Adversarial Search and Machine Learning applied to Bamboo

Debnath.L Hartmann.A Gauthier.B Terragni.G Guetta.G De Koster.S

Department of Data Science and Knowledge Engineering
Universiteit Maastricht

January 18, 2022

I. Abstract

Bamboo is a two player strategy game played on a hexagonal grid where each player places counters according to a rule set and the last player able to make a legal move is the winner. A number of heuristics were explored and evaluated and used to develop MiniMax, Monte Carlo Tree Search, and Neural Network AI players. These players were evaluated against a randomised player and each other to identify the best heuristic for the game, the optimal hyper-parameters for each AI and the best AI overall. The time taken for players to make a move was considered as a factor for the best AI.

MiniMax with Alpha-Beta pruning was found to provide the best player, however, it is not scale-able for larger grid sizes.

II. Introduction

Bamboo is a simple, two-player, turn-based, winner-takes-all game in which players place counters on an empty hexagonal board to form groups of counters [1]. Created in March 2021, it is relatively unknown and has not seen the attention of AI research as more established games have (Go, Chess, Draughts etc). The game is played on an initially empty hexagonal board, with each player placing counters in turn. Players must play a counter every turn, with the aim of forcing the opponent to be left on their turn without a legal move to play. The game is fully observable and deterministic.

A. Game Mechanics

Several definitions are required for the discussion of the intricacies of the subject. To facilitate the indication of locations of tiles (spaces on the hexagonal board) and counters (colours assigned to tiles to indicate player ownership) a 3-Dimensional (3D) system of coordinates will be used [2]. This will take the shape of a slice of a 3D matrix for which all the entries are those for which the sum $= 0$. More formally a vector v where,

$$v \in \{x | x \in \mathbb{Z}^3, x_1 + x_2 + x_3 = 0\}, \qquad (1)$$

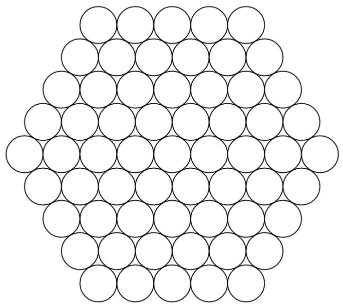
Fig. 1. An empty board at the start of the game

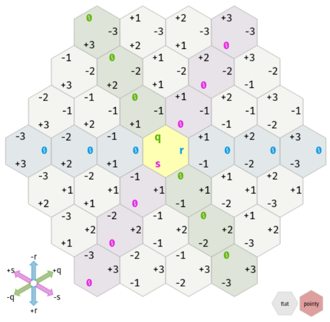

Fig. 2. A representation of a 3D coordinate system for a 2D hexagonal grid [2]

such that $\text{tile}(v)$ represents the tile at the location vector v.

Players are defined as one of two colours, in the case of this particular implementation staying true to the original author's wishes [1], the colours blue (b), red (r) and white (w) are used with the function of colour for a tile t defined as:

$$\text{col}(t) \in \{r, b, w\}. \qquad (2)$$

A group g is defined as a set of two or more counters of the same colour which share an incident edge. Thus it can be assumed that, for any tile at vector v neighbours any other vector u if the difference $v - u$ for two dimensions is equal to 1. Therefore,

$$u \text{ neighbours } v \iff \sum (u_i - v_i)^2 = 2 \tag{3}$$

Additionally the function moves(b) and moves(r) indicate the legal moves available to the blue and red players respectively.

Several metrics are important for the playing of Bamboo. The size of a group $|g|$ is defined as the number of counters that exist within the group, this can be associated to a player such that $|g_b|$ is the size of a group for the blue player. The largest group for a player is denoted $\hat{g}_b$ the size of the largest group for a player is thus $|\hat{g}_b|$. The set of all groups for a player is indicated with G and it's size with $|G|$.

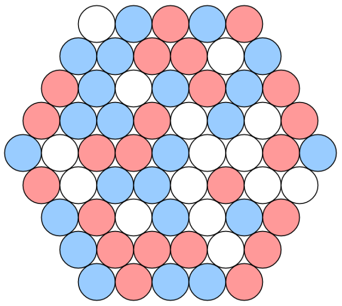

Fig. 3. A legal board showing $|G_b| = 10$, $\hat{g}_b = 6$, $|G_r| = 9$ and $\hat{g}_r = 5$

A single additional rule on placement adds a layer of complexity to the game; namely the size of the largest group cannot be larger than the total number of groups currently in play for that player. Formally,

$$\forall\ g \in G : (|g| \leq |G|). \tag{4}$$

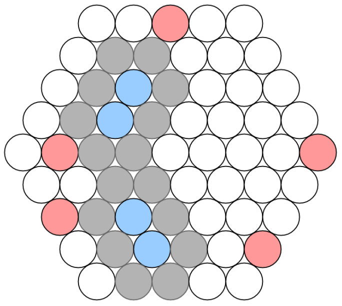

Fig. 4. Grey tiles indicate places where the blue player is unable to place counters without violating the rule $|g| \leq |G|$

The winner of the game is the last player to successfully place a counter, this requires the players to not only mind the size and quantity of their own groups, but to keep track of their opponent's metrics, identify their possible moves, and interdict wherever possible while maintaining options for a final placement for a winning move.

B. Analysis

1) Number of Moves: The number of possible moves within the board can be calculated using the formula:

$$n + \sum_{i=r}^{n} 2i \tag{5}$$

where r is the radius; the number of tiles counted from the centre tile along any of the coordinate axes and $n = 1+2(r-1)$ In a game where the grid is of radius 5 there are 61 possible moves at the start of the game.

