Alphalone: AI Agent for Abalone

Dunia Hakim
Department of Computer Science
Stanford University
dunia@stanford.edu

Yulou Zhou
Department of Linguistics
Stanford University
ylzhou@stanford.edu

Abstract

Our project is an AI Agent for the board game of Abalone. We modeled the game as a Markov decision process (MDP) and used depth-limited alpha-beta learning with state evaluations, of which the optimal weights are obtained using TD learning. To limit the computation viable on our personal computers, we tested our agents on smaller boards against each other, the baseline, and human players, to evaluate and select the best model. The code for our project can be found at `https://github.com/duniahakim/AI-Abalone`

1 Task Definition

Abalone is a two-player board game with a state-space complexity of approximately 6.5×10^{23}, between those of chess and Go [2]. The rules of the game can be found on its Wikipedia page [8]. Our project is implementing an AI Agent for the game of Abalone. Given a state and a player of the game abalone, out agent will give its best estimate of an optimal move. We will evaluate the agent using adversarial evaluations, including competition between model candidates and playing against human players.

2 Game Infrastructure

2.1 Game model

The game model is a class that consists of instance variables that stores information such as the maximum number of rounds, the integer values for each kind of marbles, and the tuple representation of moving directions. The game model itself does not have any instance variable that stores the progress of the game.

The game model contains a series of functions that take a state as an input, and outputs information such as possible actions, next states, and the utility. Winning or losing does not have infinite magnitude of utility, unlike typical game models, because infinite utility disturbs TD learning.

Compare to other game models that we have seen in class, we added two special functions, the features function and the evaluation function. The feature function takes in a game state, extract features from the state, and output a dictionary that contains these features. The evaluation function takes in a state and a weight dictionary, extra features of the state by calling the feature function, and calculate the dot product between the weight dictionary and the feature dictionary. The dot product represents an estimation of the utility of the state, or how "good" a state is.

2.2 State representation

Game board representation is crucial to the design of our model. In our abstraction, a game state of Abalone includes a representation of the game board using a hexagonal grid with a triaxial cube coordinate. Patel [1] comprehensively introduced the design of the hexagonal grid. The center of the grid is $(0, 0, 0)$, and the standard board would have a radius of 5. The three coordinates of every position (x, y, z) in the grid sum up to 0. The hexagonal grid with a triaxial cube coordinate facilitates the representation of single-step movements.

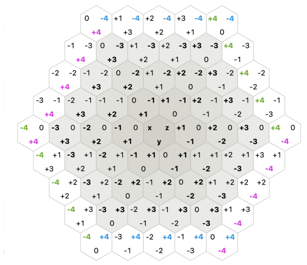

Figure 1: The hexagonal grid for Abalone

A lossless representation of a game state includes the configuration of marbles, the current player, and the current round number. The state stores marble configuration as a dictionary that maps each coordinate with its value, 1 for holding a black marble, -1 for holding a white marble, and 0 for holding no marble. It stores the current player and the number of rounds played each as a integer.

To facilitate judging the progress of the game, the state also stores the numbers of black marbles and white marbles off the grid. In the standard game, a player wins when the number of the opponent marbles off the grid reaches 6.

2.3 Reduced Game

Since the Abalone game has an enormous state space, it might be difficult to run a search algorithm on the standard game board with a radius of 5. Thus, we gave flexibility to our game infrastructure such that it is possible to run Abalone on reduced game boards of radius of 4 and 3, which Connor Settle [3] called "small board" and "mini board" respectively.

Also, to reduce the search space at a given state, we implemented a reduced game model such that only inline moves are allowed. The Wikipedia page introduced the difference between in-line move and broadside move [8].

3 Approach

3.1 Baseline and Oracle

The baseline that we have used for this project and that we have referred to throughout this paper is an agent that we call a "heuristic agent". Given a certain state of the game, this agent basically considers all possible actions that it could take and for each of those actions. It evaluates the successor game state that would result (from using that action) using a simple evaluation function and select the state with the highest score.

To facilitate agent comparison and evaluation, we used evaluation function that we describe below in section 3.3 in the later experiments. The weights that we use in this evaluation function are the optimal weights that we learned from running the adversarial games in section 4.1. Notice that the weights are different depending on whether the baseline agent is playing as the white player or the black player.

