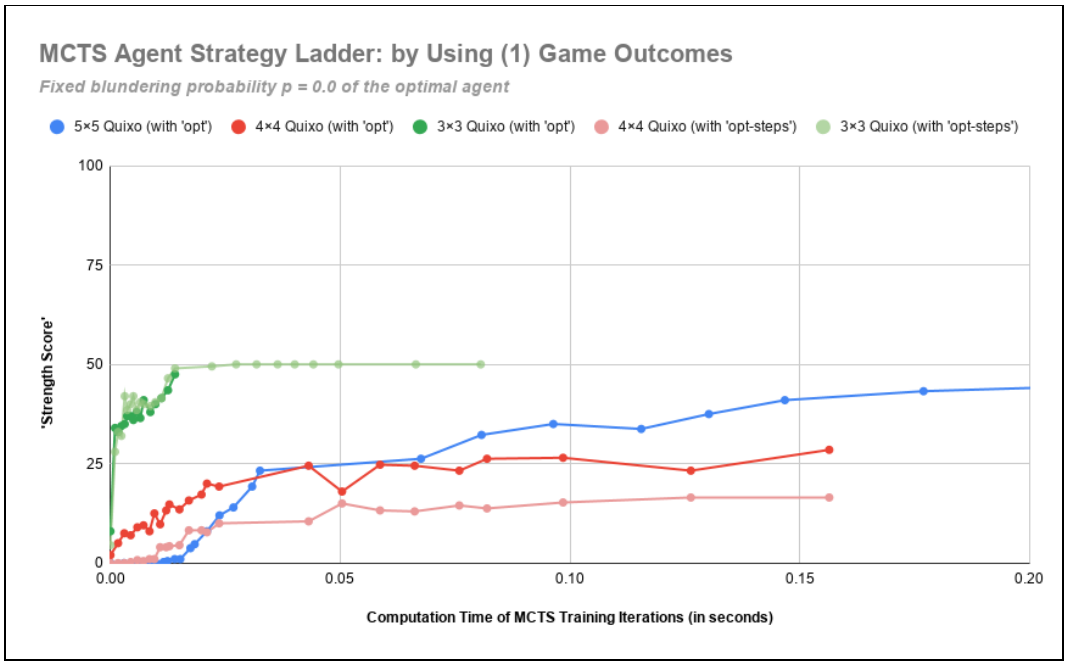


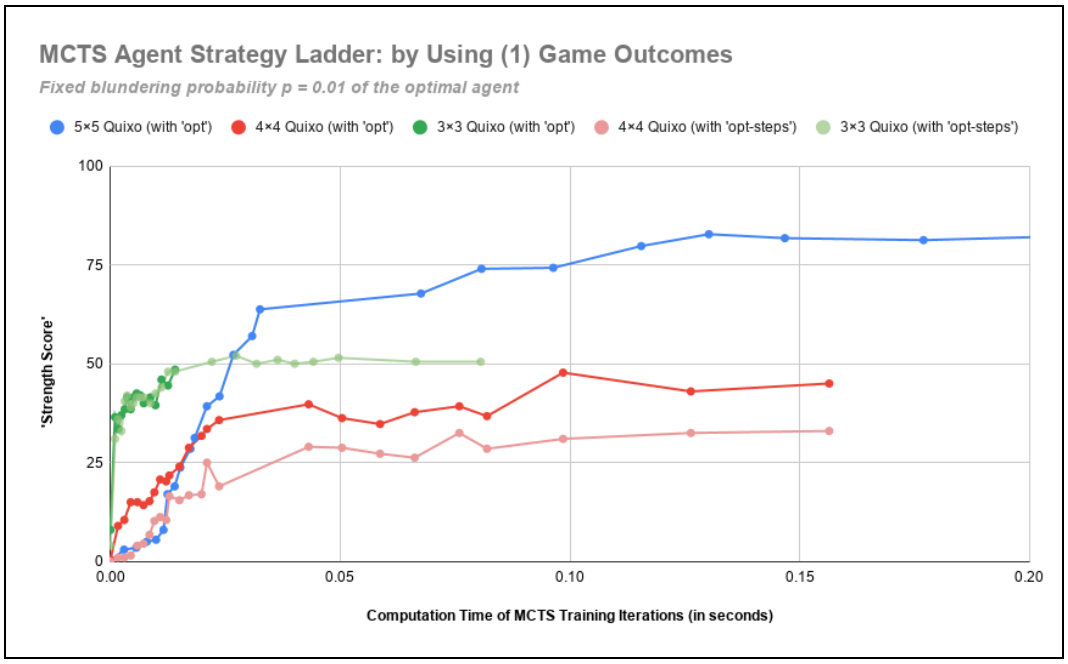
These graphs qualitatively look the same as the previous graphs. This is expected, since the computation time for training iterations is roughly linear in the number of training iterations, for each fixed size of Quixo — each training iteration takes approximately the same amount of time, for each fixed size of Quixo.