C. Branching Factor Bounds

Assuming that each player is able to take any unoccupied place in the grid, and evaluates every possible branch, a total of 91! or 1.3520×10^{140} moves are possible. This provides a very high upper bound complexity of $O(b!)$, that is far from the actual value. An estimate closer to reality can be estimated using a Monte Carlo method by running games randomly and measuring the number of moves until game completion, taking the minimum number of turns to finish a game, 72. This reduces the lower bound of total evaluations to 72! or 6.1234×10^{103}
a reduction by a factor of 10^{37}. Finally if the number of evaluations can be measured for each game, then an average bound of 1.37863×10^{126} is arrived at experimentally. This still leaves the search problem as unsolvable in any reasonable time period.

$$E(\gamma) = (|G_r| + moves(r)) - (|G_b| + moves(b)) \tag{6}$$

D. Problem Statement

To generate an Artificial Intelligence model capable of routinely winning against an average human player of the game Bamboo.

E. Research Questions

1) What is the heuristic that provides the highest success rate if followed?
2) Is there an first move or last move advantage for the game Bamboo?
3) What is the optimal search depth for MiniMax within a reasonable time limit?
4) With optimal hyper parameters, can MCTS out-perform MiniMax in this "sudden-death" style game?
5) Can a neural network be trained to the same or higher standard of play than MiniMax/MCTS?
6) How can performance be increased using a hybrid of two algorithms?
7) What is the best algorithm for an agent to use in Bamboo?

III. Methods

A. Random

A player that has no knowledge of the game can effectively play by selecting a tile at random as their next move. This agent, on the balance of probability, has a 50% chance of winning against another randomly playing agent. An

implementation of a random agent provides the ideal baseline for testing the effectiveness of human and AI players by playing a statistically significant number of games and evaluating the number of wins and losses.
In addition to the random uniform selection of moves, several heuristics for move selection were used. Instead of selecting moves completely at random, the agent then orders moves according to some heuristic criterion and uses uniform shuffling only as a tie breaker. The heuristics were defined as follows:

1) Outer Weighting: It was observed that selecting the outermost tiles allowed the player to form groups around the board that were unlikely to be connected later on in the game. Thus the distance from zero to each tile was calculated by taking the maximum value of its vector, such that $\text{dist}(0, v) = \max(v)$. When selecting using the outer weighting heuristic, the tiles with the higher distance were prioritised and ties were broken randomly (see Figure 5).

2) Maximise Group Count: Keeping the number of groups high allows a player to keep open as many options as possible until the last possible play, increasing the likelihood of winning. The maximise group count heuristic selects the move that keeps the group count as high as possible.

3) Maximise Sparsity: Sparsity is defined as the average distance between all tiles of the same colour. As with the maximise group count, increasing the distance between tiles early on was observed to provide the player with an advantage later in the game as placing a tile was less likely to merge two groups. The distance from each candidate tile u, to a tile $v \in V$ where V is the set of all tiles of the same colour, was calculated. The tile with the highest average distance to all other tiles would be selected, with ties broken randomly.

4) Maximise Sparsity and Outer Weighting: The final heuristic used the Maximum Sparsity heuristic and breaks ties using the Outer Weighting method.

TABLE I
HEURISTICS FOR RANDOM AGENT MOVE SELECTION

Heuristic	Goal
Uniform	Entirely pseudo-random selection
Outer weighted	Prioritise outermost moves
Group count	Maximise the number of player groups
Sparsity	Maximise distance to closest friendly tile
Sparsity & Outer weighted	Sparsity breaking ties with Outer weighted

B. Minimax

MiniMax [3] is an algorithm that evaluates the states of the game and assumes that the opponent will always choose to minimise their losses. It builds a tree of possible game states from a given starting point using depth-first search. As it is not possible to search until every possible terminal game state is found, an evaluation function is used as a heuristic to evaluate a state at any point with a positive or negative value providing a metric for how beneficial the state is for the current player. MiniMax can be very slow in games with a branching factor as large as Bamboo's, its performance must be optimised. Alpha-beta pruning [4] is a method in which branches that cannot have an effect on the final evaluation of the MiniMax tree are pruned before further evaluation. To further improve the efficiency, a method of move ordering is applied. By ordering child nodes by applying a lightweight evaluation heuristic for each node in the tree, it is possible that more branches can be pruned.

C. Monte Carlo Tree Search

Monte Carlo methods utilise random sampling to obtain accurate predictions. The use of the technique competitively in Crazy Stone won the 9x9 Go tournament at the 2006 Computer Olympiad [5]. It is however unable to evaluate states mid-game and suffers from poorer performance in sudden-death games where a single position change can terminate the entire game [6]. Monte Carlo Tree Search (MCTS) is a best-first search method that generates a search tree by exploring new nodes, adding them to the search tree and randomly playing those nodes to completion. The wins or losses of these randomly played games are back-propagated up the tree and inform the selection of future nodes to be evaluated. The key strength of MCTS is that it does not require an evaluation function of the game state to evaluate the board, for an unknown game such as Bamboo this can allow for generating sets of moves that a heuristic based search may not find.

D. Neural Network

The game of Bamboo can be likened to many other turn based, fully observable games such as chess, draughts, Chinese Checkers and Go. The search space of potential moves is large, too large for a rapid and exhaustive analysis of all moves and future reactions from opponents. Humans do not analyse games in this exhaustive way [7], instead perceiving strategies in a less systematic, more intuitive style.
To implement this approach computationally, the use of Neural Networks can mimic the way that humans perceive game play. To generate the input to the network, the grid can be turned into a vector $\vec{x}$ where each entry $\in \{1, 0, -1\}$ (for the current player, empty, and opponent's tiles respectively) and is passed into the first layer of the network. The level of activation for each layer depends on the activation added to a weight w for each node within the network. By propagating forwards the network can indicate the output on the grid by selecting the element of the output vector with the highest activation through the use of a Softmax layer[8]. Generating a large sample of data allows the calculation of the difference between the output of the network $\hat{y}$ and the labeled output y. Back-propagating this difference through the network and adjusting the values of the weight by the partial derivative allows the network to learn the correct responses to the input stimuli. Forward propagation can be extremely quick when compared to search based agents. Further inspiration could be taken from

