Introduction to Artificial Intelligence

Abalone Project

Yuval Ussishkin
The Hebrew University of Jerusalem
yuval.ussishkin@mail.huji.ac.il

Nir Amzaleg
The Hebrew University of Jerusalem
nirabraham@gmail.com

Tal Avraham Hai
The Hebrew University of Jerusalem
tal.avraham12@gmail.com

Omer Hagage
The Hebrew University of Jerusalem
omer.hagage@mail.huji.ac.il

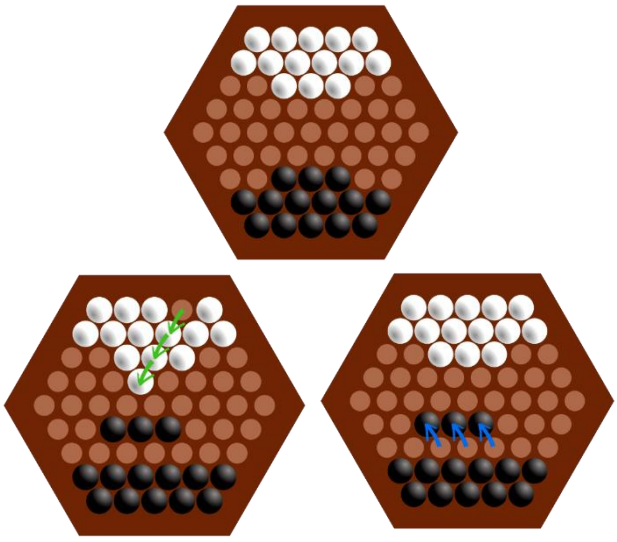
I. Introduction

The problem we chose to explore is Abalone- a strategic board game for two players developed by Michel Lalet and Laurent Levi in 1987.
Players are represented by opposing black and white marbles on a hexagonal board with the objective of pushing six of the opponent's marbles off the edge of the board.

II. Rules

The board consists of 61 circular spaces arranged in a hexagon, five on a side. Each player has 14 marbles that rest in the spaces and are initially arranged as shown below, on the right image. The players take turns with the black marbles moving first. For each move, a player moves a straight line of one, two or three marbles of one colour in one of six directions, one space only. The move can be either *broadside / arrow-like* (parallel to the line of marbles) or *in-line / in a line* (serial in respect to the line of marbles), as illustrated below.

A player can push their opponent's marbles (a "sumito") that are in a line to their own with an in-line move only. They can only push if the pushing line has more marbles than the pushed line (three can push one or two; two can push one). Marbles must be pushed to an empty space (i.e., not blocked by a marble) or off the board. The winner is the first player to push six of the opponent's marbles off the edge of the board.[1]



Starting state, vertical push, horizontal push

III. State complexity

The board contains 61 spaces, each player starts with 14 marbles and could lose 6 marbles and therefore each player will have 9- 14 marbles.
For every combination k, i of the number of marbles that each player can have playing currently, we can choose k spaces out of 61 for the first player, and i spaces out of the 61 – k remaining spaces for the second player. Hence, the number of states is:

$$\sum_{k=9}^{14}\sum_{i=9}^{14}\binom{61}{k}\binom{61-k}{i}$$

Since there is symmetry between the two players, some possibilities are redundant. There are three symmetry axes in the board, therefore we will divide the result by six which results in approximately 10^{24} states.

IV. SOLUTION APPROACH

For this problem, we chose to implement two methodologies that we learned throughout the semester that seemed to be the most appropriate for a complex game with many states:

1. Alpha-Beta agent
2. Q- Learning agent

In addition, we implemented several basic agents which are used for comparison as we wanted to have a basic notion if our agents fare well against trivial opponents:

1. Random agent
2. Minimax agent
3. Random Prioritized agent – this is an agent that divides actions to a few predetermined prioritized types and chooses an action randomly from the highest prioritized type available.

Alpha-Beta Agent
Definitions:

The first method we implemented was an Alpha-Beta agent. An Alpha-Beta agent is based on the Minimax algorithm.

A *minimax algorithm* is a recursive algorithm for choosing the next move in a deterministic game, usually a two-player game. A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. If it is **A**'s turn to move, **A** gives a value to each of their legal moves.