Because of the recent development of reinforcement learning in game playing, we think that oracle of this game is an AlphaZero-like Monto-Carlo tree search model that is optimized using a deep neural network [7]. For simplicity, we can also think of the oracle as the best human Abalone player.

3.2 Minimax Search

We implemented Minimax search for this model. Minimax algorithm tries to find the optimal policy for the game by trying to maximize the utility of the worst case scenarios. Our initial experiment with the naive Minimax algorithm shows that the Minimax algorithm is unrealistic for our task. As it tries to explore every possible state that the game can reach from the start state, and the game has 6.5×10^{23} states. Thus, it is necessary to reduce the search space.

3.2.1 AlphaBeta Pruning

In order to speed up the search for the optimal policy for our AI agent, we decided to try adding AlphaBeta pruning to our MiniMax Algorithm as the papers from the literature review suggested. This was still not fast enough. So we then moved to depth-limited AlphaBeta pruning, which would now require the use of the evaluation function.

An evaluation function is a measure that estimates the superiority of a particular position at a certain state of the game. Its value depends only on the layout of the game board and the player whose turn is next. It is a static estimation, without taking into account any previous sequence of moves that has been played on the board. In designing our evaluation function, we tried to strike the perfect balance between accuracy (accuracy of the evaluation/estimation of the current state of the game for the agent) and time complexity (the computation time required to compute that estimation). In the next subsection we give more in-depth details about our evaluation function and we describe the features that we decided to use.

3.2.2 Move Ordering

We know that the efficiency of AlphaBeta pruning depends on the order of the game states. When the game states are ordered in the worst possible way, then the time complexity of depth-limited AlphaBeta pruning is $O(b^{2d})$ where d is the maximum depth chosen to explore (per player) and b is (the average or constant) branching factor [9]. If the game states are ordered in the best ordering possible, then the time complexity is $O(b^d)$ [9]. Interestingly, a random ordering would lead to a $O(b^{1.5d})$ time complexity [9]. Therefore, a commonly used technique to make AlphaBeta pruning more efficient is by ordering the game states based on their evaluation using an evaluation function. We used the evaluation function that we describe in section 3.3 and ordered the successors of max nodes by decreased order of their evaluation, and we ordered the successors of min nodes by increasing order of their evaluation. While we did notice slight improvements in how fast our agents played, their speed was still not fast enough, especially since at this point, the only reasonably running agent time-wise was the one using depth of $d = 1$.

3.2.3 Early Stopping with Thresholding

To further reduce the search time, we added early stopping into the searching process of AlphaBeta. We first normalized the weight vectors to unify their magnitude all to 1, then we set two thresholds for the Minimax search. Instead of selecting the maximum value among the worst cases for the player, the model choose the first action of which the value is greater a threshold (we used 0.5 in our implementation). Similarly, when Minimax is calculating the opponent values, instead of minimizing the player, the model choose the first action of which the value is smaller than a threshold (we used 0.3).

Intuitively, early stopping enabled our model to select a "good enough" actions instead of the best action. Although this compromised the ability of the model, it sped up the searching by two times in our preliminary experiments.

3.2.4 Early Stopping with Time Limits

Another strategy to cut search space is to apply time limit to the search and return the current best result at the time limit. Our implementation set the time limit in the recursive function of AlphaBeta pruning. Since the out-most recursive call costs most time, it is effectively setting the time limit for the out-most recursive call.

The advantage of early stopping with time limits compared to that with thresholds is that each step takes around the same time. However, the time limits do not guarantee that the algorithm finds a "good enough" result.

3.3 Features

As explained in the previous subsection, in order to use depth-limited search, we had to come up with an evaluation function that takes the current state of the game and estimates the state's superiority for the particular player whose turn it is. We decided on the features (of the state of the game) that we will use for our evaluation function by coming up with intuitive good heuristics for evaluating a game state, by looking up good strategies for the game that some of the Abalone game-masters use, and by seeing which features the papers from our literature review used. Eventually, here is a list of the features that we decided to implement for our evaluation function and currently use:

3.3.1 Number of Marbles Off Game Board

This consists of two features. One computes the number of black marbles that have already been pushed off the board game. The second computes the number of white marbles that have already pushed off the board game.

