Exploration of Abalone game-playing agents

Ture Claußen, 202132027, ture.claussen@stud.hs-hannover.de

Dept. of Software and Computer Engineering, Ajou University

Abstract. Perfect information games provide a good environment for artificial agents to navigate in, as they have a clear performance measure for comparison with each other and humans. Their determinism removes some of the engineering problems of agents in the physical world. In the following we implement and compare alpha-beta pruning, Monte Carlo Tree Search and Q-Learning for the game Abalone, to come to a conclusion about their resource-consumption and performance.

Keywords: AI · Alpha-beta · Q-Learning · Abalone · Intelligent Agents

1 Introduction

Abalone is a fairly new game, that was devised in 1987 by Michel Lalet and Laurent Lévi. Nevertheless, with more than four million global sales it has established itself as a classic game [4]. Abalone is a two-player game consisting of a hexagonal board with 61 fields and 14 marbles for black and white respectively. The abstract nature of the game requires the player to plan ahead and find the right strategy in the plethora of moves. The goal is to create an agent that is up to par with human players and moreover, has realistic computational requirements and reacts quickly.

In search of the optimal move it is not possible to expand all of the possible paths the game could take, even for modern computers. Hence, more sophisticated approaches for navigating the state space and evaluating good paths are needed. On the other hand, the game does not have piece-specific rules or large distance moves which reduces the need for a very domain specific knowledge about the game like e.g. for chess to find sensible heuristics.

1.1 Motivation

Overall, this degree of complexity makes the game a good project for the design of a game playing-agent, as it is meant to be an opportunity to apply the fundamental principles and algorithms learned in the class, as opposed to being distracted by the engineering aspects. This matches my personal background on the subject matter, as I have no prior (formal) exposure to the design of artificial intelligence. In addition, this project is only created for the purpose of this class.

Over the course of my current study of applied computer science I gained versatile proficiency in programming and the handling of data which will help implement the algorithms efficiently and provide the empirical foundation for the paper. The project will also be a valuable training for my upcoming bachelor thesis.

1.2 Related work

Considering the existing landscape of papers, there is unquestionably a wide array of papers exploring the application of minimax and alpha-beta pruning on the game of Abalone. Some of the most prominent include:

1. ”Algorithmic fun-abalone” (2002) Considers foundational heuristics for the game and analyzes minimax and its refinements in the form of (heuristic) alpha-beta pruning. Furthermore it sheds light on the performance differences between those. [5]
2. ”A Simple Intelligent Agent for Playing Abalone Game: ABLA” (2004) Implementation of a game-playing agent with minimax, alpha-beta pruning and some custom heuristics. The evaluation of the performance is done by comparing the agent to existing software in the form of ABA-PRO and RandomSoft. [9]
3. ”Constructing an abalone game-playing agent” (2005) Provides a very thorough explanation and analysis of the game's fundamentals, such as the state space, rules and positions. In regards to the alpha-beta pruning it also explains strategies for ordering the nodes and performance concerns. [8]
4. ”Implementing a computer player for abalone using alpha-beta and monte-carlo search” (2009) This master thesis is a very exhaustive analysis of the game, alpha-beta pruning and Monte Carlo tree search, conferring many of the previous results. [7]

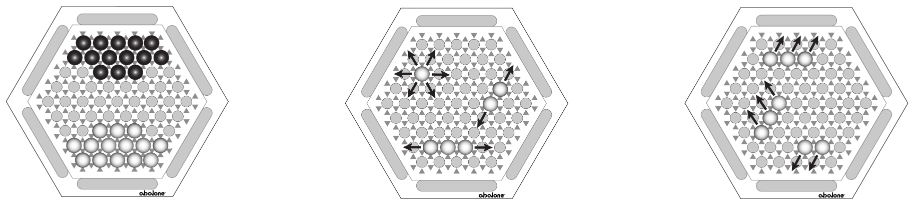
These resources give great insight into the classical approaches, but they are lacking certain qualities:

- Accessible and freely explorable code that underlies the analysis
- Comparison with modern approaches like Q-Learning that might reduce the resource demand on the client side

The proposed project seeks to build upon the given insight to improve upon these missing qualities.

