AI Implementation in Shobu

Eduardo Yap, Ngoc Tran, and Saugata Paul

University of Minnesota, Twin Cities

Abstract. Shobu is an abstract board game for two players designed by Manolis Vranas and Jamie Sajdak. It was released for the general public in August 2019. Since its recent release, the game has quickly gained popularity among board game fans due to its simple and elegant ruleset that lends itself to a high-skilled complex game. It immediately recalls the feeling of other mentally intensive and tactical games like Go and Chess, but has its own charm and charisma. We predict that the game will soon prove to be massively popular among enthusiasts and an AI race will begin. Our goal is to take the first step in the race and lay the groundwork for future Shobu AI development. In the end, we present two AIs that can beat novice to intermediate players through the use of a learned evaluation function.

1 Introduction

Throughout history, game playing has been perhaps the most remarkable application in the field of artificial intelligence. Humans are most remarkable for their intelligence and to see evidence of even higher intelligence is fascinating. In 1996, IBM's Deep Blue shocked the world by defeating the then best player in the world, Garry Kasparov, using optimized minimax algorithms [1]. Fast forward to now, game playing algorithms keep fascinating us by tackling harder challenges. In 2016, AlphaGo became the first AI to beat a Go world champion with the use of deep neural networks and novel Monte Carlo Tree Search methods [2]. In this paper, we apply many of the game playing algorithms developed throughout the field's history to the game of Shobu.

Shobu is two-player abstract perfect information game, that was recently released in August 2019. It was designed by Manolis Vranas and Jamie Sajdak and has received high praise from the abstract strategy game community. It's been praised as a game that looks simple on the surface but provides enormous room for complexity and depth. As more people discover the complex and strategic nature of the game, it's only a matter of time until it becomes a tournament game.

There is currently no published work in the development of computer programs for Shobu. So, one goal is to lay the groundwork for future Shobu AI development. The game of Shobu poses some interesting challenges. First of all, Shobu has a moderately high average branching factor of 91. But, there's also the uncertain tree complexity of the game. It's difficult to calculate how many

turns a game will last. If one wanted to, a game could go on forever through infinite stalling, unlike Go which is guaranteed to terminate after 170 moves.

We try to tackle these challenges by exploring various classical game playing algorithms, such as Minimax and Monte Carlo Tree Search, and through the construction of a strong evaluation function using temporal difference learning (TDL). The development of an AI that can defeat most of the current players can shed a light on the best strategies to adopt when playing the game and accelerate the development of Shobu's metagame.

1.1 Shobu

Rules [3] The game consists of four 4x4 square boards (2 white and 2 black boards), a rope, and 32 stones (16 black and 16 white). Figure 1 shows the initial setup. The black stones are placed on the bottom line of each board and white stones are placed on top. Each player has two homeboards (black and white). Black player starts the game.

Fig. 1. Setup of a game

A player's turn consists of two moves: a passive move and an aggressive move. During the passing move, a player moves one of the stones in his homeboards in any unblocked direction up to two spaces. Figure 2 shows the directions in which a chosen stone can move.

During the aggressive move, the player has to move a stone in the same direction and number of spaces as the passive move. This move must be made on the board colored opposite to the passive move. For example, if the passive move was made on a black board, the aggressive move must be made in one of the two white boards. The aggressive move can push an opponent's stone across

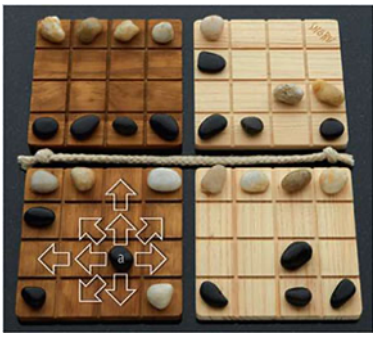

Fig. 2. Possible passive moves

the board and/or off the board. The attacking stone cannot push two stones at once or a stone of the same color. Figure 3 shows an example of possible passive/aggressive move combinations. It's possible that no aggressive moves can be made. In that case, a player only executes the passive move.

