HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Mastering the game of Abalone using deep reinforcement-learning and self-play

Ture Claußen

Bachelor thesis in "Applied computer science"

December 12, 2021

Author Ture Claußen
Matriculation number: 1531067
tu.cl@pm.me

First examiner: Prof. Dr. Adrian Pigors
Abteilung Informatik, Fakultät IV
Hochschule Hannover
adrian.pigors@hs-hannover.de

Second examiner: Prof. Dr. Vorname Name
Abteilung Informatik, Fakultät IV
Hochschule Hannover
email-Adresse

Declaration of authorship

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Hannover, December 12, 2021 Signature

Contents

1 Introduction **8**
1.1 Research goals . . . 9

2 Analysis **10**
2.1 Artificial intelligence . . . 10
2.1.1 Rational agent . . . 10
2.1.2 Task environment . . . 10
2.2 Environment . . . 11
2.2.1 Abalone rules . . . 11
2.2.2 PEAS and task properties . . . 12
2.2.3 Abalone complexity . . . 13
2.3 Classic agent . . . 15
2.3.1 Minimax algorithm . . . 15
2.3.2 Heuristic functions . . . 16
2.3.3 Alpha-beta pruning . . . 17
2.3.4 Monte Carlo Tree Search . . . 19
2.4 Learning agent . . . 19
2.4.1 Markov decision process . . . 19
2.4.2 Reinforcement learning . . . 19

3 System architecture **20**
3.1 Software . . . 20
3.1.1 Training framework . . . 20

List of Figures

2.1 The agent-environment interaction loop 11
2.2 Basic moves [S.A] 12
2.3 Sumito positions allow pushing the opponent's marbles [S.A] 12
2.4 Counts of moves available for random for random player in 5 games . . . 14
2.5 Minimax for a small search tree, resulting in an utility value of 1 16
2.6 Our previous example but with alpha beta pruning applied. The greyed out nodes indicate, that these in fact could be pruned from the tree . . . 18

List of Tables

2.1 Abalone in comparison with other games [Cho09] 15

Abstract

Explanation

1 Introduction

Board games are and have been a popular environment to test the capabilities of state of the art artificial intelligence against human opponents. Many board games are widely known making them a tangible measure of performance. The most prominent examples are the games of Chess and Go. For both, machines defeating the current best players has been representative of fundamental progress in computing.

IBM's "Deep Blue" defeated Gary Kasparov in 1996 [Hig17] by utilizing search to look ahead into the game tree and deliberate on the next move. This approach is a prime example for symbolic AI approaches, "good-old-fashioned-AI" ("GOFAI") [Hau85], which rely on logic and search on symbolic representations.

However, these knowledge-based approaches are severely limited by our ability to properly model the problem correctly and exhaustively. For example, in the case of Deep Blue it requires us to encode our knowledge about chess in a heuristic function to evaluate the board. Only then we can search for actions that maximize this function. Problems with large complexity would require tremendous efforts, which just become unfeasable at a certain point. A different approach would be devising (general) methods to learn the necessary domain knowledge from scratch, *tablula rasa*. As Alan Turing put it:

> Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child-brain is something like a note-book as one buys it from the stationers. Rather little mechanism, and lots of blank sheets. [...] Our hope is that there is so little mechanism in the child-brain that something like it can be easily programmed. [TUR50]

The recent success of "AlphaGo" in 2016 against the long-time world-champion Lee Sedol [Dee] in the game Go is a milestones that perfectly demonstrates this shift towards "bottom-up" or subsymbolic methods. [Nil98] The increasing availability in computational power (and data) has enabled two subsymbolic methods to find large success in unclaimed territory such as copmuter vision or natural language processing. Namely those are neural networks and (stochastic) gradient descent. Combined they provide a general function approximator, that can be trained in a process akin to the learning described by Turing.