1.3 Rules

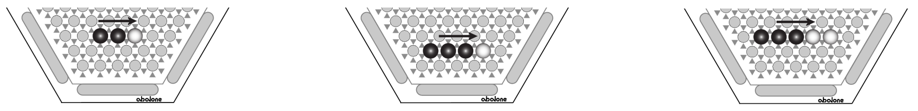
The goal of the game is to push six of the opponent's marbles off the playing field. The game's starting position is depicted in figure 1 (a). One, two, or three adjacent marbles (of the player's own color) may be moved in any of the six possible directions during a player's turn. We differentiate between broadside or ”side-step” moves and ”in-line” moves, depending on how the chain of marbles moves relative to its direction, which is shown in figure 1 (b) and (c).



(a) Starting position (b) "In-line" moves (c) "Side-step" moves

Fig. 1: Basic moves [11]

A move pushing the opponent's marbles is called "sumito" and comes in three variations, as shown by figure 2. Essentially, the player has to push with superior numbers and the opponent's marbles can not be blocked. This is the game mechanic that allows for pushing the marbles out of the game and winning.



(a) "2-push-1" sumito (b) "3-push-1" sumito (c) "3-push-2" sumito

Fig. 2: Sumito positions allow pushing the opponent's marbles [11]

2 Project details

2.1 Agent design

Based on the PEAS framework we can analyze the task environment for the agent. [10, p.107]

Performance measure Win/loss, number of moves, time to deliberate
Environment Digital playing board
Actuators Move marbles, display text to CLI
Sensors Position of marbles

If we look at the environment more closely we see that it is fully observable, two-agent, competitive, sequential, static and discrete.

2.2 Complexity

An important characteristic of a game environment is its complexity, which can be described in two relevant dimensions.

State space complexity The state space is the collection of all possible states the agent can be in. [10, p. 150] For Abalone this means we have to consider all possible board configurations with different numbers of marbles present. Additionally, we would have to correct duplicates that arise from the symmetries of the board. Ignoring this fact the following gives a good upper bound:

$$\sum_{k=8}^{14}\sum_{m=9}^{14}\frac{61!}{k!(61-k)!}\times\frac{(61-k)!}{m!((61-k)-m)!}$$

Game tree complexity The game tree defines the dependencies between board positions (nodes) and moves (edges). First we consider the branching factor (how many moves are possible in one position) of the game tree, which is on average 60. We combine that number with the height of the tree to get the total number of leaves. As the length of a game varies greatly, we use the average length of a game which is 87: 60^{87} [8]

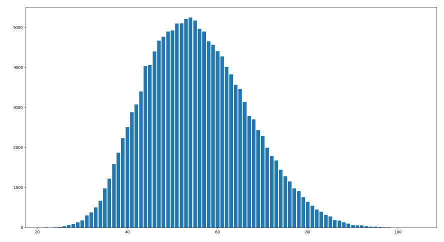


Fig. 3: Counts of moves available for random for random player in 5 games

Putting Abalone's complexity in relation with other popular games, its state space complexity is on the same level as Reversi, whilst its game tree surpasses chess in complexity (c.f. table 1)

Game	state-space complexity (log)	game-tree complexity (log)
Tic-tac-toe	3	5
Reversi	28	58
Chess	46	123
Abalone	24	154
Go	172	360

Table 1: Abalone in comparison with other games [7]

3 Algorithm design

3.1 Heuristics

As the size of the game tree ist very large, the search on the tree usually does not reach terminal leaves that indicate a clear loss or win. Rather one has to evaluate the intermediary result of a given transposition based on a heuristic function. As algorithms like minimax optimize the potential outcome of the next moves based on this function, the heuristics solely determine the performance of such an agent.

This function should judge the positions based on expert knowledge of the game to distinguish good from bad moves.

Adjacency As a majority of marbles is required to push opponent's marbles and conversely an equal amount of marbles is needed to avoid being pushed, it can be assumed that keeping one's marbles grouped together is a good move. The measure of adjacency is calculated by iterating over all marbles and counting the directly neighboring marbles that have the same color. We sum up these counts for each player and measure the difference:

$$\text{adjacency} = n_{\text{self}} - n_{\text{opponent}}$$

This puts the two counts into a relation and produces a negative sign, when the opponent has the upper hand.