3.3.2 Number of Marbles on Game Board's Edge

Given that if a marble is at the edge of the game board then there is a higher chance that it will be pushed off the game board, then we also used the number of marbles on the game board's edge. Once again, this was split into two features, one for the number of black marbles on the edge, and another for the number of white marbles on the edge. Connor Settle [3] also used this feature in his project.

3.3.3 Distance from the Center

Many online sources, including the Abalone (board game) Wikipedia page [8] and an online strategy and tips for Abalone article [6], mention that a key strategy to playing Abalone is keeping your marbles in the center. Therefore, an estimation that we used is the average distance of the marbles from the centers. The way that we computed this is by adding up the absolute value of all of the x, y, and z values for each marble (of a certain color) and then divided that sum by the number of marbles of that color that have not been pushed off the game board. The reason that this works to compute the average position of the marbles from the center is that our axis scheme has the point $(0, 0, 0)$ at its center and the farther the point is from the center then the larger its values (on the x-axis, y-axis, and z-axis) will be. Naturally, we made this into two features. The first is the black marbles' average distance from the center. The second is the white marbles' average distance from the center.

3.3.4 Marble Coherence

This feature was inspired from both the wikipedia page for Abalone Board Game [8] and from the paper by A. Papadopoulos et al. [5]. The idea is that keeping the marbles together makes for both a good defense mechanism (since the more marbles that you are trying to push then the harder it is to push them) and a good attach technique (since you can push multiple marbles against one target). Therefore, we used the estimation of how many marbles are "together". The way we did this is by counting how many neighbors each marble have, and we gave extra points when there are three marbles in a row, and then finally averaging this sum over the number of marbles still on the board. Again, this feature was split into two. One feature was for the number of black marbles "together" and another was for the number of white marbles "together".

3.3.5 Divide and Conquer

Finally, the last estimation measure that we used comes from the strategy expressed in the online strategy and tips for Ablone article [6] as well as the paper by A. Papadopoulos et al. [5]. The idea is that dividing your opponent's marbles by placing your own marbles in their midst helps to keep your marbles safe (since they cannot be pushed) and makes your opponent's marbles more vulnerable to be captured. We calculated that by counting the number the opponent's marbles near our own marbles and then averaging the sum over the number of marbles still on the board. So the first feature was the number of white marbles neighboring black marbles. The second feature was thee number of black marbles that neighbored white marbles.

3.4 Tuning Weights with TD Learning

After unsuccessful attempts to manually test different weights, we tried TD Learning to automatically learn weights. Intuitively, TD Learning tried to learn optimal weights by matching the state evaluation values with the observed utility of the state.

We have mentioned that in order to give TD Learning earlier feedback, we set the utility function of the game model to non-infinity. In a classical game model, a state only has utility when the game ends. Now, the state has a utility if there are marbles off the board. The utility for each state is calculated as

$$N * (numWhiteOff^4 - numBlackOff^4)$$

with N, a scaling integer that can be adjusted according to the feature weights. The purpose of the fourth power was to weight progress closer to the end of the game heavier than the earlier progresses.

3.5 Literature Review

We examined past attempts to implement AI agents for the game. Lemmens [4] and Papadopoulos et al. [5] had very similar approaches. They both used Minimax searching with Alpha-Beta pruning and tried to reduce the search space by using move-ordering and memoization for states. In addition, Papadopoulos et al. both explored techniques for increasing the efficiency of Alpha-Beta Search. The techniques mainly depended on incorporating probabilistic features to the otherwise deterministic Alpha-Beta search.

Our approach borrowed two evaluation functions from Papadopoulos' model, which are marble coherence and divide and conquer. They seemed to boost the performance of the model, because more in-line marbles improve the power to push the opponents marble around.

Besides the similarities, our approach of early stopping is complementary to Papadopoulos' Minimax approach. Papadopoulos' model used depth-4 Minimax searching, which is very computationally expensive even with move ordering. Our implementation of early stopping would significantly reduce the search time. The state memoization (or "transposition table") is not useful in our context, because we mostly perform depth-1 and depth-2 searches, so repetitively computing state values are unlikely.