In the case of Go, designing a powerful heuristic function was deemed not possible for humans. AlphaGo used (deep) neural networks and gradient descent to train a evaluation function based on a large database of expert moves. With the help of Monte Carlo Tree Search they used this function to play against itself and improve further. [SSS+17]

Building on this success DeepMind, the company behind AlphaGo, further improved the architecture. "AlphaGo Zero" and the generalization "AlphaZero" learn, without the help of the database of expert moves and surpassed the performance of AlphaGo significantly. Since then the architecture has been applied to Chess, Shogi and Atari games by removing the last piece of human knowledge in the system: The rules of the game. [SAH+20]

At this point our endeauvor begins, as the purpose of this writing is to apply the methods of AlphaZero to the game of Abalone.

1.1 Research goals

First, let us establish the main research questions that will guide us throughout this thesis.

The goal is to apply the general framework of self-play learning outlined in "Mastering the game of Go without human knowledge" to the board game of abalone. [SSS+17] The original paper gives clear instructions on the theoretical groundwork for the system but omits clear instructions for the implementation. There is no open source code provided.

Sub-goal 1 is to compare classical search based methods to this AlphaZero's deep reinforcement learning based on several criteria such as win/loss ratio, computational requirements, etc.

2 Analysis

Before we move to the nuts and bolts of AlphaZero and our concrete implementation, we should establish a general understanding of the problem. That includes building the necessary theoretical background in artificial intelligence in general, as well as insight into the specialized knowledge such as deep reinforcement learning in particular.

2.1 Artificial intelligence

2.1.1 Rational agent

Stemming from the latin word *agere* meaning "to act", an agent is something that acts. As we expect our agent to take sensible or intelligent actions we further qualify this definition by calling it rational. This means that it acts so as "to achieve the best outcome or, when there is uncertainty, the best expected outcome". [RN21, p. 36]

The agent exists in an environment which it percieves through sensors and it acts takes actions through its actuators. We refer to the content of the sensors output for one observation as *percept*. The cat uses eyes, ears and other organs to percieve the world and its legs, claws and so on to interact with the world. An antonomous car might use radar and cameras for acquiring information and steering and motors for navigation.

Internally our agent might have some buit-in knowledge about the world, such as rules on how the environments works. The *agent function* takes the entire history of percepts observed and this built-in knowledge and maps it an action. A concrete implementation of this abstract function is called *agent program*. The agent program might just be a simple tabular mapping from percepts to actions or could use a complex algorithm with an additional model.

2.1.2 Task environment

As we are trying to build an agent that tries to achieve some specified goal, we can consider our environment as a problem or *task* our agent tries to solve. Putting together

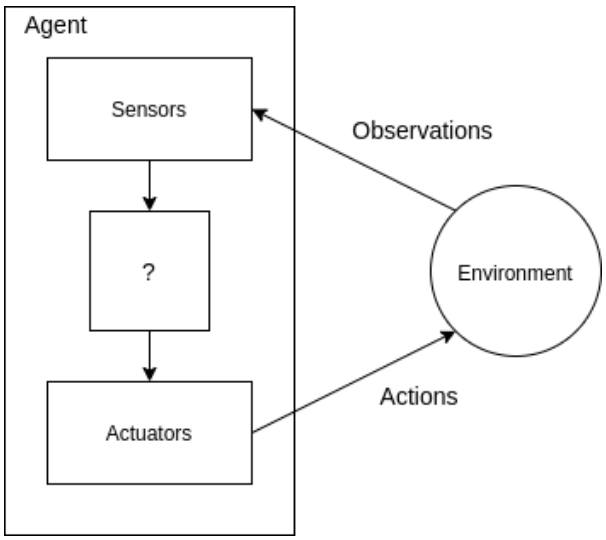


Figure 2.1: The agent-environment interaction loop

both agent and the environment we see a loop of observing, deliberating and finally taking an action as depicted in figure 2.1.

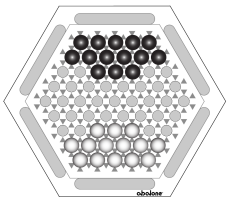
2.2 Environment

Now that we have a general understanding for agents and environments, we can use this knowledge to have a closer look at Abalone. It 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 [noa20]. Abalone is a two-player game consisting of a hexagonal board with 61 fields and 14 marbles for black and white respectively.