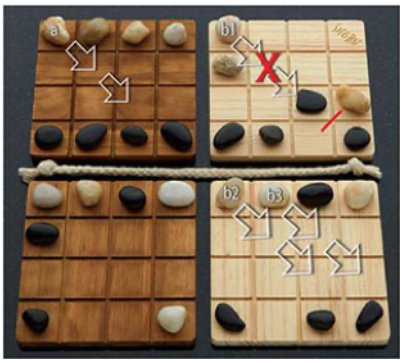

Fig. 3. Passive/aggressive moves.

To win, a player must remove all opponent's stones from just one of the boards.

Strategies A simple strategy that might come to many people's minds in their first time interacting with this game is focusing their aggressive moves on a single board. Since the goal is to knock all the opponent's stones out of a board, this seems to be the fastest way to victory at the first glance. However, this is actually a risky tactic. By prioritizing only one board, the player's stands in other boards might be weakened. Consequently, the opposing player can turn this to his own advantage and back the other player into a corner.

This leads to the existence of another more reliable strategy which is keeping the balance in all boards. Before making either a passive move or an aggressive move, a player should always make sure that any board will not be easily threatened after carrying out the action. In other words, it does not matter in which board one is holding the advantages; a player should also make sure that he still stands a chance in other boards. By managing to maintain this equilibrium state, the player will be able to evade or establish a defense for an attack and also set up a multi-goal move that guarantees a sure capture. Another key thing in playing Shobu is caution. This caution does not only come from being aware of a possible attack coming from the opposing player but also being aware of one's own move. In Shobu, an aggressive move can only be set up using a passive move. Considering a situation where there is a possible aggressive move on a board that can knock out this opponent's single stone. However, on the board of the passive move, there is not any available path to match with this aggressive move. This move, therefore, cannot be made. This will delay a possible win and might create a chance for the opposing player to turn the table.

Beside those big overall strategies, there are some other small strategy tips for the game. A good way of defending a possible attack is not only moving the threatened piece or blocking the aggressive move using another stone but also blocking the path of opponent's passive move that lead to the aggressive move. In addition, an attack can be stopped not only by using one's own piece but also manipulating the opponent's piece to obstruct the capturing path.

Properties We will specify the game properties of Shobu as outlined by Louis Victor Allis in his thesis, Searching for Solutions in Games and Artificial Intelligence [4].

Shobu is a perfect-information game, in which all players, at any time during the game, have access to all information defining the game state and its possible continuations. Throughout the game of Shobu, each player has access to all the information the other player has, including past moves and possible future moves. This is neat, because there's no need to implement probabilistic algorithms.

Shobu is a sudden-death game. Sudden death games are characterized by the termination of the game when a specific pattern is met. In the case of Shobu, that pattern is when all of a player's stones in one board are captured. In contrast, a fixed-termination game such as Go ends after a set number of turns. The sudden-death game property can difficult the use of game-playing algorithms since the path to a winning position can be narrow. When the game has a hard time converging to a goal state, one must rely on an evaluation function to approximate the theoretical value of the current state.

The game-tree complexity relies on two different values: the solution depth, d, and the branching factor, b. The game-tree complexity can be approximated as b^d. It might be unfeasible to come up with a theoretical average branching factor for Shobu. We'll rely on statistics instead to approximate the value. Figure 4 shows the distribution of number of legal moves from 21 055 different states.

The average branching factor is 90.57, which we'll round up to 91. The max branching factor is 232, which occurs on the initial state of the game.