One approach is using a rule that if the result of a move is an immediate win for **A** it is assigned positive infinity and if it is an immediate win for **B**, negative infinity. The value to **A** of any other move is the maximum of the values resulting from each of **B**'s possible replies. For this reason, **A** is called the *maximizing player* and **B** is called the *minimizing player*, hence the name *minimax algorithm*. The above algorithm will assign a value of positive or negative infinity to any position since the value of every position will be the value of some final winning or losing position. Often this is generally only possible at the very end of complicated games, since it is not computationally feasible to look ahead as far as the completion of the game, except towards the end, and instead, positions are given finite values as estimates of the degree of belief that they will lead to a win for one player or another.

This can be extended if we can supply a heuristic evaluation function which gives values to non-final game states without considering all possible following complete sequences. We can then limit the minimax algorithm to look only at a certain number of moves ahead.

The algorithm can be thought of as exploring the nodes of a *game tree*. The number of nodes to be explored usually increases exponentially with the number of turns played. The performance of the naïve minimax algorithm may be improved dramatically, without affecting the result, using alpha–beta pruning.

An *Alpha-Beta pruning agent* stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.[2]

Heuristics:

As abalone is a very complex game with an immense number of states, it is not computationally feasible to expand the entire game tree of abalone. In fact, as there are approximately 60 actions on average that a player can choose from a given state, even a tree with only three levels is not computed efficiently, since it will have about 125000 nodes[3]. Therefore, we need to come up with a heuristic function in order to estimate the value and quality of a given game state.

To come up with a successful heuristic function that will estimate and grade the states correctly as possible, we came up with a few basic properties of states that have a high impact on the chance of winning, and we created the heuristic function as a weighted combination of the following six rules (the weights were determined by trial and error) :

1. Number of marbles of the opponent – as the goal of the game is to eject six opponent marbles, then any ejection is a direct move towards winning.
2. Proximity to the center of the board – increasing the number of marbles of a specific player in the center area of the board will enable the player to have good control over the board and be dominant. Hence, a player

should aspire to have as many marbles close to the center of the board

3. Unified formation – a player should try to have his marbles grouped together as much as possible. The rationale behind this idea is that a group of marbles has a high variety of choices of actions to make, whereas the opponent has less options to push a unified group, as the rules of the game only allow pushing a lesser number of marbles than the pushing marbles.
4. Break opponent's formation – a player should try to have a marble that is surrounded by opponent marbles, thus breaking the opponent's formation. Such a marble cannot be pushed since a player can't push an enemy's marble if its own marble is behind it. Therefore, this marble will greatly decrease the power of the opponent's formed group, and strictly limit the opponent's options of choosing an action.
5. Corner avoidance – while experimenting with the game and playing it, we noticed that a player that has marbles in the corner of the boards has a high tendency of losing the game. The reason behind this is that a marble in a corner can be ejected in three different directions, making this a very vulnerable marble. Because of this, a player should avoid having marbles in the corners, apart from the start position which include marbles in the corners. The side corners are the most dangerous, as the opponent has easy access to them and can easily eject a marble lying there.
6. Margin avoidance – a player should avoid having marbles in the margins of the board if there are opponent marbles in direct proximity. A marble in such a position is in immediate danger of getting ejected and should move out of the danger zone as soon as possible. The situation is even more dangerous if there is more than one opponent marble adjacent to the margin in question.

Another important observation that we made about Abalone is that as opposed to other classical examples of deterministic games, many actions that are possible in Abalone result in a change to the opponent's formation in addition to the player's formation. As such, a good action in Abalone will consist of maximizing the player's compliance with rules 1-6 above, while minimizing the opponent's compliance with those same set of rules.

Thus, we can tune the amount of importance we give to maximizing our player's own compliance as opposed to minimizing the opponent's compliance. This way we can create different strategies, emphasizing different aspects of the game. A strategy that focuses on minimizing the opponent's compliance will be an attacking strategy – it will try to push the opponent aggressively to the margins and out of the center. A strategy that focuses on its own player's compliance will be defensive – it will concentrate on keeping the formation intact and centered and rarely venture into direct attacks at the opponent. Kicking the opponent's marbles out of the board should be incentivized in any strategy – including defensive leaning ones. The reason for this is that a strategy without incentive to eject opponent's marbles might prioritize keeping its player's own form intact rather than ejecting marbles, which is contradicting the aim of the game (ejecting marbles).