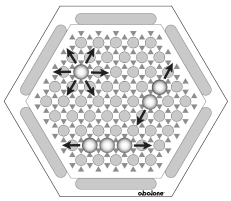
2.2.1 Abalone rules

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 2.2 (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 2.2 (b) and (c).

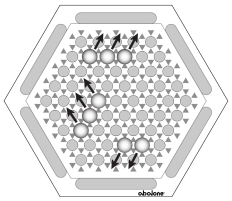
A move pushing the opponent's marbles is called "sumito" and comes in three variations, as shown by figure 2.3. Essentially, the player has to push with superior numbers and



(a) Starting position



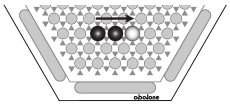
(b) "In-line" moves



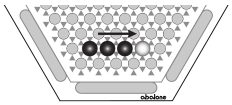
(c) "Side-step" moves

Figure 2.2: Basic moves [S.A]

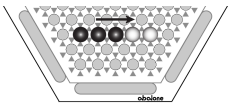
the opponent's marbles can not be blocked. This is the game mechanic that allows for pushing the marbles out of the game and winning.



(a) "2-push-1" sumito



(b) "3-push-1" sumito



(c) "3-push-2" sumito

Figure 2.3: Sumito positions allow pushing the opponent's marbles [S.A]

2.2.2 PEAS and task properties

Based on the PEAS framework we can specify Abalone as a task environment and show the key componentents for the implementation of our agent. [RN21, 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

There are a few categorizations that are extremely helpful for narrowing down potential applicability of different classes of algorithms. A key property is the observability of the environment. If the environment is *fully observable*, the sensors detect all the information that is in any way relevant for taking an action. Conversly if not all information can be oberserved we call it *partial observability*. For example in poker the other players' cards and the upcoming cards cannot be seen but are highly relevant to the agents actions. As the current board state of Abalone fully comprises all information necessary to make a move, we can classify it as fully observable.

The values the state of the environment and time can be categorized into discrete and continuous. The autonomous vehicle for instance is dealing with continous time and also continous states. The speeds of the car take a smooth range of real values and time can be meaningfully split into increasingly small intervals. However, Abalone is entirely discrete. The set of all states is a finite collection of all (legal) permutations of the board and the marbles. Time progresses on the basis of turns.

The actions that the agents takes might also be *non-deterministic*. When dealing with systems of high complexity the next state might not only depend on the previous state and the action taken. There might be other car drivers taking unexpected actions or a comet hitting our car. In Abalone none of these issues arise as it is deterministic.

Further expanding on the passage of time we have to take into account if actions have consequences for future states. If each combination of perecpt and action is independent of each other we call it *episodic* and *sequential* otherwise. If we had to classify a production line of circuit boards as either defective or functional, it would be an episodic environment. The classification of an individual board does not matter for the next one. In the case of Abalone moves taken have long drawn out consequences for later stages of the game.

Another aspect of time is whether the environment changes while the agent takes time to deliberate on the next move. In a *dynamic* environment like the autonomous vehicle operates in, the environment changes continously. In the time the car decides whether to right, to avoid collision with a wall, this decision might have already become obsolete. As any turn based game, Abalone is a *static* environment, as the board only changes after a move is made.

Lastly, an additional dimension to consider is the number of agents involved. The classification of circuit boards only involves one agent whereas Abalone is a *multi-agent* environment. We also have to distinguish whether those multiple agents compete for the performance measure. In Abalone the one players win is the other players loss. In contrast, the other vehicles apart from our autonomous vehicle all profit when it avoids a collision and vice versa. Therefore, they cooperate.

Summing this up **Abalone is a fully observable, deterministic, two-agent, competitive, sequential, static and discrete environment**. Another popular term for this type of environment is a *perfect information zero-sum game*.

2.2.3 Abalone complexity

As Abalone has a finite amount of discrete states, we can make precise statements about its complexity, which can be described in two relevant dimensions.