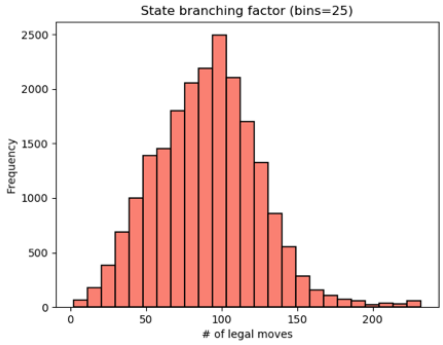

Fig. 4. Branching factor distribution.

The depth is even trickier to approximate. The sudden-death property makes reaching a goal state difficult. Using random simulations can lead to unrealistic solution depths as we can reach state sequences that fail to converge to a solution. A guided intelligent simulation would be more realistic, but difficult to generate. We'll use all the games played in the AI tournament from the Results section to get an average depth for intelligent simulations. Figure 5 shows the distribution of game length for 214 games. The average depth was 23.16, which we'll round to 23.

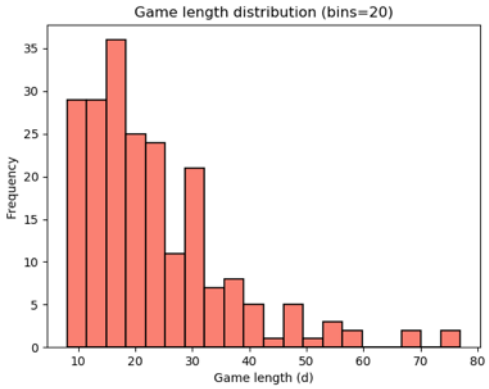

Fig. 5. Game length distribution.

This leads to a game-tree complexity of approximately $91\hat{2}3$ approx $10\hat{4}5$.

2 Problem Description

There are currently no existing Shobu AI programs, since the game is fairly new, but we anticipate them to start emerging soon. This project will serve as an exploration and implementation of candidate game playing algorithms. We'll establish what algorithms work and how they could be improved to fit the properties of Shobu. The AI's developed will serve as a benchmark for future agents.

There's also no current Shobu computer program implementation available, so the logic of the game will have to be coded from scratch. The game will be implemented in such a way that makes search easy for the algorithms.

Finally, we acknowledge that a full search of the game tree is infeasible for our purposes. Therefore, a leaf utility function will have to be replaced by a well-designed evaluation function. An important component of the project will be the development of this function through known algorithms. We hope this evaluation function can shed light to novel strategies that humans can adopt into their play in order to improve at the game.

3 Prior Work

3.1 Searching Algorithms

Minimax and Alpha-Beta Pruning The base minimax algorithm is simply a depth-first search. Once a leaf is reached, a utility function is used to calculate its value. The value is backed up through the tree and the minimax value is stored for each node. The turn player is always the maximizing player.

Alpha-beta pruning is an improvement over minimax. It exploits the fact that a node search can be cut off early if we can guarantee that the following branches will not be selected by the minimax algorithm. The algorithm is given in Figure 6 [5].

The time complexity for the minimax algorithm is $O(b^m)$. With alpha-beta pruning and optimal order search, the runtime can be as low as $O(b^{m/2})$. For obvious reasons, plain alpha-beta pruning is just not feasible for Shobu. The maximum depth of the game, for all we know, could approximate infinity.

A very efficient alternative is to cut-off the search early and apply an evaluation function to evaluate the state of the tree. With a good evaluation function, the value of a state can serve as an accurate approximation of the utility function. Let's assume we use this algorithm in Shobu and cut-off the search when a depth of 3 is reached. Let's assume the worst case where we are at the beginning of the game. Recall that early states of the game have branching factors as high as 232. Applying depth-limited minimax on the initial state will result in searching at most 232^3 other states. This runtime is still not practical.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Fig. 6. Alpha-beta pruning.

A further improvement that can be made is limiting the breadth as well. If we have a good evaluation function, we might as well only expand the best children at each minimax step. The MAX player first expands only the best 5 children, then the MIN player expands the 5 worst children, and so on. Let us assume that we have a breadth-limit of 5 and a depth-limit of 3 and assume we are at the initial state. The MAX player will have to evaluate the 232 children states, but it willll only expand the best 5. Then, the MIN player will evaluate the children and generate the 5 worst next states for each of them. Finally, the MAX player backs up the values of the 5 best states.