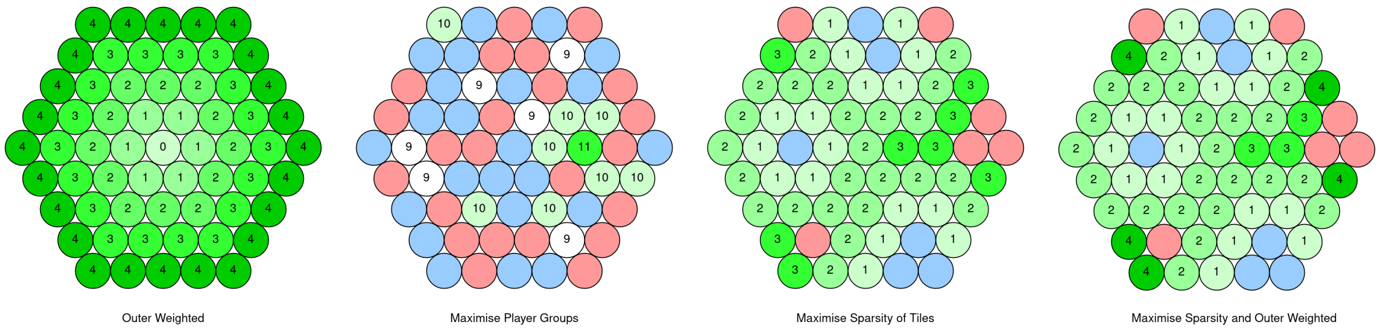
Outer Weighted
Maximise Player Groups
Maximise Sparsity of Tiles
Maximise Sparsity and Outer Weighted

Fig. 5. Visual representations of the decision making processes of each heuristic, numbers show the value given to each potential move

Deep Blue, the chess playing AI that defeated Kasparov in 1996 [9] by having two networks, a policy network and a value network. The role of the policy network is to reduce the possible number of moves, while the value network evaluates those moves to the most likely to win.

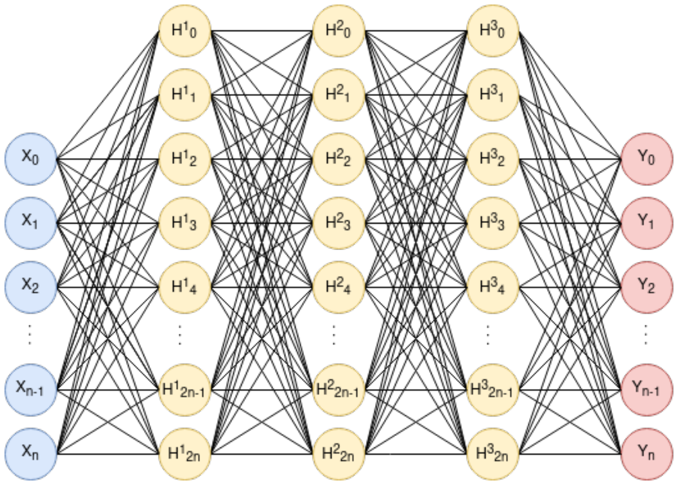

Fig. 6. Example of a neural network with three hidden layers of size $2n$ and $|Y| = |X|$

E. Hybrid agent

As a simple hybrid agent, a combination of neural network and minimax is presented here. This is done to get a decent but fast move selection in early stages of the game where many tiles are left and minimax would be slow at selecting a move. Therefore, the neural network, trained on minimax data, is employed to speed the early stages up. Later in the game, the agent will use pruned minimax with move ordering to look multiple turns ahead and select optimal moves.

IV. Implementation

A. Random

The implementation of the uniform random agent is based on a stack (see Figure 7). In the beginning all moves that would color uncoloured tiles from the grid are pushed onto the stack. The agent shuffles the stack and then pops the first move. This is then checked for legality, given the current grid state.

When applying a non-uniform heuristic (see Table I), the agent utilises a priority queue with a comparator that implements the selected heuristic. This serves the same purpose as the stack in the uniform version.
The selection process is repeated until a valid move is found, in case no tiles can be selected, the game ends and the opponent wins.

B. Minimax

For the MiniMax implementation, a node tree is built using depth-first search. Here, each node contains an evaluation, a grid state and a move that led to it. The agent looks for legal moves in the child nodes of the current node and then recursively calls MiniMax on legal child nodes until a specified search depth is reached (see Figure 8, Appendix A). At this point, the game state γ is statically evaluated.

$$E(\gamma) = (|G_r| + moves(r)) - (|G_b| + moves(b)) \tag{7}$$

The evaluation always occurs from the red player's perspective, meaning that the red player wants to maximize while blue aims to minimize the evaluations. Depending on what colour's turn it is, the minimum or maximum evaluation is passed up to the parent. The evaluations are back-propagated recursively up the tree, alternating between passing the minimum and maximum node evaluations up to the parents. This then yields the evaluation with the best worst-case outcome for the agent. The agents expects to get the best results, assuming that the opponent plays ideally (with regard to the evaluation function used). This process yields good results, but has factorial time complexity $O(n!)$. Therefore, alpha-beta pruning is employed to speed up the process.

1) Alpha-beta pruning: When using alpha-beta pruning, the underlying process of minimax remains unchanged. For each level one node, we now keep track of the best value for the maximizing player (α) and the best value for the minimizing player (β). Once the first branch is traversed, these values are set to the best values found on that branch. Before evaluating the next branch, for each node, it is checked whether the best evaluation for the current player can be greater than $alpha$. This is done by checking whether