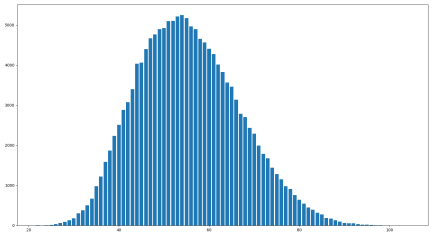


Figure 2.4: Counts of moves available for random for random player in 5 games

State space complexity The state space is the set of all possible states the environment can be in.[RN21, 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 remove duplicates that arise from the symmetries of the board. In the case of abalone we have 6 rotations and 6 axes we can mirror the board on. The following formula gives us 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 all transitions between board positions (nodes) through moves (edges). The *search tree* is potentially a subset of the game tree, if not all paths are visited. In the case of Abalone the game tree is unbound and has an infinite height as actions might be taken repeatedly forming loops. To get a measure of the complexity the number of nodes in a tree is given by:

$$b^d$$

First we consider the branching factor b, or the number of possible moves for any given state. We can only approximate this, as this number greatly varies between different states. On average abalone has $b = 60$ possible moves per state as measured in figure 2.4. The depth d of the tree depends the number of turns per game. Looking at the average again a game takes in the region of $d = 87$ turns, giving us a total of 60^{87} nodes. To be precise this is the complexity of an average search tree not the game tree, as mentioned above. [Lem05]

As those numbers in isolation are hard to grasp it is useful to put 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 2.1)

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 2.1: Abalone in comparison with other games [Cho09]

2.3 Classic agent

Now that we have an intricate understanding of the environment for our agent we have also narrowed down the choice of algorithms we can employ. In general, perfect information games can be solved by adverserial search algorithms. That means in theory we can find the optimal solution by traversing the entire game tree to terminal states (game ending state). The theory behind this type of agent was already laid out as early as 1945 by Konrad Zuse's program generating legal chess moves [KP80], but was described most comprehensively by Claude Shannon in 1950 in "rogramming a Computer for Playing Chess". [Sha50]

2.3.1 Minimax algorithm

Minimax assumes two roles: The minimizer and the maximizer. We start the search from the current board state as the role of the maximizer and then alternate between the two. The result of the minimax search gives us the maximum utility for the given state, assuming both players behave optimally. Let us define the functions

- utility(s, p) returns the utility or the payoff for the terminal state s seen from the perspective of player p. In the case of abalone this might be -1, 0 and 1 for a loss, draw and a win.
- is-terminal(s) returns whether the given state s is a terminal state or not
- to-move(s) returns the current player for the state s
- result(s, a) returns the resulting state if in state s and taking action a

such that we can define minimax recursively as:

$$\text{minimax}(s) = \begin{cases} \text{utility}(s, max) & \text{is-terminal}(s) \\ \max_{a \in \text{actions}(s)} \text{minimax}(\text{result}(s, a)) & \text{to-move}(s) == max \\ \min_{a \in \text{actions}(s)} \text{minimax}(\text{result}(s, a)) & \text{to-move}(s) == min \end{cases}$$