In the end, we would have evaluated at most $232 + 232 * 5 + 232 * 25 = 7192$ states. This is a practical amount that can be searched in a reasonable amount of time. Keep in mind that this still depends on a good evaluation function. We will talk about its creation in a later section.

Quiescence Search One final alternative to alpha-beta pruning which will be considered is quiescence search. One of the problems with depth-limited search is that the search might be stopped at a state that evaluates to a really promising value. This can sometimes be misleading if it turns out the next best move for the opponent player leads to a dramatic change in value. This is known as the horizon effect. Instead of terminating the search at a set depth, quiescence search terminates at a position that is, unlikely to exhibit wild swings in value in the near future [5].

Take for example a Shobu game in which Black has significantly more stones than White. An evaluation function might return a high value for black. However, White's next move could lead to a threatening position to one of Black's lone stones in a board. This position is extremely unfavorable for Black, so the eval-

uation function will have a high swing. If White's move is not explored, there's little chance that black makes a move to avoid it. The algorithm doesn't provide runtime improvements, but it does provide a more robust decision-making policy. The full algorithm is in Figure 7 [6].

```
int Quiesce( int alpha, int beta ) {
   int stand_pat = Evaluate();
   if( stand_pat >= beta )
      return beta;
   if( alpha < stand_pat )
      alpha = stand_pat;

   until( every_capture_has_been_examined ) {
      MakeCapture();
      score = -Quiesce( -beta, -alpha );
      TakeBackMove();

      if( score >= beta )
         return beta;
      if( score > alpha )
         alpha = score;
   }
   return alpha;
}
```

Fig. 7. Quiescence Search

Monte Carlo Tree Search Monte Carlo Tree Search (MCTS) is a heuristic search algorithm. It is an algorithm usually employed in games to predict the number of moves required to reach the final winning state. At its core, it is a tree search which gives more privilege to the promising nodes and allows their child to be searched faster. This way, it ensures that the solution is reached the fastest. Another advantage of MCTS is that it can be terminated at any time, which will output the most promising result upto that time. MCTS employs a four step round as its principle of operation. Each round of a Monte Carlo Tree Search consists of:

- **Selection** A node R is selected and expanded with its most promising child nodes until a leaf node is reached. A leaf node L is a node from which no playout has been initiated yet.
- **Expansion** If the leaf node L does not provide a definite win/loss/draw result, then the node is expanded further to make one or more child nodes C which comprise of the valid moves possible from the leaf node.
- **Simulation** A complete game is simulated taking C as the base node.
- **Backpropagation** The results of the game, assuming C as the base node are sent to the parent nodes all the way back to R which helps to decide the viability of the moves in that direction.

The above rounds are repeated for a variable amount of time [14].

Monte Carlo Tree Search Early Playout Termination Early Playout Termination in Monte Carlo Tree Search is a technique [13] to terminate the MCTS algorithm prematurely without reaching the final state. It can be applied to get the most optimised result upto that point or when the most optimal method might not be considered due to various constraints. This feature of MCTS is really an advantage considering that the machines that would probably run the algorithm might not all support the same level of specifications or can compete at the same level. Terminating MCTS near the end of the search tree outputs almost the same or a bit worse results, but which can be better return in terms of time and energy.