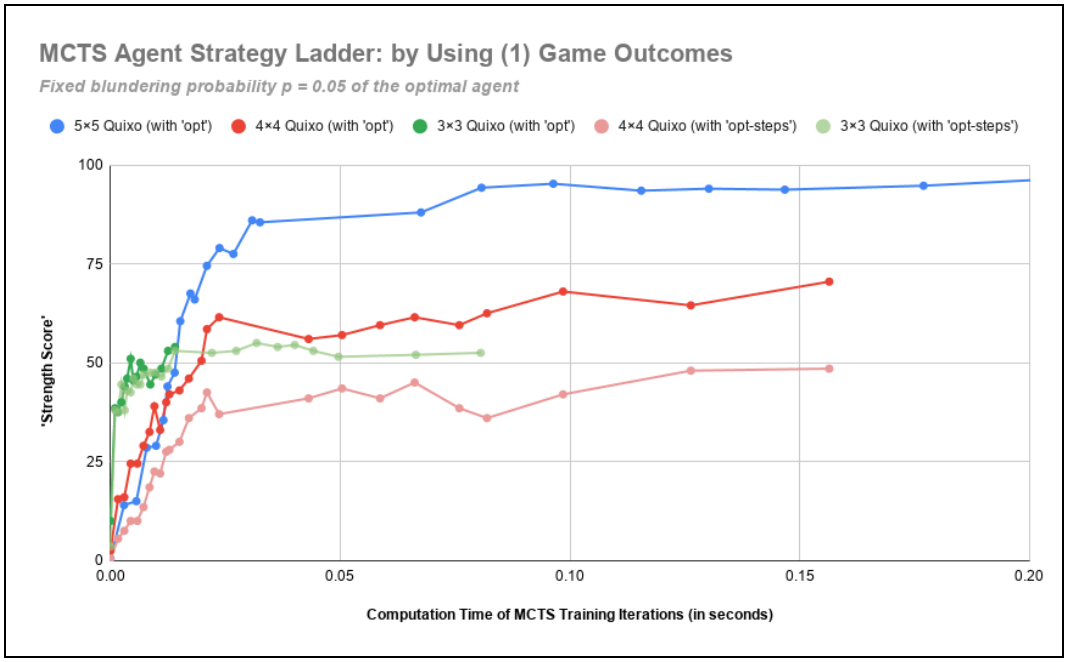
3.3.4 Results Quantified by Computation Time, Comparing Across 5×5, 4×4, and 3×3 Quixo

However, by making this correction, the graphs for 5×5, 4×4, and 3×3 Quixo can now be overlaid and compared (alongside comparing between playing against `opt` and `opt-steps`). Picking and fixing

(arbitrarily) the first three representatives of blundering probability p = 0.0, 0.01, and 0.05, the following three comparative strategy ladder graphs can be obtained (one for each value of p):







Several features are noteworthy about these comparative strategy ladder graphs:

1. *A difference in 'depth' is revealed by this strategy ladder model, when comparing between 5×5, 4×4, and 3×3 Quixo.*

   For each of the three comparative strategy ladder graphs, visually, the MCTS agent reaches a horizontal asymptote the earliest (i.e. with the least amount of computation time) for 3×3 Quixo, followed by 4×4 Quixo, and then 5×5 Quixo. For example, for the $p = 0.0$ graph, the MCTS agent reaches a horizontal asymptote for 3×3 Quixo at about 0.02s (for both the `opt` and `opt-steps` variants), for 4×4 Quixo at about 0.04s (again for both the `opt` and `opt-steps` variants), and for 5×5 Quixo at about 0.018s.

2. *The basic sanity check that `opt-steps` performs better than `opt` is satisfied.*

   By minimizing the number of moves to win, the number-of-steps-aware optimal agent makes less blunders (since this occurs with a fixed probability per move), and thus performs better — leading to the MCTS agent's 'strength score' being lower when playing against `opt-steps` than when playing against `opt`. This difference is more pronounced for 4×4 Quixo than for 3×3 Quixo, again because game lengths are generally longer for 4×4 Quixo than for 3×3 Quixo.

3. *That being said, the measurement of 'depth' via this strategy ladder model does not seem to be affected by the choice of using `opt` or `opt-steps` as the optimal agent.*

   Visually, the *x*-axis location at which the MCTS agent reaches a horizontal asymptote does not change whether it is playing against `opt` or `opt-steps`, for both 4×4 and 3×3 Quixo.