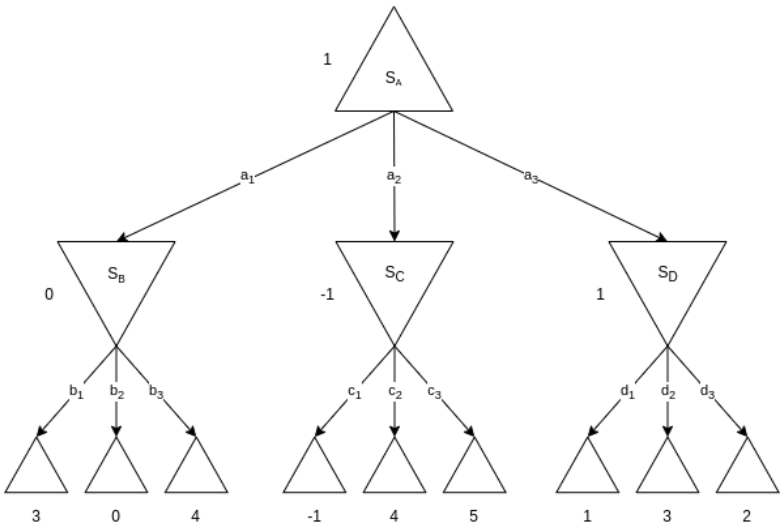


Figure 2.5: Minimax for a small search tree, resulting in an utility value of 1

Putting this together we can see in figure 2.5 a graphical representation of the search tree for an abstract example. The algorithm traverses down to a leaf node, evaluates its utility and passes the value back up to the parent node. Depending on whether it is a minimizer or a maximizer it chooses the smallest or the largest value passed up its children. This value again is passed up to the parent until we reach the parent node which is always a maximizer, yielding us the maximum utility we can achieve given our opponent plays optimally.

2.3.2 Heuristic functions

As the number of nodes of the game tree gets very large, the search on the tree usually does not reach terminal leaves that indicate a clear loss or win. Our computational resources will get exhausted first. For example minimax has already visited $60^4 = 1.2960.000$ nodes at a depth of $d = 4$ in the case of an average Abalone game.

Therefore, one has to limit the search to a computationally feasable depth and evaluate the intermediary result of a given transposition based on a so called *heuristic function*. This function replaces our previous $utility(s)$ for terminal states and is based on human knowledge. The function should give a precise feedback, on the quality of a state from the perspective of the given player. A sensible function for Abalone might be a linear combination in the form of:

$h(s) = \omega_0 f_0(s) + ... + \omega_n f_n(s)$

With functions f_i calculating different values such as

- 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.
- 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).
- Win and loss: As a more definitive measure we can indicate whether the current state is a terminal state and hence a winning or losing state.
- Etc.

By applying different weights ω_i to the functions f_i we essentially give incentives to the agent to prioritize certain behavior. If the win or loss function returns a value of either -1 or +1, we might combine it with a weight of 10.000 to make sure we choose winning states and avoid losing states above all. Armed with this heuristic function we can find good moves with minimax search even in highly complex state spaces.

However, the problem with heuristic is we need expert knowledge and a lot of empirical testing to find a suitable heuristic. In some cases like with Go, such a heuristic function might not be competitive with even moderate human players. In other cases such as chess this strategy is very powerful. As mentioned in the introduction IBM's Deep Blue could beat the world's best player Gary Kasparov based this heuristic based adverserial search.

2.3.3 Alpha-beta pruning

We can further improve minimax search markedly by using Alpha-beta-pruning. This method tries to eliminate unnecessary traversals down the search tree. In the best case, this leads to a reduction of nodes from $O(b^d)$ to $O(\sqrt{b^d})$.

The order in which we visit nodes in minimax is similar to a graph traversal with depth first search, meaning we descend down until we find a leaf node. This gives us information about the utility of that node and, consequently, part of the tree. Going up the tree we keep an alpha value for the minimum value the maximizer will recieve and a beta value for the maximum value the minimizer will achieve. For instance this lets us know if the minimizer already can choose a move worse than what we can achieve with another move, we not descend further ($\alpha > \beta$).

Looking at the example in figure 2.6 will help us illustrate this principle. Our search revealed that choosing move a_1 will yield us an utility of at least 0. Traversing down