3.2 Evaluation Function

An evaluation function is necessary for most of the algorithms described so far. To recapitulate, an evaluation function is an approximation of the utility function given a state. If a state is promising for a player, the evaluation function will assign a high value. Often the success of a game playing algorithm relies solely on the evaluation function. The design of a good evaluation function is not easy. First, one needs to come up with various features that can be extracted from each state. For example, in the game of Shobu, some obvious features could be the amount of stones you have and the amount of lone stones your opponent has in each board. Some less obvious features are the average stone mobility (number of available moves) and the number of unique stones threatened. Secondly, one needs to assign weights to these values based on the importance of each feature to the outcome of the game. This type of evaluation function is known as a weighted linear function and it can be expressed as such:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

Coming up with good handcrafted weights requires a lot of domain knowledge. Fortunately, there are techniques that help approximate weights. We'll explore one of those techniques called temporal difference learning.

Temporal Difference Learning Temporal difference learning is a reinforcement learning method that has seen great success in games such as Backgammon and Othello [7], [8]. More specifically, Richard Sutton introduces a generalized formula for TDL called TD-Lambda which is

$$\Delta w_t = \alpha(V(x_{t+1}, w) - V(x_t, w) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w V(x_k, w)$$

Here, V_t denotes the value function at time step t, x_t refers to the feature values at time step t, w_t is the vector containing the value for all feature weights at

time step t. α specifies the learning rate. It controls the rate at which the feature weights are update. The gradient $\nabla_w V(x_t, w)$ is then defined as

$$\left(\frac{\partial V(x_t, w)}{\partial w_1}, \frac{\partial V(x_t, w)}{\partial w_2}, \ldots, \frac{\partial V(x_t, w)}{\partial w_M}\right)$$

Finally, λ is a constant that lies between 0 and 1. As the number of time steps increase, λ exponentially weighs less previous steps, while the current timestep is not discounted at all.

The update procedure occurs through self-play, where steps are taken according to an ϵ-greedy policy. For every step, we choose the next step that maximizes the value function, but there's a ϵ chance we take a random step instead. This is to balance exploration and exploitation. After a move is taken, weights are updated. The process repeats until the game reaches a goal state.

4 Proposed Approach

In this section, we demonstrate our approach in the implementation of the Shobu game and the algorithms described.

4.1 Shobu Implementation

As mentioned earlier, it's in our best interest to implement the game in a way that fits a search problem description. Therefore, our implementation mainly focuses on the following elements: an appropriate state representation of the game, a function that generates all possible actions that can be taken from the current state, a function that returns a new state given an action, and a utility function to determine where the current state is a terminal state.

A Shobu board consists of 4 4x4 boards. We'll represent the board as a 4x16 array that are either three different values: 0 (Empty), 1 (BLACK) and -1 (WHITE). For example, the initial state can be represented as

```
[[-1. -1. -1. -1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
 [-1. -1. -1. -1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
 [-1. -1. -1. -1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
 [-1. -1. -1. -1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]]
```

The state representation will also hold a player variable that indicates which player has the move, and a winner variable that indicates who the winner is, if there is one.

The functions for checking the goal state and performing an action are straightforward and are reasonably fast. One thing to note is that performing an action does not mutate the current state. It returns a new state data structure. This makes it easier to represent the states as nodes for the search algorithms.

The function for generating all possible valid moves is the most expensive and poses a bottleneck on the move simulation process. It's slow, because it has to

validate every single possible move for each active stone. The validation process can be extraneous, because a move actually consists of 2 different moves with different constraints to them. Efforts had to be made to optimize the algorithm by coming up with clever shortcuts that could skip the validation process for some moves.

Full implementation details can be found in our GitHub repository [10].

4.2 Implementation of Algorithms

For the algorithms, we were able to implement most of them without any deviation from their description. The search implementations are game-agnostic, meaning that any game implementation can use our interface as long as they provide the right API. Search implementations are abstracted through a Player class. The game is passed as a parameter to the class and it's ready to play using any of the algorithms.

The first agent Is called GreedyPlayer. During its turn, it always chooses a move that can capture an opponent stone. If there is no such move, it chooses a random move. This will serve as a baseline for the other players.