3.4 MCTS Agent Strategy Ladder: Quantifying Solution Strength via (2) Move Accuracy

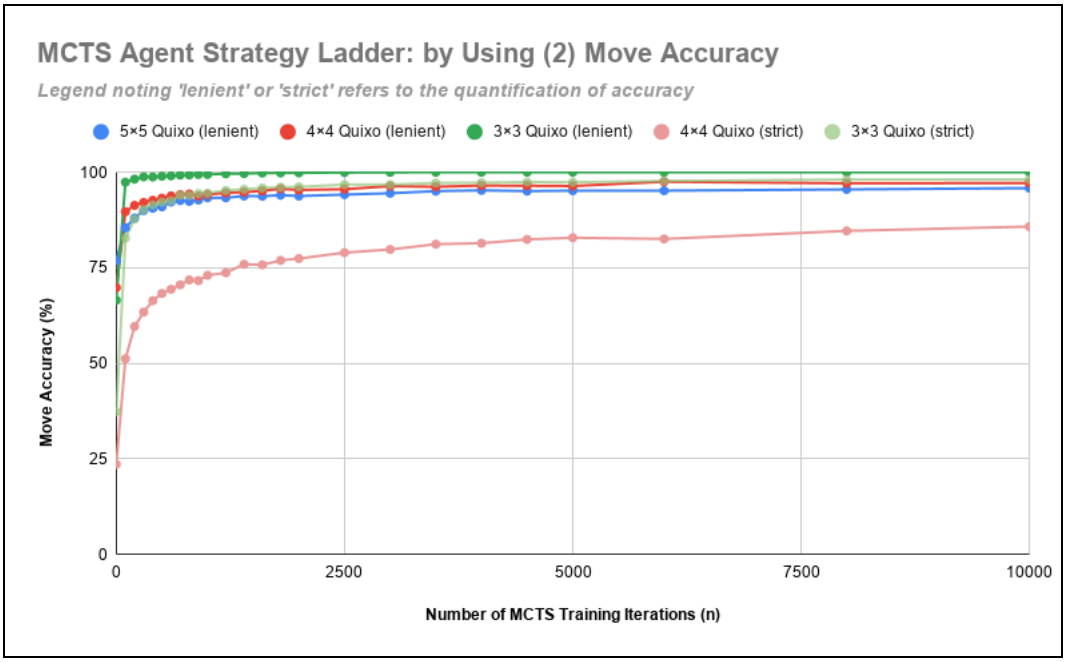
3.4.1 Experimental Setup

The next most intuitive way to quantify the performance of the MCTS agent is via the accuracy of its chosen moves on specific boards states. Specifically, this experiment involves giving the MCTS agent 10,000 particular board states (i.e. to its `selectMove()` function), where these states are uniformly randomly chosen from among all non-terminal states. Then, the accuracy of the MCTS agent's chosen move for that state can be evaluated based on the optimal solution's classification of all states. Two levels of quantifications of accuracy, corresponding respectively to the straightforward optimal strategy and the number-of-steps-aware version of the optimal strategy, are possible:

1. *A more lenient quantification of accuracy, corresponding to the optimal strategy of* `opt`*.*
    - If the chosen non-terminal state is a win state, then any move that leads to a lose child state is considered 'accurate';
    - If the chosen non-terminal state is a draw state, then any move that leads to a draw child state is considered 'accurate'; and
    - If the chosen non-terminal state is a lose state, then any move (at all) is considered 'accurate'.
2. *A stricter quantification of accuracy, corresponding to the optimal strategy of* `opt-steps`*.*
    - If the chosen non-terminal state is a win state, then only the moves that minimize the number of steps to win are considered 'accurate' (these moves necessarily also lead to a lose child state, by definition);
    - If the chosen non-terminal state is a draw state, then any move that leads to a draw child state is considered 'accurate' (i.e. no change from the lenient quantification of accuracy in this case); and
    - If the chosen non-terminal state is a lose state, then only the moves that maximize the number of steps to lose are considered 'accurate'.

3.4.2 Results Quantified by Number of Training Iterations, Comparing Across 5×5, 4×4, and 3×3 Quixo

Using this quantification of the performance of the MCTS agent, the strategy ladder graphs are as follows:



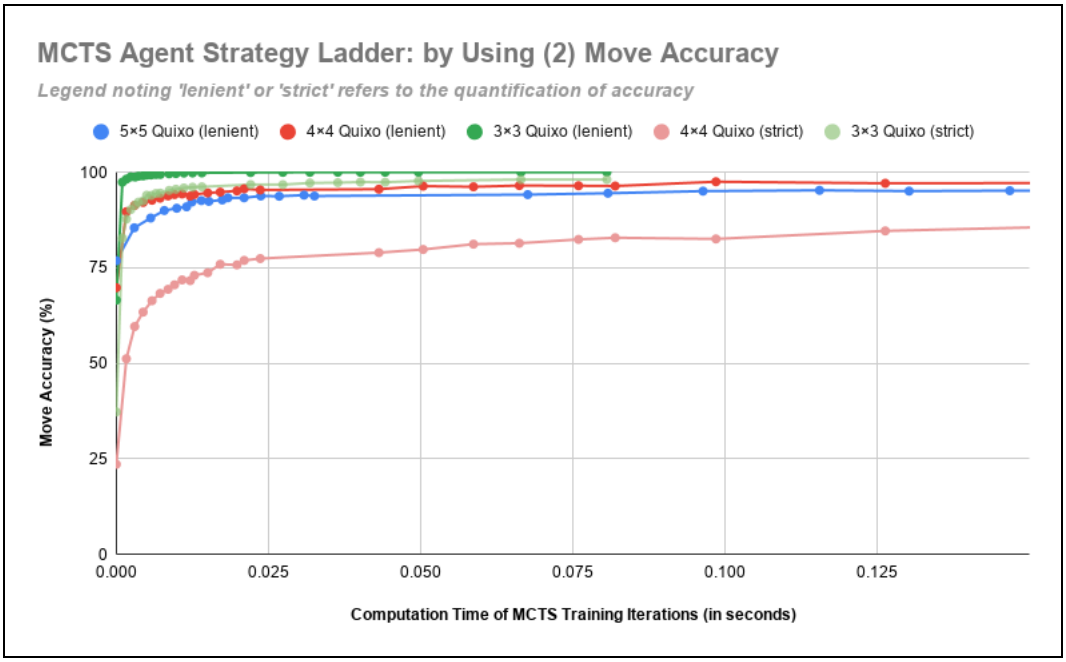
Similarly, this graph reflects the 'skill chain' of the MCTS agent learning to play the game better with an increasing number of training iterations, and eventually reaching a horizontal asymptote (at approximately the perfect player's move accuracy of 100%).