Implementation time complexity:

Our goal was to implement the alpha-beta agent up to a recursive depth of two levels, in order to compare the agents with the different depth levels. A major concern was the long runtime of such an agent. In an Abalone game, there are on average approximately 60 legal actions that can be taken from a state, resulting in an upper bound of 3600 nodes to be expanded in an agent of depth level 2, and an upper bound of 216000 nodes to be expanded by an agent of depth level 3, which will take a lot of time to compute. An agent with depth level of 4 will expand o (10^6) nodes [4]. We came up with a couple solutions to this problem:

1. Prioritizing the actions: we created priority categories for the possible actions (the same categories we used for the random prioritized agent). The categories we created are a winning move, an ejection move, a push, an inline move and a sideways move.
2. Creating a dictionary of states: in order to remember some states, we created a dictionary which maps a tuple of a game state and a level in the alpha-beta paradigm. This creates a slow start to the game, as all states encountered are new, but as the game advances reduces some computations to a simple check in the dictionary.

These two simple tweaks reduced the runtime significantly and enabled us to conduct more thorough research.

Results

In order to explore our alpha-beta agent's properties, we simulated a few games that interest us. For a starter, we confirmed that our heuristic indeed makes sense and achieves good results. We ran our alpha-beta agent against a random prioritized agent (as defined above) and against fellow students from the university, and our agent achieved even better results than we expected. The alpha-beta agent achieved a 100 percent success rate against both humans and a random prioritized agent, from depth two. An alpha beta agent with a depth of one won all games against the random prioritized agent, but sometimes failed against a human player.

After ensuring that our agent and our heuristic achieve good results, we started to compare properties of our agents and heuristics. At first, we compared four agents:

- Alpha-beta agent with depth 1
- Alpha-beta agent with depth 2
- Minimax agent with depth 1
- Minimax agent with depth 2

We ran each agent 100 games against the random prioritized agent and checked the following couple properties: average number of turns until winning, and average damage until winning (damage is the average number of marbles the opponent ejected from the board). As shown in *figure 1*, all agents perform roughly the same amount of turns until winning.

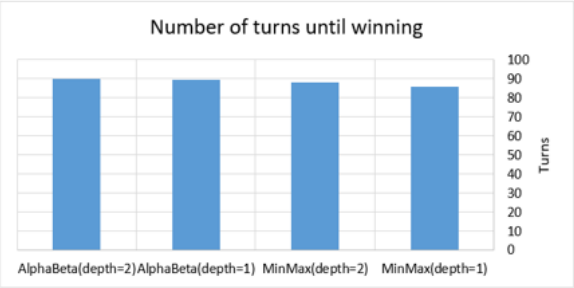


Figure 1

As expected, the alpha-beta and minmax agents of same depth perform similarly, as they are the same agent with the same choosing process and are differentiated only by the runtime. The small differences are explained by the fact that the adversary agent is a random prioritized agent, which chooses some steps at random. We were surprised however by the fact that also the change of depth didn't affect the amount of turns so much, even though two similar agents of different depth behave differently.

As opposed to the number of turns, the depth of the agent indeed affects the number of the agent's marbles ejected by the opponent, as shown in *figure 2*.

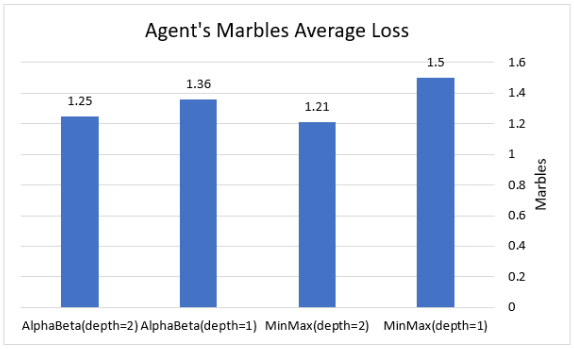


Figure 2

As seen, both the alpha-beta agent and the minmax agent lose less marbles with depth two than with depth one. This can be easily explained by the fact that a search agent with one depth only checks the value achieved by one action and doesn't check what the opponent's reaction might be. Because of this, a one level agent might position marbles in dangerous locations and lose more marbles. A two-level agent will look ahead and play more carefully, losing less marbles in the process.

We also wanted to check the affect of using different heuristics on these two properties. We did this by using our attacking and defensive heuristics and tuning the parameter (as explained above in the heuristics section). For both the attacking and defensive heuristic, we used five different parameters, and checked the affect on the amount of turns and damage, as shown in *figure 3* and *figure 4* (The percent indicates how attacking/defending the heuristic is, with 100 indicating a parameter of 500, 20 percent indicating a parameter of 100, etc.).