The next player is the MinimaxPlayer. The minimax with alpha-beta pruning was implemented based on Russell's pseudo-code [5]. However, it takes additional depth, breadth and evaluation parameters. The depth determines the depth at which search should stop and the evaluation function should be applied. The breadth specifies how many children to expand at each depth.

There's also the MonteCarloPlayer. It takes in a timeout parameter that determines for how long Monte Carlo can perform its exploration. The Monte Carlo algorithm implemented is based on dbravender's implementation [11], but modified significantly to fit our API.

Further modifications had to be made in order to implement the MonteCarloEPTPlayer (Early Playout Termination). Our implementation takes in timeout, evaluation and depth as parameters. The depth parameter determines the depth at which the Monte Carlo simulation should be cut off. The evaluation function then assigns a value to this state that gets backpropagated back to the root.

For TD-Lambda, guidance for the implementation of the algorithm was taken from EllaBot's own Python implementation [12]. Significant modifications had to be made for the algorithm to fit our API and for our purposes. It mainly allows trained weights to be stored in a JSON file, so that learning can be continued later on. Also, we simplified the calculation of the weight gradient by assuming that our evaluation function will always be linear, which it is.

4.3 Evaluation Function

For the evaluation function, we came up with 48 different features, shown in Figure 8, related to the board state. We assign initial weights to each feature based on merely intuition. Later on, temporal difference learning will be applied to learn better weights.

my_stone_count	2	my_threatened_ones	-5	my_back_stone_count	0.1
his_stone_count	-2	his_threatened_ones	4	his_back_stone_count	-0.1
my_mobility	0.5	my_defensive_stones	0.1	my_vertical_stone_count1	0.1
his_mobility	-0.5	his_defensive_stones	-0.1	my_vertical_stone_count2	0.1
my_stones_threatened	-1	my_offensive_stones	0.1	his_vertical_stone_count1	-0.1
his_stones_threatened	1	his_offensive_stones	-0.1	his_vertical_stone_count2	-0.1
my_unique_stones_threatened	-0.8	my_unthreatened_stones	1	my_connected_stones	0.1
his_unique_stones_threatened	0.8	his_unthreatened_stones	-1	his_connected_stones	-0.1
my_ones	-2	my_active_stones	0.1	my_average_mobility	0.5
his_ones	2	his_active_stones	-0.1	his_average_mobility	-0.5
my_twos	-0.3	my_diagonal_stone_count1	0.1	my_average_threatening	0.5
his_twos	0.3	my_diagonal_stone_count2	0.1	his_average_threatening	-0.5
my_threes	0.3	his_diagonal_stone_count1	-0.1	diff_stone_count	1
his_threes	-0.3	his_diagonal_stone_count2	-0.1	diff_mobility	0.5
my_fours	2	my_offensive_front_stone_count	0.1	diff_threatened	0.5
his_fours	-2	his_offensive_front_stone_count	-0.1	diff_unique_threatened	0.5

Fig. 8. Initial feature weights.

The evaluation function is simply a linear combination of all the features and weights.

5 Results/Findings/Discussion

5.1 Results

Trained Evaluation Function We ran TDL on the evaluation function for 4000 games with a learning rate α of 0.001 and ϵ of 0.1. At first, the TDPlayer played against the GreedyPlayer. Once we saw that it could consistently beat it, we matched it up against a static version of its current self. Once it could consistently beat that version, we upgraded it to play against itself, and so on. After 4000 games, we observed no significant changes in weights, so we stopped training at that point.

Figure 9 shows the trained feature weights after 4000 games.

Shobu Tournament To assess the strength of the agents implemented, a round-robin tournament was designed in which each agent player 20 games against all the other agents. We also participated in the tournament, but only played 5 games instead of 20, since games can take too long. To avoid stalling, games that surpass 100 turns will be considered a draw. Our estimate for how long it takes us to plan a move is around 30-50 seconds. Then, to be fair, each agent is allowed a maximum of 60 seconds to plan their next move. If a move is not performed after 60 seconds, the agent loses.