In addition, this graph fulfills the basic sanity check that the accuracy quantified by the more lenient measure is higher than the accuracy quantified by the stricter measure.

It is an interesting observation that these move accuracies are much higher than would be expected based on the results of Section 3.3. For example, at just 1,000 training iterations, the MCTS agent attains a 95% (strict) / 99% (lenient) move accuracy for 3×3 Quixo — which both seem reasonably high. But, in contrast, when playing against the `opt` optimal agent with blundering probability p = 0.01 (modelling the 99% move accuracy), or when playing against the `opt-steps` optimal agent with blundering probability p = 0.05 (modelling the 95% move accuracy), this very same MCTS agent (i.e. of 1,000 training iterations) is only able to attain a 'strength score' of 40 and 47 respectively. Recalling that a 'strength score' of 50 means that the MCTS agent is performing equally well compared to the optimal agent, this discrepancy can be made more stark by the following (intentionally somewhat loose) restatement of the phenomenon: *"even with individual move accuracies of up to 99%, the overall 'accuracy' of the MCTS agent based on final game outcomes is only approximately between 40/50 = 80% and 47/50 = 94%""*. This discrepancy could be indicative of Quixo being a 'mistake-punishing' game, whereby occasional single-move mistakes among mostly good moves, can still be exploited by a good opponent easily, and still end up leading to an overall game loss.

3.4.2 Results Quantified by Computation Time, Comparing Across 5×5, 4×4, and 3×3 Quixo

Next, again correcting the x-axis to use the computation time for training iterations instead of the number of training iterations (similar to Section 3.3.3), the corrected comparative strategy ladder graph is as follows:



Interestingly, from this comparative strategy ladder graph, there is less evidence of a difference in 'depth', when comparing between 5×5, 4×4, and 3×3 Quixo, since the MCTS agent reaches a horizontal asymptote at approximately the same amount of computation time for all of 5×5, 4×4, and 3×3 Quixo. It is possible that this quantification of the performance of the MCTS agent does not lend itself to revealing the notion of 'depth' as easily, as compared to the first quantification of performance.

3.5 MCTS Agent Strategy Ladder: Quantifying Solution Strength via (3) Game Length

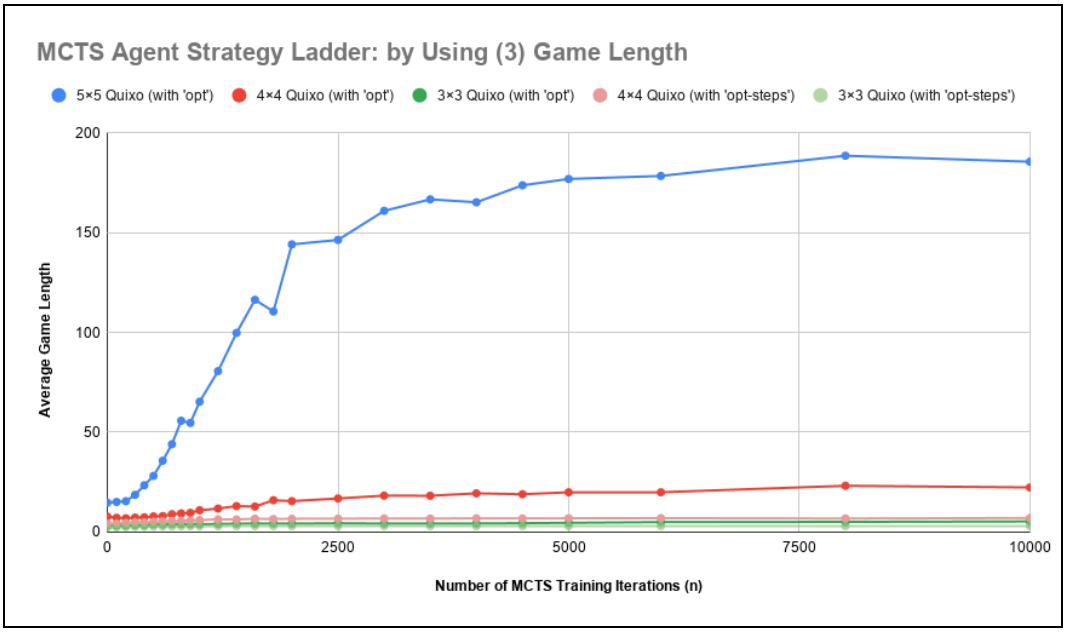
3.5.1 Experimental Setup

The last quantification of the performance of the MCTS agent, conducted for this project, is via game lengths. Specifically, this experiment involves playing the MCTS agent as player O, against the optimal agent as player X, 200 times in total, and computing the average game length of these games. Similar to Section 3.3, a turn limit of 200 turns is imposed for each game, and the game is concluded as a draw (with a game length of 200) if the turn limit is reached without either player winning.