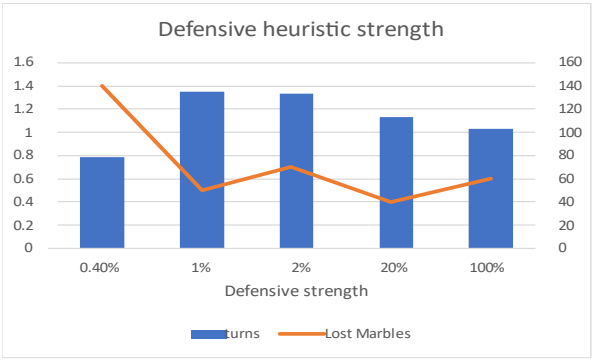


Figure 3

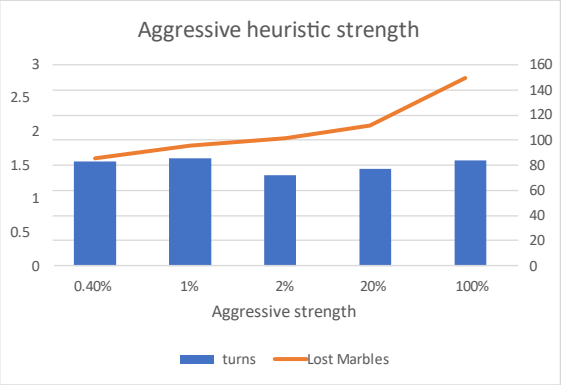


Figure 4

In *figure 3*, defining a defensive parameter (even a low one) causes the agent to lose less marbles on average than in a balanced heuristic. In *figure 4*, increasing the attacking parameter causes the agent to lose more marbles, as much as seven times more on average than some defensive heuristics. The reason behind this is that the defensive heuristics are much more conservative in their gameplay, and don't attempt risky actions, defending their marbles better in the process. On the other hand, increasing the aggressive parameter results in a heuristic that constantly tries to hurt the enemy, resulting in higher vulnerability to ejection.

Another important observation from *figure 3* and *figure 4* is that aggressive heuristics achieve victory on an average of 80 turns. On the other hand, defensive heuristics achieve victory on an average of above 100 turns. The explanation for this result is that a defensive strategy emphasizes keeping the agent's marbles in a united formation, "wasting" turns on building the formation. As opposed to that, an aggressive strategy will emphasize attacking the opponent directly without taking time on keeping the form. The immediate conclusion is that we will most likely prefer an aggressive heuristic in terms of runtime, as long as it doesn't lose too many marbles in the process.

In order to further evaluate the different heuristics, and understand their properties more accurately, we staged a competition designed as a league tournament between different agents. We created ten alpha-beta agents as following:

1. Balanced agent of depth one.
2. Balanced agent of depth two.
3. Agent with a small aggressive parameter of depth one.
4. Agent with a small aggressive parameter of depth two.
5. Agent with a large aggressive parameter of depth one.
6. Agent with a large aggressive parameter of depth two.
7. Agent with a small defensive parameter of depth one.
8. Agent with a small defensive parameter of depth two.
9. Agent with a large defensive parameter of depth one.
10. Agent with a large defensive parameter of depth two.

Each agent played against all the other agents twice, with the difference between the two being the agent who starts the match. A win is awarded three points, a tie one point and a loss isn't rewarded with points.

The results of the league are presented below:

The ABALONE agents league*		
Rank	**Agent**	**Score**
1	Alpha-Beta (Aggressive=10, depth=2)	**40**
2	Alpha-Beta (depth=2)	**37**
3	Alpha-Beta (Aggressive=100, depth=2)	**37**
4	Alpha-Beta (depth=1)	**25**
5	Alpha-Beta (Defensive=10, depth=2)	**22**
6	Alpha-Beta (Defensive=100, depth=1)	**21**
7	Alpha-Beta (Defensive=10, depth=1)	**18**
8	Alpha-Beta (Defensive=100, depth=2)	**16**
9	Alpha-Beta (Aggressive=10, depth=1)	**9**
10	Alpha-Beta (Aggressive=100, depth=1)	**8**
* Each Agent played against all other agents twice, achieved 3 points for winning, 1 point for tie.		

