Abalone Game Agent

Hendrik Baacke *i6175085,*
Nick Bast *i6187111,*
Janneke van Baden *i6202983,*
Souzan Abboud *i6182703,*
Fred Bedetse *i6192629,*
Wafaa Aljbawi *i6176322*

A report presented for the findings of Project 2-1

Department of Data Science and Knowledge Engineering Maastricht University
January 2020

***Abstract*—**
This paper states and discusses multiple approaches to design an AI that is able to play the board game Abalone[1]. The algorithms that are used, are: a simple greedy algorithm, an alpha-beta tree search algorithm, a Monte-Carlo tree search algorithm and a greedy algorithm that is optimised using genetic optimisation. Evaluation of different game states is performed using various strategies, which constitute numerical scores that are added together by an evaluation function. The highest score denotes which move needs to be performed in the next game instance. It is investigated which of these approaches is the best at playing Abalone, meaning, which one has the highest win-lose rate in an acceptable time per game as well as moves per game and in which areas (such as time complexity and space complexity) the algorithms perform well.

***Index Terms*—Abalone Game Agent, Heuristic Strategies, Genetic Optimisation, Genetic Algorithm, Alpha-Beta algorithm, Tree Traversal, Monte-Carlo tree search**

I. Introduction

Since civilisation came into existence, board games have been a major source of entertainment for humans. Board games have a central board that depicts a player's position in relation to one or multiple opponents.
Inherent to board games are explicit rules. These, for instance, limit the number of players that a game can have, the number of spaces on a board, the number of possible moves and the limits of what can be done in a particular move. An important assumption about games such as Abalone is that the rules for all participants are the same.
A couple of games rely upon explicit systems. However, many board games contain a component of luck, i.e, information that affects the outcome of the game. Board games generally have an objective that a player aims to accomplish. In zero-sum games, such as Abalone in its 2-player form, the loss of the opponent in turn means the win of the player [1], who therefore strives to outperform the other player using intelligent strategies that counteract the playing style of the opponent.
As a consequence of this, scientists take great interest in developing algorithms in a way that computers can play board games and win against humans or even other computers. Furthermore, there are a variety of board games that have been concocted over the years such as Chess, Go, Backgammon and Abalone. Only one of many attempts, but certainly the most prominent of the past years is described by [2].

A. Problem Definition

The paper focuses firstly on the Abalone game and how to design an AI abalone agent that is successfully playing against a human player in a reasonable time such that the response time of the computer agent is within $(0, 5]$ seconds :

1. Develop the framework for an Abalone agent that can be optimised further with different algorithms.

Secondly, the objective is to compare the different strategies with each other in a normal as well as multi-player set up and to find out the upsides and downsides of the different algorithms in terms of time and space complexity :

2. Compare the approaches with regard to a success metric, for example win-rate and average moves to win the game. Moreover, conduct statistical analysis on the outcome of the experiments in section VIII.

B. Research Questions

Finding an answer for the following research questions is the scope of this report and serves as an orientation when deciding which experiments to conduct and how to set these up.

1. *Which agent has the highest win-rate regarding playing Abalone, greedy or alpha beta?*

1.(2) *Is there a difference in outcome when comparing the win rate with the average amount of moves needed to win a game?*

1.(3) *Which agent, greedy, alpha beta, or Monte Carlo tree search can perform a move the fastest?*

2. *How much impact do different types of playing, using different strategies, have on the win-rate of an algorithm?*

4. *How much -if at all- does the implementation of an evolutionary algorithm improve the existing selection algorithm?*

4.(2) *How does altering the cross-over and mutation process affect the outcome of the optimisation process?*

In order to find an answer to these questions, experiments are conducted and then their results are interpreted in sections VIII and IX .

C. Literature Review

Game AI development is an active field of research because of the intricacies and complexity of the task at hand, while the framework of games serve as a level playing ground (known and equal rules) that enables scientists to develop new approaches at the forefront of AI research that then gradually trickle down to other fields, like with the reinforcement learning framework