Since 5×5 Quixo is a draw game, and 4×4 Quixo and 3×3 Quixo are both first-player-win games (i.e. winning for player X), notably, the MCTS agent can never win by playing as player O — at best, the MCTS agent can force a draw in the 5×5 Quixo case. Therefore, the number of turns that the MCTS agent is able to last without losing against the optimal agent, in these games, is another quantification of the performance of the MCTS agent (i.e. the longer the MCTS agent lasts, the better its performance).

3.5.2 Results Quantified by Number of Training Iterations, Comparing Across 5×5, 4×4, and 3×3 Quixo

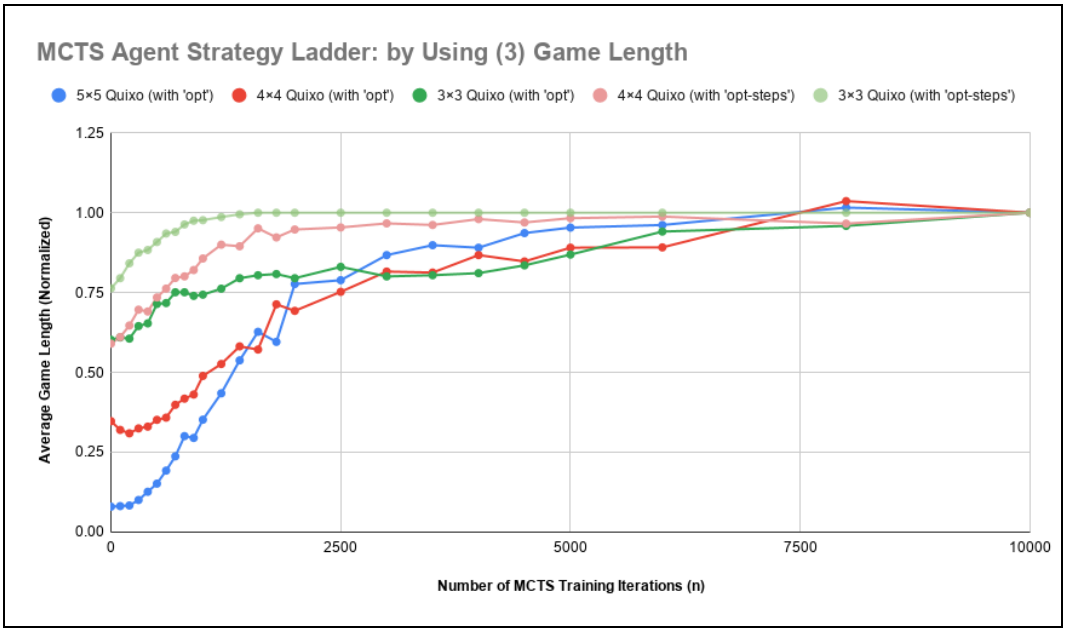
Using this quantification of the performance of the MCTS agent, the strategy ladder graphs are as follows:



Again, this graph reflects the 'skill chain' of the MCTS agent learning to play the game better with an increasing number of training iterations, and eventually reaching a horizontal asymptote.

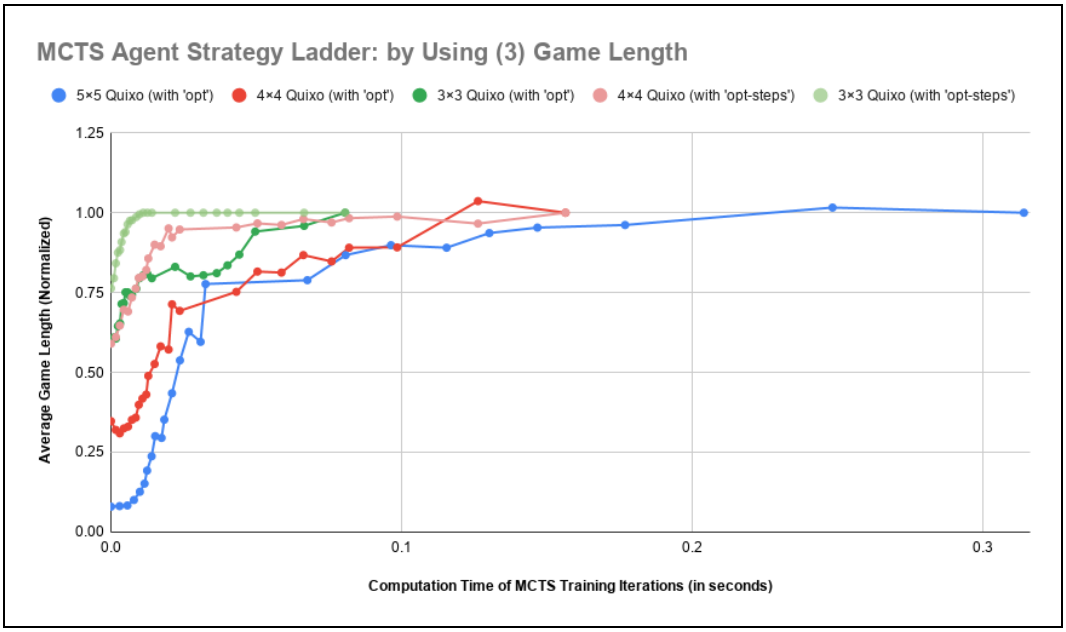
In addition, this graph fulfills the basic sanity check that the average game length when the MCTS agent plays against `opt` is longer than that when the MCTS agent plays against `opt-steps`, which is expected since the latter optimal agent actively aims to minimize the number of steps to win.