The winner of our league tournament was the lightly aggressive agent of depth two. Also, the balanced agent and highly aggressive agent, both of depth two, fared very well in the tournament.

Those agents with depth of one were at the bottom of the table, therefore we can conclude that the depth parameter has a high impact, since increasing the depth enables the agent to gain a better perspective about the board and to foresee upcoming dangers.

At the middle of the table, we can see the defensive agent, at all depths. The reason for this is that these heuristics prefer to defend their marbles rather than taking actions that could lead them towards winning, by doing so these heuristics settle for mediocrity resulting in many ties. The majority of ties occurred in matchups between defensive agents of different types, as neither agent in those games took actions towards winning.

It can be deduced from the league table that the depth makes a high impact on aggressive agents, as aggressive agents with different depths fared rather differently, as opposed to defensive agents, where changing the depth didn't appear to make such a

difference in the standing in the table. The explanation for this observation is pretty straightforward. Aggressive agents emphasize attacking the opponent, and as a result of this leave their marbles vulnerable. This gives a high significance of looking ahead as much as possible, in order to limit the possibilities of the opponent to eject the agent's marbles. A defensive agent doesn't face this problem and has much less need to plan ahead.

Q-Learning Agent

Definitions:

Reinforcement learning-

Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

Reinforcement learning differs from supervised learning in not needing labelled input/output pairs be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead, the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

Reinforcement learning involves an agent, a set of states , and a set of actions per state. By performing an action , the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward (a numerical score).

The goal of the agent is to maximize its total reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward. This potential reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state.

Q-Learning - is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards.

When q-learning is performed we create what's called a *q-table* or matrix that stores pairs of state and action and we initialize our values to zero. We then update and store our *q-values* after an episode, using the Q function. This q-table becomes a reference table for our agent to select the best action based on the q-value.

The next step is simply for the agent to interact with the environment and make updates to the state action pairs in our q-table.

An agent interacts with the environment in two different ways. The first is to use the q-table as a reference and view all possible actions for a given state. The agent then selects the action based on the max value of those actions. This is known as exploiting since we use the information we have available to us to make a decision.

The second way to take action is to act randomly. This is called exploring. Instead of selecting actions based on the max future reward we select an action at random. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process. You can balance exploration/exploitation using epsilon (ε) and setting the value of how often you want to explore vs exploit.

The Q function is defined as follows:

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t))$$

Where r_t is the reward received when moving from the state s_t to the state s_{t+1} and α is the learning rate ($0 < \alpha \leq 1$). [5]

Reward:

The main challenge involved in implementing a Q-Learning agent is finding the best fitting reward for our problem.

To complete this challenge, we initially set a basic reward value, that only gives rewards to victory or defeat. This approach proved to be much too basic, as this type of reward is only relevant to very few states in our problem. Most states encountered during gameplay are pretty far off from an end-state, and it is very hard to give them a reliable reward based solely on the victory or loss states.

Therefore, we decided to implement a reward that is based on our alpha-beta agent heuristic, which was

proved to be a very effective heuristic. In this way, we give an action a reward that is based on the difference of the heuristic score between the two states the action moves between. This heuristic incorporates both an assessment of the own player's situation and of the opponent's, giving a high grade to a good self-formation and a bad opponent formation. Thus, if the agent succeeds in learning we can potentially earn an agent with similarly good performance to the alpha-beta agent but with far superior runtime.

Results:

Our approach to the Q- Learning problem was cautious. Abalone has a huge state space, and as such it isn't clear that a Q- Learning approach can solve this problem successfully. We started out by training our agent against a random player, for a thousand episodes. After training the agent, we tested it and observed that our agent isn't trained well enough, as it doesn't win most games. The reason for this is that most states our agent encountered while playing a game were states that it didn't encounter during the training phase.

We made a few attempts to improve our agent's learning process. At first, a game against a random agent seemed a bit too trivial for a training process, as the states encountered during the training are generated in a random way. Changing the training opponent to a random prioritized agent is a way to encounter more relevant states during gameplay, as this opponent plays in a standard way and thus our agent will learn states that it has a higher probability of encountering during an actual game. Another change attempted was increasing the training period to ten thousand episodes. An average game in the training period culminated within 130 steps, so this strategy should guarantee the agent learns approximately a million states during the training phase. This is still much less than the actual number of states in Abalone, but as reasoned earlier, if the agent learns a million states which are states that are encountered more frequently than others, this agent might have a shot at solving the game adequately.