Both Papadopoulos et al. and Chorus [2] implemented Monte-Carlo Search for Abalone, but both discovered that Monte-Carlo Search operates less efficiently and won less than Minimax with Alpha-Beta. According to Settle, Monte-Carlo Search performs deep and interruptable search, so it is an orthogonal approach to our implementation [3].

4 Results and Error Analysis

To control the run time in a manageable range, we decided to focus on experimenting on radius-3 (mini) and radius-4 (small). With the help of move-ordering and thresholding, a depth-2 AlphaBeta agent with takes less than 1 second to move on a mini board and around one minute on a small board. When we add time limits, the move time depends on the search time we set.

4.1 Testing Against Baseline

We performed comprehensive tests on mini boards with move-ordering and thresholding. After developing an efficient search function, well-approximating evaluation features, and using TD-learning to learn good weights for the evaluation function, we found that our agents are far better than the baseline. The baseline, as we discussed in section 3.1, is an agent that simply evaluates every possible successor state using the best evaluation function we came up with (without any further search). All four of the potential models that we discuss below beat the baseline agent 100% of the times and with an average of 5 moves.

4.2 Testing Using Adversarial Games Among Agents

To differentiate agent performances, we made agents play with each other. We trained four agents with different step sizes (η) for TD learning. The weights for all agents were produced using 100 iterations of TD-learning with the same initial weights for both players and a γ value of 0.2.

Table 1: The percentage of black victory in adversarial games

White / Black	Agent 1 $\eta = 10^{-2}$	Agent 2 $\eta = 10^{-3}$	Agent 3 $\eta = \mathbf{10^{-4}}$	Agent 4 $\eta = 10^{-5}$
Agent 1 $\eta = 10^{-2}$	45% won 45.65 moves	40% won 32.9 moves	0% won 25.2 moves	70% won 18.3 moves
Agent 2 $\eta = 10^{-3}$	50% won 53.1 moves	90% won 29 moves	50% won 17.5 moves	25% won 41.4 moves
Agent 3 $\eta = \mathbf{10^{-4}}$	60% won 30.7 moves	80% won 22.0 moves	10% won 21.3 moves	35% won 36.5 moves
Agent 4 $\eta = 10^{-5}$	60% won 48.0 moves	80% won 22.0 moves	20% won 18.0 moves	20% won 26.0 moves

Table 1 shows the percentage of games won in adversarial games by the agent playing with the black marbles against the agent playing with the white marbles. These percentages were calculated from running 20 games between the two agents on the mini board. The table also shows the average number of moves made in those games. The result shows that Agent 2 performs the most well when operating the black size (moving first), and Agent 3 performs the most well when operating the white size (moving second).

4.3 Testing Against Humans

We also performed human tests. We played 3 times against Agent #2 with us playing as the white player and the agent playing as the black player, and we could not win a single time. The agent won all 3 times with 10 moves on average. We played 4 times against Agent #2 with us playing as the black player and the agent playing as the white player. We could win twice, and the agent won 2 out of 4 times with 12 moves on average.

We tried to perform human test on small board of radius 4, but the agent took half to two minutes to make each move. We only played two games against the best agents, once as black and the other time as white. The agent won both games.

4.4 Testing Time Limited Agents

We performed human tests on the small board of radius 4 using early stopping with a time limit of 10 seconds. We played against the best black agent (Agent 2) and the best white agent (Agent 3) each for 5 games. The black agent beat us in 2 games, and the white agent beat us in 3 games. The performance of the agent substantially worsened.

We evaluated the effects of different time limits on the the mini board. We performed another set of adversarial games, where Agent #2 plays as black and Agent #3 plays as white. We ran each agent with time limited early stopping against another agent without time limits. We played 10 rounds for each combination. Table 2 shows the percentage of games won by the black player. It is easy to see that the time limit had huge impact on the black side. The winning rate from $1/2$ when there is no time limit to 0 when time is limited to 10^{-4} second. It entirely lost its first-move advantage. The performance of the white player, however, was less influenced by the time-limit.