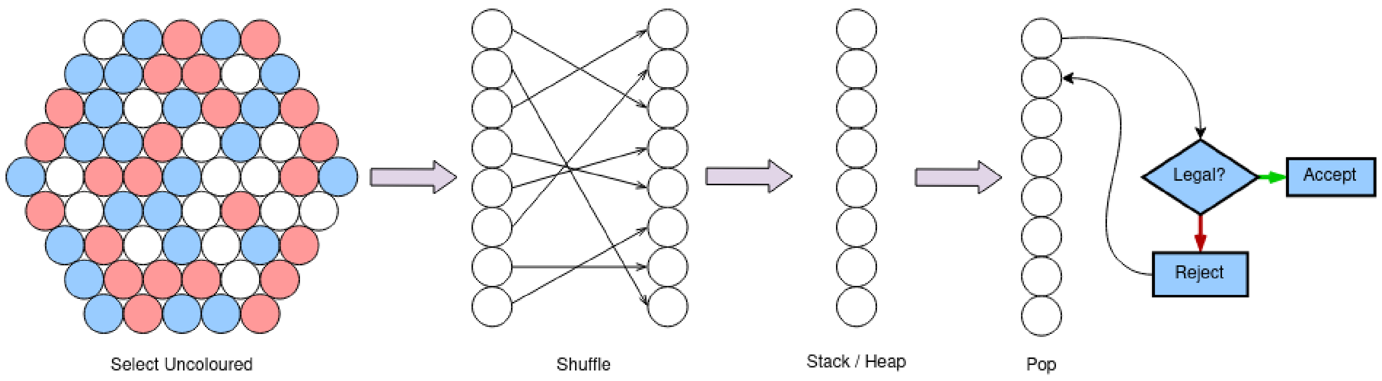

Fig. 7. Stages of selection of a pseudo-random move

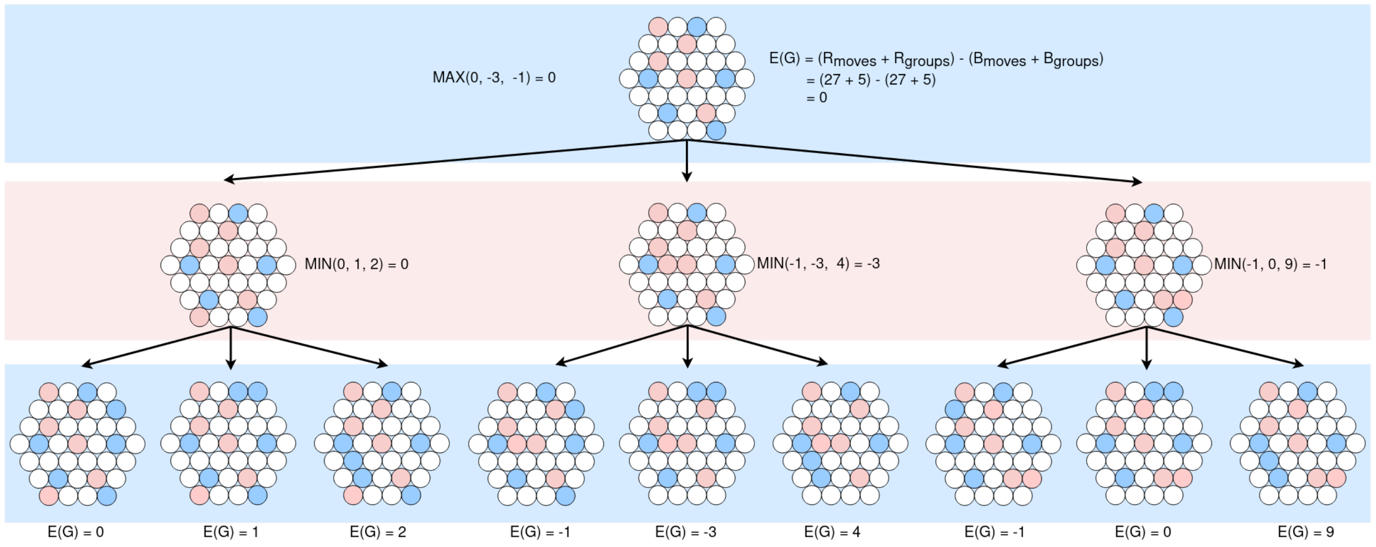

Fig. 8. MiniMax decision making

the other player has an option that is better than β, which would mean that this cannot be the ideal branch for the maximizing player, since the minimizing one will play a move that yields better results than one that was found earlier (and stored in the β variable). This can be seen in Figure 9 (and Appendix B), where it is shown how pruning branches can lead to fewer evaluations needed to achieve the same results. This performance gain of pruning branches heavily depends on the ordering of leaf nodes. In the best case, with child nodes perfectly ordered, pruned Minimax can achieve $O(\frac{n!}{(n-d)!} + d \cdot n)$, fully exploring one branch and pruning all others after one evaluation. In the worst-case scenario, a worst to best ordering, alpha-beta will never prune a single branch. For the inverse, performance can increase by a lot. Therefore, time complexity remains $O(n!)$, although pruning makes the worst case performance far less likely.

2) Alpha-beta sorted: As the gain from pruning branches heavily depends on node order, to further increase the likelihood of achieving best case pruning performance, one can sort the child nodes before evaluation (see Appendix C). In order not to have to evaluate the entire game state for sorting, a heuristic function is employed. Since every new node results from making a move (placing a tile), the heuristic for the resulting grid state will be based on the location of the target vector of the move.

The heuristic function used was the sum of squares of that vector, this was based on the observation that tiles at the extremities of the board are less likely to be linked into larger groups, keeping $|G|$ high and g_max low. This has the drawback of prioritising outside tiles, even when the outsides are already filled to some extent, increasing the likelihood of merging groups.