We'll next define the parameters passed in to each agent for use in the tournament. We'll define eval_handcrafted as the evaluation function with the initial untrained weights, and eval_trained as the evaluation function with the final trained weights. The settings for the agent are as follows:

my_stone_count	2.335428	my_threatened_ones	-12.3717	my_back_stone_count	1.580074
his_stone_count	-4.67548	his_threatened_ones	5.673368	his_back_stone_count	-0.69728
my_mobility	0.774575	my_defensive_stones	-2.8794	my_vertical_stone_count1	0.322555
his_mobility	-1.68432	his_defensive_stones	-3.57831	my_vertical_stone_count2	0.548302
my_stones_threatened	-0.84166	my_offensive_stones	3.750253	his_vertical_stone_count1	-3.00231
his_stones_threatened	-0.40532	his_offensive_stones	-1.97265	his_vertical_stone_count2	-2.54865
my_unique_stones_threatened	-1.21317	my_unthreatened_stones	1.748599	my_connected_stones	1.583959
his_unique_stones_threatened	0.481887	his_unthreatened_stones	-3.35737	his_connected_stones	-0.46663
my_ones	-6.45448	my_active_stones	-1.19821	my_average_mobility	-0.57342
his_ones	6.615632	his_active_stones	-2.6494	his_average_mobility	-3.4279
my_twos	1.672251	my_diagonal_stone_count1	0.548302	my_average_threatening	0.284903
his_twos	-3.284	my_diagonal_stone_count2	0.322555	his_average_threatening	-0.56592
my_threes	0.855215	his_diagonal_stone_count1	-2.54865	diff_stone_count	4.010908
his_threes	-3.79897	his_diagonal_stone_count2	-3.00231	diff_mobility	1.95889
my_fours	2.046511	my_offensive_front_stone_count	-1.04465	diff_threatened	-1.06366
his_fours	-1.41316	his_offensive_front_stone_count	-2.1782	diff_unique_threatened	0.595058

Fig. 9. Trained feature weights.

1. Greedy Player: No settings
2. Monte Carlo: timeout=60
3. Monte Carlo EPT: timeout=60, depth=5, evaluate=eval_trained
4. Minimax: depth=3, breadth=9, evaluate=eval_handcrafted
5. Minimax: depth=3, breadth=9 evaluate=eval_trained
6. Human Player: 1 month of experience playing Shobu

Figure 10 shows the results.

	Greedy	Monte Carlo (MC)	Minimax w/ Initial Weights	MC EPT w/ Trained Weights	Minimax w/ Trained Weights	Human
Greedy		5%	75%	100%	100%	100%
Monte Carlo	95%		85%	95%	100%	100%
Minimax w/ Initial Weights	25%	15%		40%	100%	20%
MC EPT w/ Trained Weights	0%	5%	55%		80%	20%
Minimax w/ Trained Weights	0%	0%	0%	10%		0%
Human	0%	0%	40%	60%	100%	

Fig. 10. Shobu tournament results.

5.2 Discussion

Analysis of the Evaluation Function The two features with highest magnitude are my_ones and his_ones, with values of -6.45 and 6.61, respectively. my_ones is the number of boards that only have 1 of your stones. his_ones reflects the opponents'. This emphasizes the huge disadvantage you're put in if you only have 1 stone in a board. Not only are you 1 capture away from losing, but your options get severely limited.

It's interesting to see the disparity between my_twos and my_ones. While my_ones weights negatively, my_twos weights fairly positive. Perhaps, it's okay to sacrifice some stones for the sake of aggression as long as you don't put yourself in a position where one of the stones can be captured.

Another interesting analysis is the positive weight for my_offensive_stones (3.75) and negative weight for my_defensive_stones (-2.87). This means that the AI found more success being aggressive during training, rather than play conservatively. So, perhaps it can be a good strategy to focus on the opponent's homeboards stones rather than on your own.