However, this graph also reveals that the average game length differs greatly between 5×5, 4×4, and 3×3 Quixo (which is expected). As such, for a more illuminating strategy ladder, each strategy ladder on the graph can be normalized (somewhat arbitrarily) by the average game length attained by the MCTS agent of 10,000 training iterations (i.e. the right endpoint of each strategy ladder will be normalized to a *y*-axis value of 1) — in particular, this is worthwhile doing because only the shape of the graph is important for this project's investigation of 'depth'. Applying this normalization yields:



3.5.2 Results Quantified by Computation Time, Comparing Across 5×5, 4×4, and 3×3 Quixo

Finally, again correcting the *x*-axis to use the computation time for training iterations instead of the number of training iterations (similar to Section 3.3.3), the corrected (and normalized) comparative strategy ladder graph is as follows:



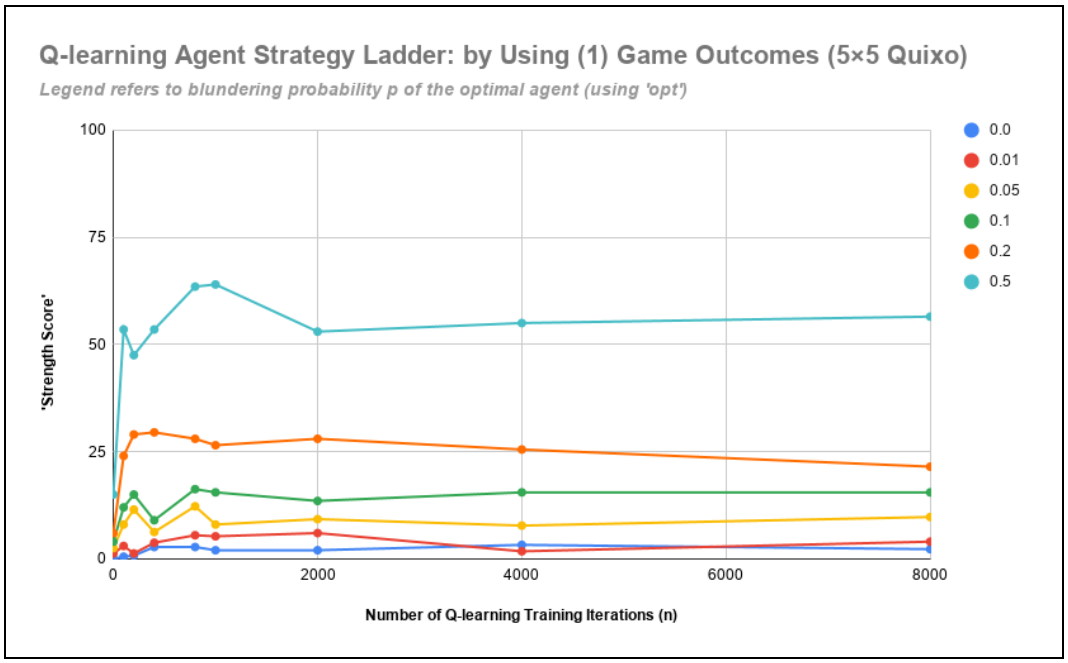
From this comparative strategy ladder graph, a difference in 'depth' is somewhat revealed, when comparing between 5×5, 4×4, and 3×3 Quixo. Visually, the MCTS agent seems to reach a horizontal asymptote the earliest for 3×3 Quixo, followed by 4×4 Quixo, and then 5×5 Quixo, similar to the comparative strategy ladder graphs for the first quantification of performance (i.e. in Section 3.3.4).