Since all vectors in the game are integer-based, the sum of squares will always be an integer. This allows sorting nodes using Radix sort [10], which runs in $O(d(n + k)$, where d is the number of digits of the largest element, k is the largest element (see Appendix D). This allows the agent to sort even larger lists of nodes without compromising performance. The pruning gain caused by sorting now depends on how well aligned static evaluation function and heuristic function are. If, in fact, outside tiles were always ideal, sorting would always lead to best case pruning performance. Since this is not the case though, and more likely to be the case in the

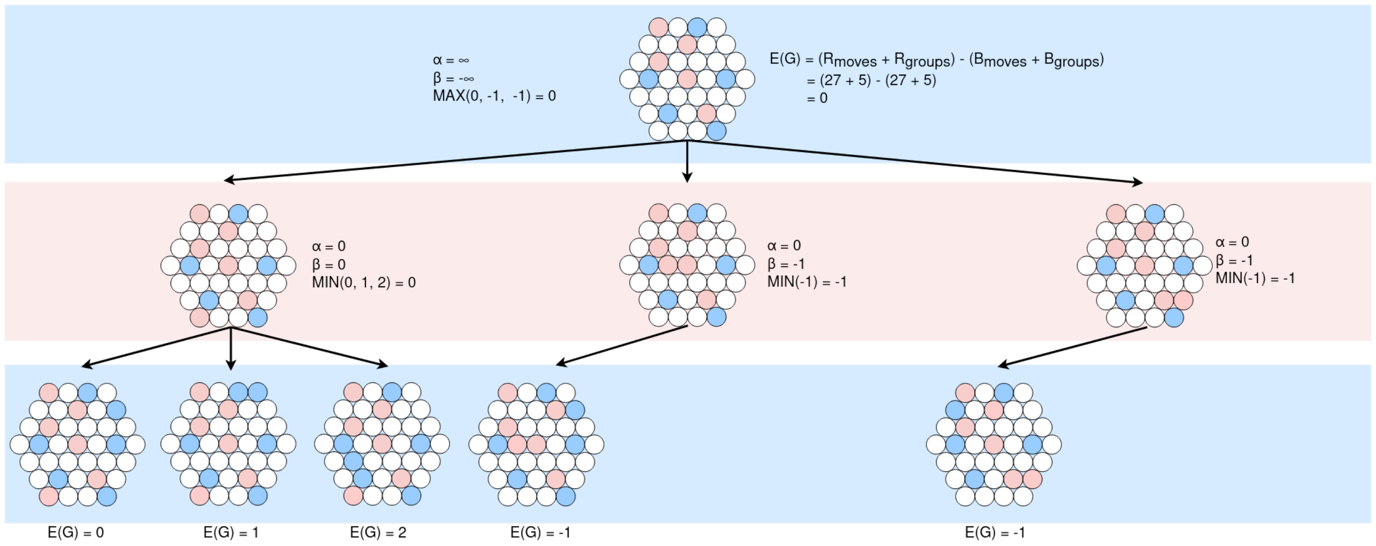

Fig. 9. MiniMax evaluation with pruning

beginning, sorting might prove most useful in the early stages of the game.

C. Monte Carlo Tree Search

The MCTS algorithm is composed of four stages that are run iteratively for t repetitions. The stages are Selection, Expansion, Play-out and Back-propagation. Each node i stores the grid state as well as several variables with information on the children of the node. Namely the value of the current position v_i from the average wins of games originating at this node and the visit count n_i of the node. To aid in retaining as much computation as possible, the tree from each iteration is saved in RAM and, when future moves are to be taken, the top is pruned to the current

1) Selection: In the selection phase the tree is traversed from the root node until a child is found that is not part of the tree. The child is selected using the Upper Confidence Bound for Trees (UCT) [11] which uses a value C to tune the exploration/exploitation decision making of the agent.

$$k \in \text{argmax}_{i \in I}\left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}}\right) \tag{8}$$

where i is the current node, v_i is the value of wins for the node i, n_p is the number of visits to the parent of i and n_i is the number of visits to node i.

2) Expansion: In the expansion step, the selected node is added to the tree.

3) Play-out: During play-out, a full game is simulated, with each player randomly playing moves until a terminal game state is reached. The game is recorded as either a win, returning a value 1, or a loss, returning 0. As there are no possible draws, the use of the set $\{1, 0, -1\}$ was eschewed in favour of a binary win/loss rate.

4) Back-propagation: During the back-propagation phase the results of the play-out are updated to the parent of the selected node, these changes are propagated up the tree until the root node. The move played by the program is the child of the root node with the highest visit count.

D. Neural Network

The Neural Network implementation consisted of two parts; modelling, data collection, and hyperparameters.

1) Modelling: The game Bamboo can be modelled in a simple manner as a data set of inputs X (the state of the current board) and outputs Y (actions to take given X). In this case $\vec{x}$ and $\vec{y}$ can both be formed as vectors where $\vec{x_i} \in X, \vec{y_i} \in Y$ and $\vec{x_i}, \vec{y_i} \in \mathbb{Z}^n$. The vectors are ordered lowest to highest, with all values in $X \in \{1, 0, -1\}$ encoded for current player, unoccupied, and opponent tiles respectively. The outputs are one hot encoded such that $Y \in \{0, 1\}$, with only one dimension of any value of $\vec{y}$ being non-zero.

2) Data Collection: The data was gathered by simulating games between MiniMax, MCTS, and Human agents and recording moves made. This was done to add pertubations to the data so that it did not become a clone of a single algorithm. Given the higher standard expected of the algorithmic solutions, their contribution to the data set was weighted by repeating their values within the set.

3) Hyperparameters: Initially a simple 3-hidden-layer, fully connected network was used. The size of each hidden layer was two times the size of the input vector, with a ReLU activation function. The output layer was fully connected and used a Softmax activation, outputting a one hot encoded