Further inspection of these values can lead to some interesting strategy considerations and sheds lights on the priorities of the AI.

Analysis of the AI's The winner is the Minimax with trained weights. It had a record of 81 wins, 2 losses and 2 draws. Monte Carlo EPT and minimax with initial weights also prove to be strong agents, showing a positive winrate against the Human player, Monte Carlo and Greedy Player. Monte Carlo was the weakest agent, because of the sudden-death problem we outlined earlier. Due to the difficulty of finding a goal state, Monte Carlo failed to generate enough simulations to back up a strong move. Monte Carlo EPT shows a significant improvement to the algorithm.

The tournament results demonstrate the agents developed are strong enough to beat players in the novice to intermediate range. However, we did not explore different settings combinations that might lead to better AI performance. Our main focus was to satisfy the 1 minute tournament constraint. With further optimizations and hyperparameter tuning, we could get stronger AI's.

6 Conclusion and Future Work

After exploration of various game playing algorithms and evaluation functions in the context of Shobu, we ended up with two strong agents that are capable of beating amateur to intermediate level players. These are Monte Carlo early playout termination and depth-limited breadth-limited alpha-beta pruning. More importantly, exploration of these algorithms will hopefully open up the path to trying new improved AI strategies and motivate developers to outperform our AI and other implementations that may arise.

It's difficult to objectively gauge the strength of the Minimax agent when compared to other human players. It can definitely become stronger, but it'd be informative to test how strong it currently is and see how much effort needs to be put into its improvement. There's no need to start applying deep learning methods if the current AIs are strong enough to beat human players. Work will be put into a front-end that allows other players to easily play against our AI, where statistics will be recorded.

The trained weights of our evaluation function have opened many questions about optimal strategy in Shobu. For example, developing early offensive stones in the opponent's homeboards seem to have been weighed more heavily than other features. It'll be interesting to see if these features pan out to be accurate representations of efficient Shobu strategy, or they're merely an early step in Shobu's metagame evolution.

References

[1] F.-h. Hsu, IBM's Deep Blue chess grandmaster chips, IEEE Micro (MarchApril 1999) 7081.
[2] Silver, D. et al. *Mastering the game of Go with deep neural networks and tree search*. Nature 529, 484489 (2016).
[3] Vranas, M. and Sadjak, J. Shobu. *Board Game, Smirk Laughter Games*, Sandy Hook, CT (2019).
[4] Allis, L. V. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Univ. Limburg, Maastricht, The Netherlands (1994).
[5] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs, NJ, 1995.
[6] Elnaggar, A.A., Aziem, M.A., Gadallah, M., El-Deeb, H.: *A comparative study of game tree searching methods*. International Journal of Advanced Computer Science and Applications (IJACSA) 5(5) (2014).
[7] Tesauro, G. (1995). *Temporal difference learning and TD-Gammon*. Communications of the ACM, 38 (3), 58-67.
[8] Y. Osaki, K. Shibahara, Y. Tajima, and Y. Kotani, *An Othello Evaluation Function Based on Temporal Difference Learning using Probability of Winning* in Proc. IEEE Conf.
[9] Sutton, R., Barto, A. (1998). *Reinforcement learning, an introduction*. Cambridge: MIT Press/Bradford Books.
[10] Yap, E. Shobu AI (2019). https://github.umn.edu/YAP00014/shobu-ai.
[11] Bravender, D. Man In The Table (Information Set) Monte Carlo Tree Search (2016). https://github.com/dbravender/mittmcts.
[12] Walker, N. True Online TD() (2014). https://github.com/EllaBot/true-online-td-lambda.
[13] R. Lorentz, "*Early Playout Termination in MCTS*," in The 14th Conference on Advances in Computer Games (ACG2015), Leiden, The Netherlands, 2015.
[14] Browne, C. et al. *A survey of Monte-Carlo tree search methods*. IEEE Trans. Comput. Intell. AI in Games 4, 143 (2012).