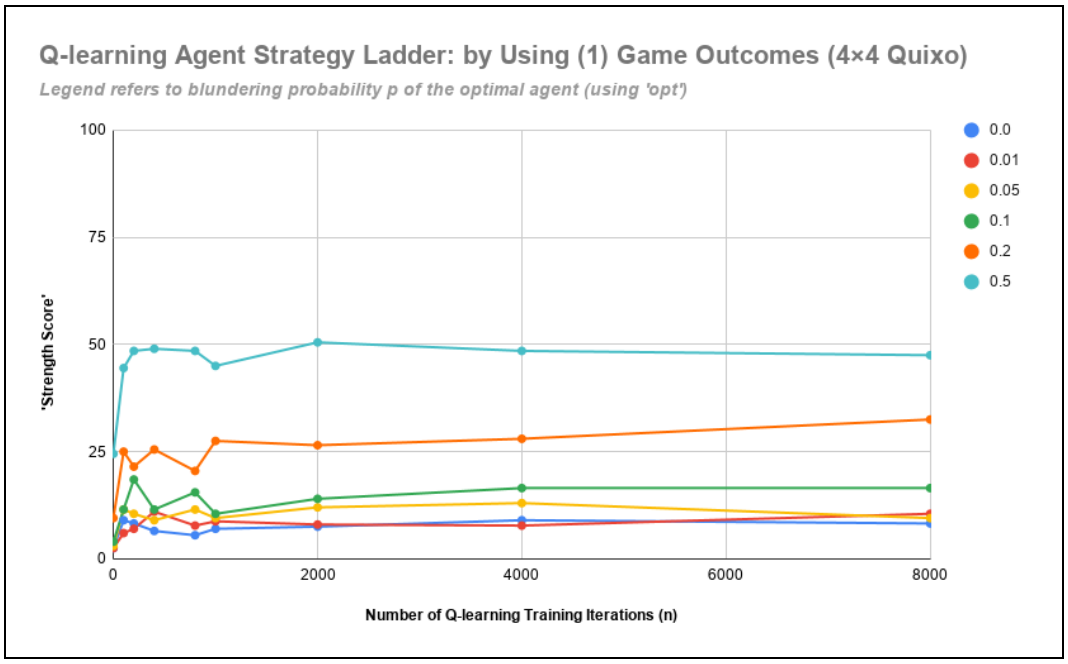
Although, it is worth noting that this phenomenon is much less pronounced in this comparative strategy ladder graph, than in those comparative strategy ladder graphs.

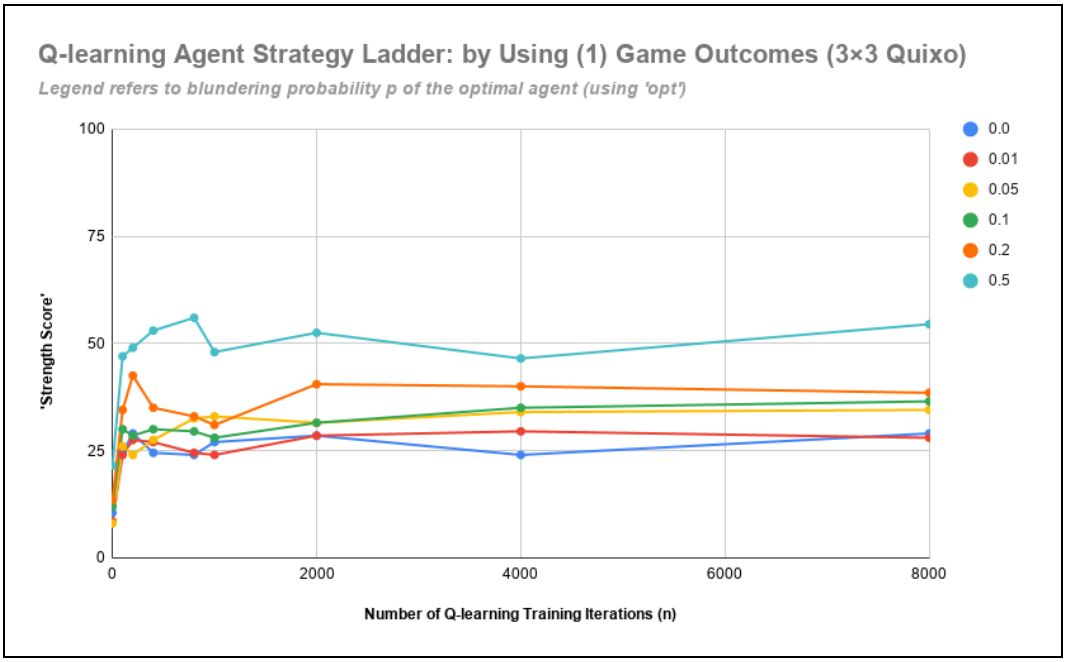
3.6 Q-learning Agent Strategy Ladder: Quantifying Solution Strength via (1) Game Outcomes

3.6.1 Results Quantified by Number of Training Iterations, Individually for 5×5, 4×4, and 3×3 Quixo

Moving on from the MCTS agent, using the same experimental setup of Section 3.3 to quantify solution strength via game outcomes, the following three strategy ladder graphs are obtained for the Q-learning agent, when playing against `opt`, for 5×5, 4×4, and 3×3 Quixo:







Again, this graph reflects the 'skill chain' of the Q-learning agent learning to play the game better with an increasing number of training iterations, and eventually reaching a horizontal asymptote.

And again, these graphs fulfill the basic sanity check that the Q-learning agent attains increasing values of its 'strength score' as the optimal agent blunders more often (i.e. with increasing p).