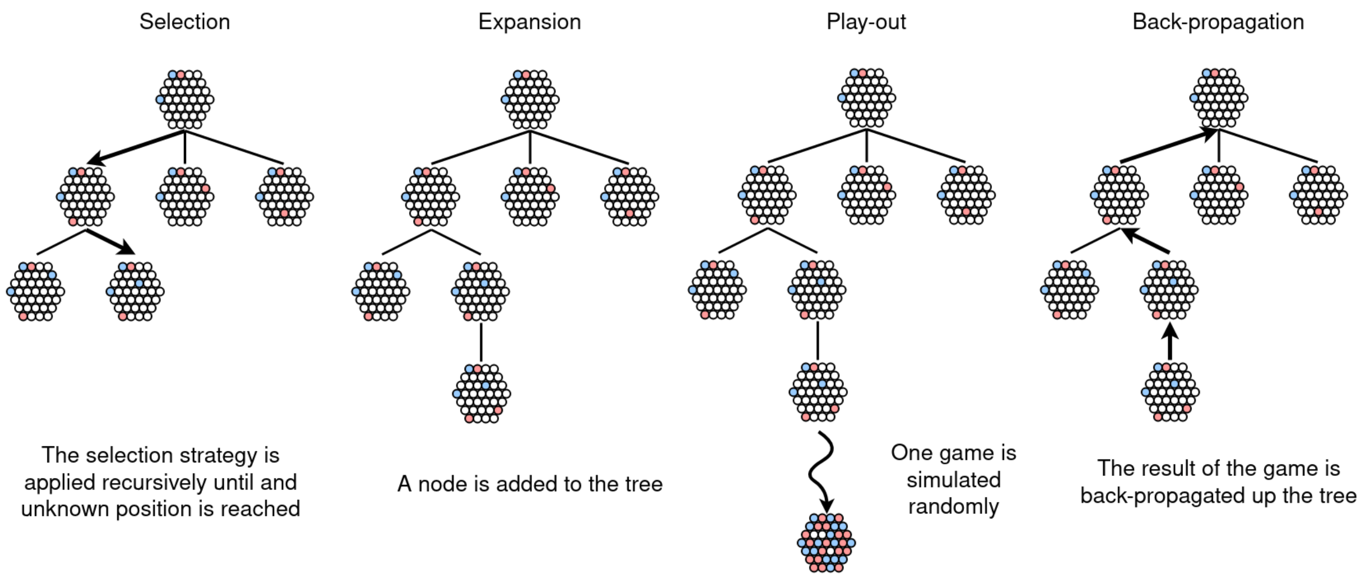
Selection
Expansion
Play-out
Back-propagation
The selection strategy is applied recursively until and unknown position is reached
A node is added to the tree
One game is simulated randomly
The result of the game is back-propagated up the tree

Fig. 10. The four stages of the MCTS algorithm

vector for the move with the highest activation.

E. Hybrid agent

The hybrid agent is supplied with a counter for empty tiles on the grid. Once this counter reaches a certain threshold, the agent switches from using NN to using Minimax. This threshold value is subject to experimentation in the following section.

V. Experimentation

A. Coin Toss

In order to run reliable experiments, we must control for as many outside influences as possible. One such influence is whether there is a significant first or last move advantage. This was tested by recording win rates of a random agent against another random agent in 100 games and recording if the first or last move player won. The results can be seen in Table II. It is quite obvious that, especially for small grid sizes, taking the first turn is a great disadvantage. Therefore, to control for this effect, for all experiments run in this section, each player starts exactly half of the games recorded.

TABLE II
Win Rates of starting agent in random vs random

Size	Starting win rate	LLCB	ULCB
1	12%	1.44%	22.56%
2	38%	29.05%	46.95%
3	45%	33.86%	56.14%
4	46%	34.57%	57.43%
5	49%	36.75%	61.25%

Note. 95% CIs based on assumed binomial distribution of win rates.

B. Heuristics

In this section, the win rates of random agents applying the different heuristics from Table I are presented. The performance of every heuristic against every other heuristic is also shown to point out possible strengths and weaknesses of the methods.

TABLE III
Win rate matrix of random agents applying different heuristics

	Agent 2				
Agent 1	U	O	G	S	S & O
Uniform (U)	-	-	-	-	-
Outer weighted (O)	100%*	-	-	-	-
Group count (G)	36%	0%*	-	-	-
Sparsity (S)	44%	0%*	54%	-	-
S & O	100%*	12%*	100%*	100%*	-

* Statistically significant difference from 50% at $\alpha = .05$

In Table III, one can see that the outer weighted heuristic allows the random agent to win consistently against all other heuristics. The only matchup that it does not win all the time is the one against the combined sparsity and outer weighted heuristic. This is expected though, since it employs outer weighted at its core as well. From this analysis, it is concluded that preferring outer tiles is the most effective heuristic of the ones discussed in this article.

C. Evaluating MiniMax

The aim is to find out at what search depth Minimax stops improving its efficiency, defined as milliseconds per win. To calculate this, win rates and time spent on turns were collected for 100 games against a random agent employing the outer weighted heuristic. In Figure 11, one can see that a search depth of 2 yields the best efficiency. Thus, it is concluded that a search depth of 2 is the most efficent way for Minimax to play Bamboo, given that search depth is kept constant.

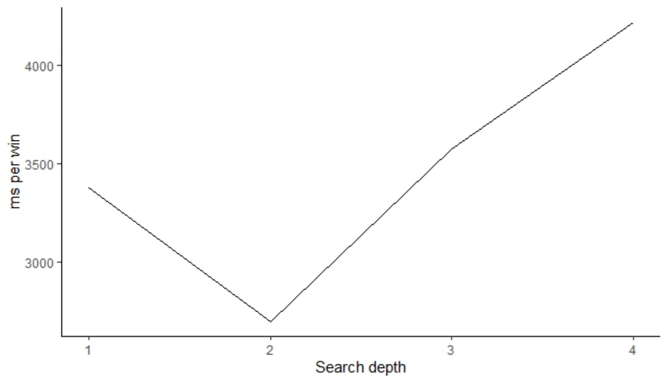

Fig. 11. Minimax efficiency plotted against search depth

D. Optimise MCTS

In order to judge whether MCTS could beat Minimax in Bamboo, the parameters c and iterations k must be optimized. To do so, MCTS win rates against a random uniform agent are recorded with $k \in \{1, 250, 500, 1000\}$, $c \in [0, 1]$ over 100 games. As expected, the agent performs best with the maximum iterations setting (1000). Note that when testing, MCTS took less than a second per turn, meaning that 1000 iterations is well playable. Thus time is not a factor here. A Loess-interpolated plot of the win rates can be seen in Figure 12. With 1000 iterations, the agent reaches a mean win rate of 87%, not correcting for the effect of c.