Even after the aforementioned improvements, our Q-Learning agent didn't perform well enough. It was defeated by most other agents and by human opponents. However, we observed that our Q-Learning agent achieved fairly acceptable results against a random agent, which reassures us that our agent manages to learn some states, and that the reason for the underwhelming results against complex agents is the vast number of states in Abalone. In order to show that the learning process works as expected, we increased the number of training episodes and checked the impact on the agent's performance. As shown in *Figure 5*, as the number of episodes increases, our agent's percent of games ending in defeat decreases dramatically. It should be noted though that many games ended in a tie, so for this exploration we introduced a limit on the number of turns (300), with the number of marbles ejected being the tiebreaker in case of a tie.

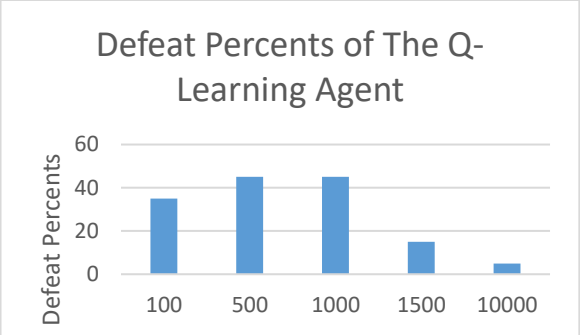


Figure 5

V. CONCLUSION

We had very interesting results for our project. The alpha-beta agent performed very well and gave excellent results. The agent won against every human opponent that we tried it on, including opponents who play regularly. Our agent also won other agents online.

There were many interesting insights that we learned throughout this project. On the one hand, Abalone is a non- trivial game, with many states. On the other hand, we deduced that playing Abalone is best with relatively simple strategies. Our alpha-beta agent with level two gave excellent results, which teaches us that there isn't a need to look ahead many steps while playing Abalone. Since Abalone is a deterministic game, looking deeper into the game tree can always improve an agent's performance, but overall, the level two agent gained excellent results. Our league table shows that it is best to use a mild attacking strategy in order to win the game in the most efficient matter (in terms of marbles lost and turns taken). The defensive heuristic has a much higher rate of tying the game, which could be useful in some situations, for instance in a multi-game tournament if one player takes the lead and just

needs to keep the lead until the end of the tournament.

The Alpha-Beta agent does take a while to calculate the best action, even with depth two. We made a few improvements on the runtime of the agent, which helped considerably. As pointed earlier, a fairly simple strategy with low depth works well for abalone, so the runtime we achieved works well. For other work on different problems with a need for more complex strategies, it is recommended to enhance the runtime of the agent further or find a more efficient agent. For this problem, an alpha-beta agent of depth two gives us the best tradeoff between runtime and results, as it performs excellently and in reasonable runtime. An alpha-beta agent of depth two with our heuristic performs impressively, with very smart tactics. We observed that this agent sometimes sacrifices marbles at the beginning of the game in order to gain a better control over the board, as opposed to the end of the game where our agent keeps the marbles safe at all cost.

On the other hand, the Q- learning approach is too naïve for such a complex game. Even after various improvements and tweaks, the agent couldn't learn enough states in order to play the game properly. In future work on this project with more time on our hands, we will attempt to implement deep Q-Learning, which makes use of a neural network in order to learn the Q- values of each state properly. As this method makes use of a neural network, a deep Q- learning agent will be able to generalize the knowledge it gains during the training process and predict Q- values for unforeseen states encountered during games.

VI. REFERENCES

[1] Abalone: https://en.wikipedia.org/wiki/Abalone_(board_game)

[2] Alpha Beta pruning in AI (2020) Great Learning site: https://www.mygreatlearning.com/blog/alpha-beta-pruning-in-ai/

[3] Playing tips for Abalone (2015) by Wayne Shmittberger: https://abaloneonline.wordpress.com/2015/03/02/playing-tips-for-abalone-by-wayne-schmittberger-2/

[4] Constructing an Abalone Game-Playing Agent (2005): https://project.dke.maastrichtuniversity.nl/games/files/bsc/Lemmens_BSc-paper.pdf

[5] Simple Reinforcement Learning: Q-learning (2019) by Andre Violante: https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56