Distance to center Marbles that are close to the brink of the board put them into danger of being attacked, wherefore it is generally good to place all of the marbles into the center of the board. For each player's marbles we measure their distance from the center of the board as the smallest amount of moves it would take to reach the center (Manhattan distance). Then again we sum up the distances and weigh them against each other to get the final measure:

$$\text{distance} = n_{\text{self}} - n_{\text{opponent}}$$

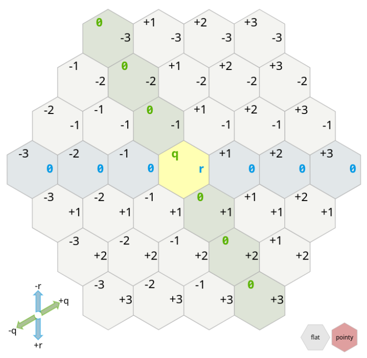
For both measures it is more convenient to represent the internal array indices of the marbles in a different coordinate system that has better mathmatical properties. In case of the hexagonal shape of the board a cube grid or an axial are very suitable.

That way we can obtain the neighbors by just incrementing and decrementing our x-, y- and z-components.

The same goes for calculating the distance which becomes in this space much more intuitive **??**.

Score By comparing the current configuration of the board to following states in the search tree we can obtain a count of how many marbles were lost and how many were won and again weigh those against each other.

$$\text{marbleRatio} = n_{\text{won}} - n_{\text{lost}}$$


(a) Axial grid coordinate system

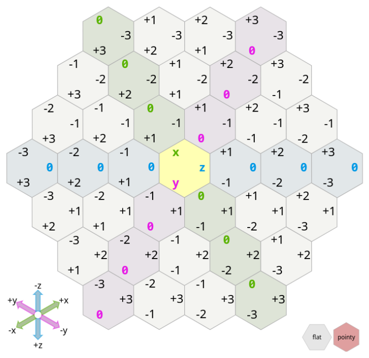

(b) Cube grid coordinate system

Fig. 4: Different options for hexaganonal grids [1]

```
DIRECTIONS = {
  (+1, 0, -1): Direction.NORTH_EAST,
  (+1, -1, 0): Direction.EAST,
  (0, -1, +1): Direction.SOUTH_EAST,
  (-1, 0, +1): Direction.SOUTH_WEST,
  (-1, +1, 0): Direction.WEST,
  (0, 1, -1): Direction.NORTH_WEST
}
```

Fig. 5: Movements mapped to required increments and decrements

Win and Loss Lastly as a more definitive measure we can indicate whether the current state is a terminal state and hence a winning or losing state.

$$\text{winLoss} = \begin{cases} 1 \text{ if game won} \\ -1 \text{ otherwise} \end{cases}$$

3.2 Alpha-beta-pruning agent

Alpha-beta-pruning is an improvement of the minimax algorithm, that tries to eliminate unnecessary traversals down the search tree, which, in the best case, leads to a reduction of moves from $O(b^d)$ to $O(\sqrt{b^d})$. The implementation of the alpha beta agent could be improved in several ways.

The heuristic function is a linear compbination of the above mentioned heuristics was implemented as a linear combination:

Move ordering To increase the likelihood of pruning taking place we need to order the moves, such that for the maximizer the best moves come first and for

```
def distance(self, other: Cube) -> int:
    return max(abs(self.x - other.x), abs(self.y - other.y),
        abs(self.z - other.z))
```

Fig. 6: Code to calculate the distance between two cube coordinates [2]

Heuristic	weight
adjacency	1
distance	-1.5
marbleRatio	100
winLoss	100000

Table 2: Weights for the linear combination

the minimizer vice versa. What constitutes a good move is determined by the heuristic function. That means our move ordering has to be predictive of the resulting heuristic evaluation of the move.