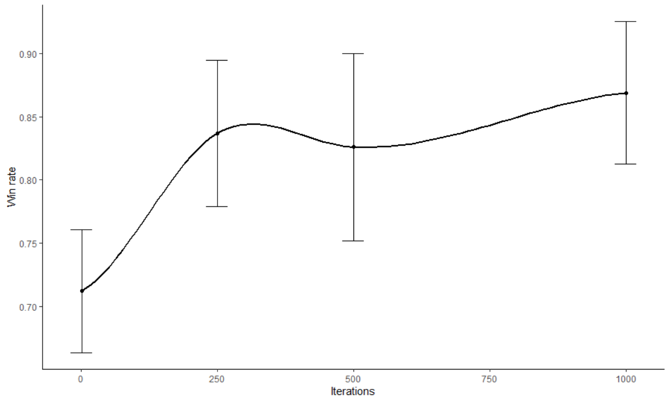

Fig. 12. Win rates by iterations with 95% CIs

For tuning the c value, it is less clear what the optimum is here. The maximum win rate with 1000 iterations is achieved with $c = .2$ and $c = .4$. Since the 95% lower CI bound over all iterations values is higher for the latter, it is decided that $c = .4$ represents the value closest to the optimum for this agent in Bamboo. This can be seen in Figure 13.

With this pair of settings, 1000 iterations and $c = .4$, the agent reached a mean win rate of 91%.

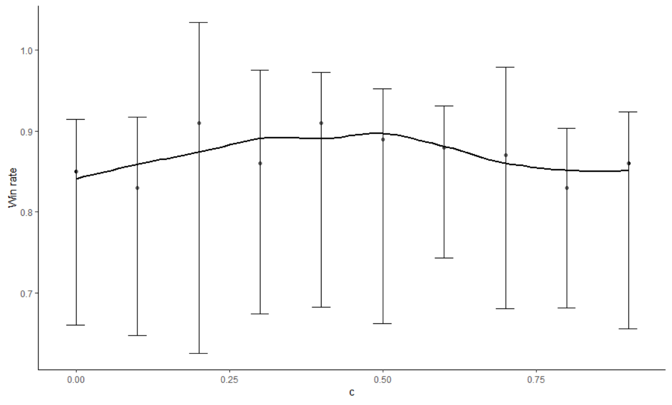

Fig. 13. Win rates by c value for 1000 iterations with 95% CIs over all iterations settings

E. Optimising hybrid agent

To find an optimal threshold for the hybrid agent to switch its approaches from NN to Minimax, an efficiency analysis is run. Over 100 games, win rates and time expenditure are recorded. To find a good threshold value, the aim is to find points at which the efficiency $\eta(i)$, defined as

$$\eta(i) = w_i - \frac{t_i}{\max(t)} \tag{9}$$

is optimal. Here, w_i represents the win rate achieved with threshold value i, t_i represents the time spent on selecting moves for threshold value i. Note that w_0, t_0 refer to w, t for a pure NN agent, since the threshold to switch to Minimax is 0, which can never be reached. In Figure 14, one can see that

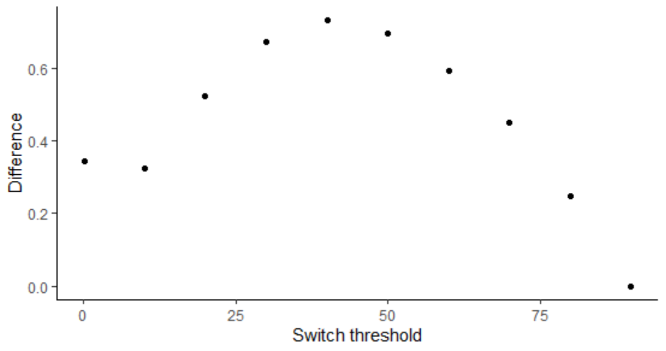

Fig. 14. Efficiency values for hybrid agent by threshold values

the optimal value for the switch threshold is 40. To understand what this means, see Figure 15.

Thus, 40 represents the point at which the difference between achieved win rate and relative time expenditure is maximal. Thus, for an optimally efficient result, the hybrid agent should employ NN until 40 tiles are left to be colored, at which point it switches to Minimax.

F. Determining the best agent

With the agents optimized to some degree, a round-robin style evaluation of agents is performed. Every agent plays

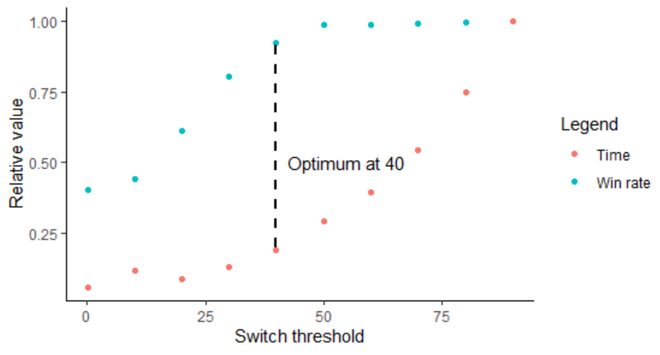

Fig. 15. Win rates and relative time expenditure by threshold value

100 games against every other agent, starting exactly half the games. The results can be seen in Table IV.

TABLE IV
WIN RATES OF AGENTS AGAINST EVERY OTHER AGENT

Agent	Random	Minimax	MCTS	NN	Hybrid
Random	-	-	-	-	-
Minimax	100%*	-	-	-	-
MCTS	88%*	1%*	-	-	-
Neural Net	52%	0%*	1%*	-	-
Hybrid	91%*	0%*	45%	100%	-

* Statistically significant difference from 50% at $\alpha = .05$

It is clear that the Minimax agent appears to perform best with win rates $\geq$ 99%. MCTS and the hybrid agent have the second highest win rates against the random uniform agent, with hybrid scoring a little higher. Still, MCTS is the only agent that managed to win a game against Minimax. Furthermore, the direct comparison goes to MCTS. Since this observation is not statistically significant, it is concluded that MCTS and hybrid perform similarly in Bamboo. The neural network performed the worst, not performing significantly better than the random uniform agent despite being trained on, mostly, Minimax data. Overall, it appears that Minimax is the best way to go for an agent, soundly beating MCTS as well as a feedforward neural network.