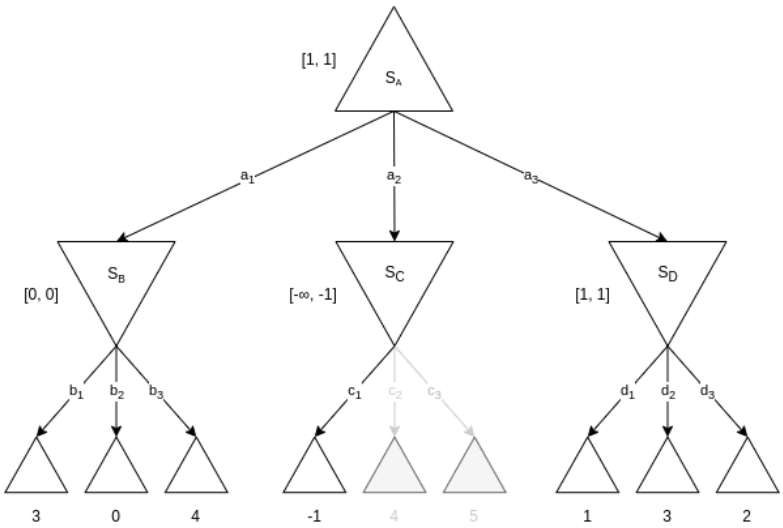


Figure 2.6: Our previous example but with alpha beta pruning applied. The greyed out nodes indicate, that these in fact could be pruned from the tree

move a_2 the first leaf has a utility of -1. Hence, the minimizer will choose a move that is at most -1 which is already worse than the utility of 0. We need not look further at this part of the tree.

The example also shows us a important prerequisite for this method to work. The order in which we expand which node matters decides how much nodes we can prune. Had we visited move c_3 and c_2 first, pruning wouldn't have been possible. The best case of $O(\sqrt{b^d})$ is entirely dependent on this ordering. We could find different ways of ranking the moves:

- *Killer move heuristic* prioritizes that are usually undoubtedly good like taking a marble in Abalone.
- *Iterative deepening* Performs a minimax search only to a depth of one and uses the resulting values to rank the moves. Then searching one level deeper we use this ranking for ordering the moves. Even though there is a lot of redunancy, we make up for this more than enough by pruning much more effectively.

Other improvements to the procedure are thinkable as well. Once we performed a search for a certain state, we can store the resulting utility. If we encounter this position again, because of a different permutation of the move sequence (transposition), we can just look up the state utility in the *transposition table*.

2.3.4 Monte Carlo Tree Search

2.4 Learning agent

2.4.1 Markov decision process

2.4.2 Reinforcement learning

3 System architecture

3.1 Software

3.1.1 Training framework

As there are existing frameworks that have implemented the system described in the AlphaZero paper in a more general and adaptable fashion, it has to be considered building on their foudation.

Bibliography

[Cho09] Pascal Chorus. Implementing a computer player for abalone using alpha-beta and monte-carlo search. Master's thesis, Citeseer, 2009.

[Dee] DeepMind. Match 1 - Google DeepMind Challenge Match: Lee Sedol vs AlphaGo. https://www.youtube.com/watch?v=vFr3K2DORc8&t=7020s.

[Hau85] John Haugeland. *Artificial Intelligence: The Very Idea*. MIT Press, Cambridge, Mass, 1985.

[Hig17] Chris Higgins. A Brief History of Deep Blue, IBM's Chess Computer — Mental Floss. https://web.archive.org/web/20170803130439/https://www.mentalfloss.com/article/503178/brie history-deep-blue-ibms-chess-computer, July 2017.

[KP80] DONALD E. Knuth and LUIS TRABB Pardo. The Early Development of Programming Languages. In N. Metropolis, J. Howlett, and GIAN-CARLO Rota, editors, *A History of Computing in the Twentieth Century*, pages 197–273. Academic Press, San Diego, January 1980.

[Lem05] NPPM Lemmens. Constructing an abalone game-playing agent. In *Bachelor Conference Knowledge Engineering, Universiteit Maastricht*. Citeseer, 2005.

[Nil98] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Elsevier, April 1998.

[noa20] Abalone (board game). https://en.wikipedia.org/w/index.php?title=Abalone_(board_game)&old December 2020.

[RN21] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc, fourth edition, 2021.

[S.A] Abalone S.A. Abalone rulebook. https://cdn.1j1ju.com/medias/c2/b0/3a-abalone-rulebook.pdf.

[SAH+20] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020.

[Sha50] Claude E. Shannon. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, March 1950.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

[TUR50] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, October 1950.