Two different approaches were tested. The first one was based one the following hierarchy:

- Move capturing marble: +3
- Move pushing marble: +1
- Move involving 2/3 marbles: +1/+2

Evaluating this function is computationally much less expensive than calculating the full heuristic function so this approach was tried first (Evaluation 1). In comparision to the algorithm without any ordering the visited nodes could be reduced drastically. Depending on the composition of the heuristic the node count for this ordering fluctuates significantly. If we compared it to the approach of using the heuristic itself for ordering, we see that this decreases the node count even further (c.f. table 5). A possible explanation for that would be, the more predictive the move ordering is of the final heuristic the less nodes are visited.

Depth	Without ordering	Evaluation 1	Evaluation 2	$\sqrt{b^d}$
1	45	45	45	8
2	1594	304	132	60
3	9755	4971	2423	464
4	457309	94650	6918	3600

Table 3: Nodes visited with/without move ordering and the optimal case

Transposition table Due to the nature of abalone there is multiple ways to reach the same configuration of the board. If we save the final value for a state determined by the algorithm, we can potentially save node visits if we visit that node again. To encode the board configurations efficiently, Zobrist Hashing [3] was used. We hold a table with 9 x 9 entries (only 61 of 81 used) and within cell we hold another 2 cells for the distinct game pieces (black and white marbles). In each cell we store a 64 bit random string.

For each marble on the board we use the indices of the positions on the board to retrieve a bitstring from our table and connect them with XOR to retrieve out final hash h.

Branch cutting A problem with abalone is that for each configuration there are many possible moves (high branching factor) and many of those possible moves are very bad. If we have a sensible move ordering we might be potentially exclude many of the useless moves to have a faster agent. For this implementation we only include at most the first 30 first moves from the ordered list.

Depth	Without TT	With TT	With TT and cutting	$\sqrt{b^d}$
1	45	45	31	8
2	132	132	90	60
3	2432	2423	1019	464
4	6918	5829	2435	3600

Table 4: Nodes visited with/without transposition table, branch cutting and the optimal case

Further improvement At this point the choice of the library also becomes relevant. Even though the reduction of node visits brings the most noticable difference in time for deliberation, the implementation can make a significant impact as well. There are two components of the game library that are called extremely often and thereby heavily profit from performance improvements.

For one that is the routine to generate all possible moves. For each node expansion this function is called. By forking the game library [6] and make improvements to that function the execution time could be reduced to 10% of its original value. Furthermore, a tradeoff between execution time and memory was made, by storing and updating the current positions of the marbles, instead of iterating the entire board array.

Monte Carlo Search agent

Black player	White player	Marbles lost black	Marbles lost white	avg. time p. move
1	45	45	31	8
2	132	132	90	60
3	2432	2423	1019	464
4	6918	5829	2435	3600

Table 5: Nodes visited with/without transposition table, branch cutting and the optimal case

3.3 Algorithm comparison

4 Conclusion

References

1. Red Blob Games: Hexagonal Grids. https://www.redblobgames.com/grids/hexagons/
2. Ture / abalone. https://gitlab.com/CampFireMan/abalone
3. Zobrist Hashing - Chessprogramming wiki. https://www.chessprogramming.org/Zobrist_Hashing
4. Abalone (board game). https://en.wikipedia.org/w/index.php?title=Abalone_(board_game)&oldid=994557581 (Dec 2020)
5. Aichholzer, O., Aurenhammer, F., Werner, T.: Algorithmic fun-abalone. Special Issue on Foundations of Information Processing of TELEMATIK **1**, 4–6 (2002)
6. campfireman: Campfireman/Abalone-BoAI (Jun 2021)
7. Chorus, P.: Implementing a Computer Player for Abalone Using Alpha-Beta and Monte-Carlo Search. Master Thesis, Citeseer (2009)
8. Lemmens, N.: Constructing an abalone game-playing agent. In: Bachelor Conference Knowledge Engineering, Universiteit Maastricht. Citeseer (2005)
9. Özcan, E., Hulagu, B.: A Simple Intelligent Agent for Playing Abalone Game: ABLA (Jan 2004)
10. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education, Inc, fourth edn. (2021)
11. S.A., A.: Abalone rulebook. https://cdn.1j1ju.com/medias/c2/b0/3a-abalone-rulebook.pdf