VI. Conclusion

The MiniMax algorithm has a great performance in terms of winning but, due to the requirement to evaluate many leaf nodes, does not scale well for gird sizes larger than those evaluated in this paper. It is noticeable when comparing the run time required to compute the next move with the run time of MCTS agent. The former takes approximately two seconds for the grid of size 5, while the latter takes just 0.5 seconds. The MiniMax algorithm, although evaluated as the best of the algorithms examines, is not necessarily optimal as it uses an evaluation heuristic that has not been proven to make optimal decisions.

Conversely, the Monte Carlo Tree Search algorithm does not require a heuristic or evaluation function involved. The downside of this algorithm is that it is unable to deal with the sudden-death nature of many of the boards it evaluates. Moreover, it is extremely slow at higher iteration values, thus a balance needs to be struck between speed and accurate simulation.

The neural network is much faster than MCTS, but exhibits much lower quality. Furthermore, it is constrained to the grid size it was trained and designed for, creating the need to create separate networks for each grid size. To combat the low move selection quality, which is on par with a random uniform agent, a different network architecture could be employed. For example, to retain the spatial aspects of the grid, instead of flattening it to an input vector, one could use a convolutional neural network in a follow up study.

References

[1] M. Steere, “Bamboo,” 2021. [Online]. Available: https://www.marksteeregames.com

[2] RedBlobGames, “Hexagonal grids,” 2020. [Online]. Available: https://www.redblobgames.com/grids/hexagons/#basics

[3] J. Pearl, “Asymptotic properties of minimax trees and game-searching procedures,” *Artificial Intelligence*, vol. 14, no. 2, pp. 113–138, 1980.

[4] D. E. Knuth and R. W. Moore, “An analysis of alpha-beta pruning,” *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[5] G. Chaslot, “Monte-carlo tree search,” *Netherlands Organisation for Scientific Research*, 2010.

[6] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J. B. Chaslot, and J. W. H. M. Uiterwijk, “Single-player monte-carlo tree search,” *Lecture Notes in Computer Science*, vol. Vol.5131, pp. 1–12, 2008.

[7] K. C, “How the computer beat the go player,” *Scientific American Mind*, vol. 27, no. 4, p. 20–23, 2016.

[8] R. A. Dunne and N. A. Campbell, “On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function,” in *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*, vol. 181. Citeseer, 1997, p. 185.

[9] T. Munakata, “Thoughts on deep blue vs. kasparov,” *Communications of the ACM*, vol. 39, no. 7, pp. 91–92, 1996.

[10] P. Horsmalahti, “Comparison of bucket sort and radix sort,” *arXiv preprint arXiv:1206.3511*, 2012.

[11] S. Levente.K, “Bandit based monte-carlo planning,” 2006.

Appendix

A. Pseudocode MiniMax

```
Minimax(rootNode,depth,maximizing)
    if maximizing
        currentColor := red
    else
        currentColor := blue
    end if
    grid := grid state of rootNode
    if depth == 0 or grid is finished
        rootNode value := static evaluation
        return static evaluation of grid
    end if
    add legal children to rootNode
    if maximizing
        maxEval := -Inf
        for each child in rootNode children
            eval := Minimax(child,
                            depth-1,false)
            maxEval := max(eval,maxEval)
        end for
        set rootNode value to maxEval
        return maxEval
    else
        minEval := Inf
        for each child in rootNode children
            eval := Minimax(child,
                            depth-1,true)
            minEval := min(eval,minEval)
        end for
        set rootNode value to minEval
        return minEval
    end if
```

B. Pseudocode pruned MiniMax

```
Minimax(rootNode,depth,alpha,beta,maximizing)
    if maximizing
        currentColor := red
    else
        currentColor := blue
    end if
    grid := grid state of rootNode
    if depth == 0 or grid is finished
        rootNode value := static evaluation
        return static evaluation of grid
    end if
    add legal children to rootNode
    if maximizing
        maxEval := -Inf
        for each child in rootNode children
            eval := Minimax(child,
                depth-1,alpha,beta,false)
            maxEval := max(eval,maxEval)
            if maxEval >= beta
                break
            end if
            alpha = max(alpha,maxEval)
        end for
        set rootNode value to maxEval
        return maxEval
    else
        minEval := Inf
        for each child in rootNode children
            eval := Minimax(child,
                depth-1,alpha,beta,true)
            minEval := min(eval,minEval)
            if minEval <= alpha
                break
            end if
            beta = min(beta,minEval)
        end for
        set rootNode value to minEval
        return minEval
    end if
```

C. Pseudocode sorted pruned MiniMax

```
Minimax(rootNode,depth,alpha,beta,maximizing)
    if maximizing
        currentColor := red
    else
        currentColor := blue
    end if
    grid := grid state of rootNode
    if depth == 0 or grid is finished
        rootNode value := static evaluation
        return static evaluation of grid
    end if
    add legal children to rootNode
    sort(rootNode children)
    if maximizing
        maxEval := -Inf
        for each child in rootNode children
            eval := Minimax(child,
            depth-1,alpha,beta,false)
            maxEval := max(eval,maxEval)
            if maxEval >= beta
                break
            end if
            alpha = max(alpha,maxEval)
        end for
        set rootNode value to maxEval
        return maxEval
    else
        minEval := Inf
        for each child in rootNode children
            eval := Minimax(child,
            depth-1,alpha,beta,true)
            minEval := min(eval,minEval)
            if minEval <= alpha
                break
            end if
            beta = min(beta,minEval)
        end for
        set rootNode value to minEval
        return minEval
    end if
```

D. Radix sort

```
sort(nodes)
    maxDigits := digits(max(nodes))
    buckets := List of 10 empty Deques
    for each digit i of node evaluations
        for each node in nodes
            digit := ith digit of node value
            add node to buckets[9-digit]
        end for
        clear nodes
        for each bucket
            while bucket is not empty
                add first element of bucket to nod
            end while
        end for
    end for
    return nodes
```