Table 2: The percentage of black victory in time-limited adversarial games

Time limit (s)	Black limited	White limited
None	50%	50%
1	20%	40%
10^{-1}	20%	60%
10^{-2}	10%	30%
10^{-3}	0%	60%
10^{-4}	0%	70%

4.5 Error Analysis

Our entire development process was guided by error analysis. This was especially true during the design process of our evaluation function. For instance, in the beginning, after we implemented depth-limited AlphaBeta pruning and designed our initial evaluation function, we noticed that our agents were making the mistake of leaving some of their marbles on their own, which leaves them in danger of being pushed off the board. This made us look for additional features that we could add to our evaluation function that would make the agents try to keep their marbles together while trying to separate the opponent's marbles. This led us to find and include the features Marble Coherence (section 3.3.4) and Divide and Conquer (section 3.3.5).

Additionally, when running adversarial games between different agents, we constantly ran into the problem of the game getting stuck in a specific loop of game moves. For instance, the black player would make a specific move, let us call it a, then the white player would make another specific move, let us call it b, then the black player would undo its previous move a, and the white player would undo its previous move b, and then that loop would start all over again with black player making move a, white player making move b, and so on. This error is obviously due to the agents' inability to remember previous game states and figure out that the game could loop this way forever. However, since these loop could be extremely long, meaning that instead of a loop of 4 moves, it could be a loop of 8 or even 30 moves, therefore we found that it would be more efficient to simply allow the agents to randomly pick between the two most optimal actions rather than deterministically pick the optimal action. This solution would solve the problem of cyclical game moves without requiring potentially extremely large space to store previous game moves.

As for analyzing the final state of our agents and its errors, it is quite obvious that, without time-limits, the black agents are harder to beat than the white agents. This is probably due to the fact that the black player always has an advantage in Abalone, especially in smaller boards where movement is more restricted, because the black player is the one who makes the first move according to the rules of the game. Therefore, it makes sense that the white agents make more "errors" than the black agents.

As for the results for the time limited agents, shown in section 4.4, our guess is that the white agents were less influenced by the time limit because, according to the weights learned from TD-learning, white agents tend to be more of defensive players whereas black agents tend to be more of offensive players. Being an offensive player typically means putting your own marbles at risk because you are putting them closer to the opponent's marbles. Therefore, putting a time limit on the black agent means that it will still put its marbles at risk without making the best decision possible. This makes the black player a lot more likely to lost every time the time limit is made more strict. On the other hand, while putting a time limit on the white agent will still worsen its performance, it will not have as much to lose because its moves are not very risky and are even mostly defensive at time.

5 Conclusion

Overall, to design a well-performing AI agent for Abalone, we had to go through a long process of development where each step built off of the previous by analysing its errors. The model that we ended up using relied on Minimax along with depth-limited AlphaBeta pruning, move ordering, early threshold stopping, optional time limits, and TD-learning. Fortunately, the final model ended up performing impressively well in our opinion. It was able to beat us in most cases, even when we tried to use its own strategies against it.

Possible future directions that would have been valuable to explore, if we had the resources, include testing our model on the original board size using more computationally powerful computers. This would give us insight to which other features might be helpful to include in our evaluation function. Furthermore, Abalone is known to have various initial setups that professional Abalone players like to use, so it would be valuable to see how the optimal AI agents for various initial setups would differ. Lastly, it would be interesting to try to develop AI agents for Abalone that are based on neural networks, rather than search algorithms, and see how their performance compares to the agents developed in this paper.

References

[1] Amit Patel. Red blob games: Hexagonal grids, 2019. [Online; accessed 11-June-2019].

[2] Pascal Chorus. *Implementing a computer player for abalone using alpha-beta and monte-carlo search.* PhD thesis, Citeseer, 2009.

[3] Connor Settle. Cs 221 p-final, 2018.

[4] NPPM Lemmens. Constructing an abalone game-playing agent. In *Bachelor Conference Knowledge Engineering, Universiteit Maastricht*, 2005.

[5] Athanasios Papadopoulos, Konstantinos Toumpas, Antonios Chrysopoulos, and Pericles A Mitkas. Exploring optimization strategies in board game abalone for alpha-beta search. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 63–70. IEEE, 2012.

[6] PATRICK J. KIGER. How abalone works, 2011. [Online; accessed 11-June-2019].

[7] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[8] Wikipedia contributors. Abalone (board game) — Wikipedia, the free encyclopedia, 2019. [Online; accessed 11-June-2019].

[9] Wikipedia contributors. Alpha–beta pruning — Wikipedia, the free encyclopedia, 2019. [Online; accessed 11-June-2019].