However, crucially, the Q-learning agent's asymptotic solution strength falls significantly short of those of the MCTS agent's (comparing these graphs with the corresponding ones in Section 3.3.2). For example, for 5×5 Quixo, the Q-learning agent attains asymptotic 'strength score's of only about 3 when p = 0.0, and about 4 when p = 0.01; while the MCTS agent attains much larger asymptotic 'strength score's of about 50 when p = 0.0, and about 85 when p = 0.01. This huge discrepancy could be indicative of Quixo being a very hard game to learn solely using the easily humanly-identifiable features of the Quixo board that are provided to the Q-learning agent (as described in Section 2.4.1). In particular, this is potentially due to the dynamic nature of the Quixo's tile movement mechanism, which renders any of these features inconsequential after only one or two moves. For this reason (i.e. that the Q-learning agent does not learn to play the game to any reasonable competency), no further experiments were conducted for the Q-learning agent.

4 Conclusion

4.1 Are Differences in 'Depth' between Sizes of Quixo Revealed by the Strategy Ladder Model?

To conclude this project, it is worthwhile revisiting the original goals of this project described in Section 1.3. Specifically, recall that this project seeks to understand whether the 'depth' of Quixo can be revealed by Lantz et al.'s strategy ladder model — and further, whether the differences in 'depth' between sizes of Quixo can be revealed by this strategy ladder model.

To summarize all the results presented in Section 3, the short answer to both of these questions are 'yes' to both. In particular, the comparative strategy ladder graphs quantifying solution strength via (1) game outcomes — presented in Section 3.3.4 — demonstrate that the differences in 'depth' between different sizes of Quixo are revealed by the differences in the position along the *x*-axis of the strategy ladder (i.e. "Computational Resources"), at which the strategy ladder starts to reach a horizontal asymptote (i.e. of "Solution Strength"). And, to a lesser extent, the same phenomenon can also be seen in the comparative strategy ladder graph quantifying solution strength via (3) game length — presented in Section 3.5.2.

That being said, there are also some caveats as to the extent to which these differences in 'depth' can be revealed by this strategy ladder model. Notably, the comparative strategy ladder graph quantifying solution strength via (2) move accuracy — presented in Section 3.4.2 — do not display the same phenomenon.

As such, to sum up, even though the answers to both of the original research questions of this project are 'yes', this project also reveals that the choice of quantification of the *y*-axis of the strategy ladder graph (i.e. "Solution Strength") is extremely important for the investigation of 'depth' via the strategy ladder model.

4.2 Other Notable Features of Quixo

As an aside to the main goals of this project, a brief summary of the other notable features of Quixo that are also incidentally revealed by this project's investigation, as well as some potential implications, is as follows:

1. *Quixo seems to be a very 'mistake-punishing' game (detailed in Section 3.4.2).*

   Specifically, the occasional single-move mistake among mostly good moves, can still be exploited by a good opponent easily, and can lead to an overall game loss. It is unclear whether this is a positive or a negative, from a game design standpoint.

2. *Quixo seems to be a very difficult game to play solely using easily humanly-identifiable features of the Quixo board (detailed in Section 3.6.1).*

   At least prima facie, this seems to be a negative, from a game design standpoint, since this suggests that Quixo is hard to learn for normal human players, who likely base their learning of the game on these features.

4.3 Possible Future Research Directions

Some possible future research directions, based on the results of this project, are as follows:

- *Develop a formal statistical model to quantify the 'depth' of a game based on its strategy ladder graph.*

  While the differences in 'depth' between strategy ladder graphs can be visually inspected (i.e. via the differences in the position along the *x*-axis of the graph at which the graph starts to reach a horizontal asymptote), there is currently no formal statistical way to quantify this *x*-axis position at which the horizontal asymptote 'starts'. It is worthwhile developing a formal statistical model to quantify this, to allow for a more objective quantification of 'depth'.

- *Investigate the same strategy ladder graphs for other games, apart from Quixo.*

  The same experimental setups for the various quantifications of the *y*-axis of the strategy ladder model (i.e. the performance of the computational learning agent) can be re-used for these further investigations. It is worthwhile investigating whether similar differences in 'depth' between variants of the same game can be revealed by the strategy ladder model, in other games besides Quixo.

- *Investigate the strategy ladder graphs specifically for a Q-learning agent (or other computational learning agents), in games that can be better learnt with Q-learning (i.e. not Quixo).*

  The significance of investigating the strategy ladder graphs specifically for a different type of computational learning agent (i.e. apart from the MCTS agent), is that the notion of 'depth' revealed by those strategy ladder graphs — and possibly the shape of those graphs — can potentially be qualitatively different from the strategy ladder graphs for the MCTS agent. And if so, it becomes worthwhile investigating which type of computational learning agent yields strategy ladder graphs, and correspondingly notions of 'depth', that best aligns with the game designing and playing community's existing notion of 'depth'.

5 References

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games, 4*(1), 1-43.

Lantz, F., Isaksen, A., Jaffe, A., Nealen, A., & Togelius, J. (2017). Depth in strategic games. In *31st AAAI Conference on Artificial Intelligence, AAAI 2017* (pp. 967-974). AI Access Foundation.

Tanaka, S., Bonnet, F., Tixeuil, S., & Tamura, Y. (2020). Quixo Is Solved. *arXiv preprint arXiv:2007.15895.*

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning, 8*(3-4), 279-292.

6 Acknowledgements

Thank you to my advisor Professor James Glenn for his invaluable support and guidance throughout this project, and to the Hopper community for giving me the opportunity to present this project at one of the weekly Hopper Mellon forums (slides attached here).