[1]Abalone is a registered trademark of Abalone S.A. - France

that is construed in [2]. The foundation for these self-learning reinforcement algorithms is first laid down in the temporal difference backgammon TD(λ) implementation by [3] and implemented for Abalone by [4]. Ideas from these works served as orientation for the development of the tree DST [5] and design of the evaluation component in this work. However, the design of a reinforcement-learning framework similar to [4] is not the intention of this paper. Rather than conducting boundary expanding research with comparably new AI techniques such as R-L learning, this paper aims to get a 'good' result with a well defined and modular framework based on heuristic strategies (that are explicitly selected by humans, compared to using unsupervised learning techniques). Some inspiration in that sense is drawn from [6], [7], [8]. Along with the other works, [6] proposes a linear evaluation component that evaluates a certain non-terminal depth of the game tree and assigns a numerical value based on the strategy values. Non-terminal, since search to the end of the game tree is not possible because of the state-space complexity involved [9], [10]. Research on agents that are specifically trained on or deal with inherent domain knowledge of the game 'Abalone' is furthermore conducted by [8], [11]. The findings and analysis of the game in [7], especially regarding the agent design, the evaluation function and the pruning of the game tree for the alpha-beta selection algorithm (alpha-beta pruning) has influence on the thought and work process for this paper. Some new heuristics are introduced when deemed necessary and beneficial for the play outcome of the agent. Additionally to the agent framework and the basic algorithmic structure already explored by [6], [7], [8], this paper introduces an evolutionary algorithm for optimising the agent as well as exploring a new selection algorithm, that is Monte-Carlo Tree Search. The paper then compares these two new approaches to the conventional implementations with regard to the posed research questions I-B.

D. Report Structure

Firstly, to give context, in section II the basis of the game with the rules of Abalone and an explanation of the different game modes and starting positions is laid out. Section III provides the reader with preliminary information about the methods and data structures that are used for the internal representation of the game such as states of the game and construction of the game tree as well as move ordering. After that, in section IV, the evaluation part of the agent framework is explained in detail.

Section V describes the greedy and alpha-beta algorithms for selecting a specific game state and concludes with insights about the Monte-Carlo tree search method. Then in section VI, the process for the genetic optimisation of the Abalone game agent with greedy selection is highlighted.

In VII, the time complexity for the different methods is compared. Then in section VIII, the experiments that are conducted to find answers for the given research questions are justified and the boundary conditions are explained. This is in turn followed by a description of the results of these experiments with regard to the predefined performance metrics in section IX.

The paper provides a discussion of the results and an interpretation of the obtained data in section X, again in regard to the posed research questions and then ends with a concise summary of the process and suggestion of complementary future research in section XI.

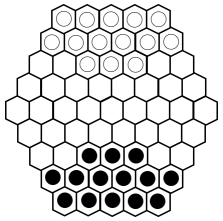
Additional information can be obtained in the appendix.

II. Assumptions About The Game

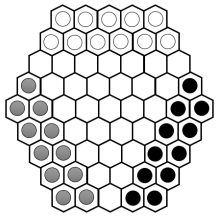
A. Game Rules

Abalone is a game that can be played by two up to four players. A complete guide on the rules for 2-player Abalone, as well as all valid moves can be found in [12]. In this paper, games with only two players and games with three players are discussed.

The objective of the game is to push six marbles in total out of a board consisting out of 61 hexagons. The number of marbles for every player can differ, as it is dependent on the amount of players. In two-players Abalone, both of the players have fourteen marbles at the start of the game. In three-players Abalone, every player has eleven marbles. The initial board state for both, two-players and three-players Abalone, are shown in figures 1a and 1b.



(a) Initial state for a game with two players.



(b) Initial state for a game with three players.

Players can move their own marbles when it's their turn. There are six different directions in which it is possible to move, as shown in figure 2. A player can move up to three of their own marbles, on condition that they are all in the same row and that the marbles

are all connected. So, if there is be a gap between two marbles, those marbles cannot be moved simultaneously.

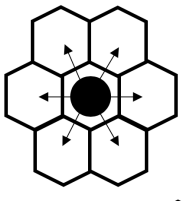


Fig. 2: The six directions in which a marble can move

Marbles can be moved if the destination hexagons are empty, or if it is possible to push marbles out of the hexagons the player wants to move them to. Marbles of the opponent side can be pushed if the number of marbles belonging to the pushing side is superior to the number of marbles being pushed and if none of the pusher's marbles are blocking the push sequence.

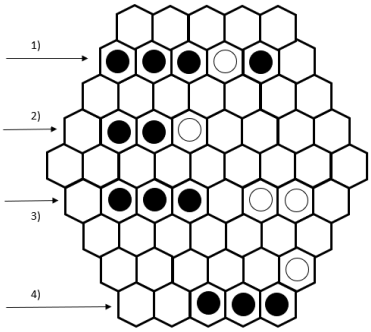


Fig. 3: Possible positions

1) The white marble is sandwiched between the black marbles. Therefore, there is no hexagon free for the white marble to be pushed on.
2) Black has more marbles and there is a free hexagon behind white, so black can push here. Other dominant positions where black is able to push include a 3-1 and 3-2 dominance, the same goes vice-versa.
3) Black and white are separated by a free hexagon. Black can get closer to white, without any pushing.
4) A side stepping column cannot push any marble.

Furthermore, there is one rule that was added to make the game more interesting: It is impossible for a player to inverse his/her last move, independent of the opponent's action. So, if a player regrets the previous move, it is impossible to change this.

The state-space complexity [9] of Abalone is 6.5×10^{26} and the average branching factor is 60, according to [6]. Moreover, the *ply* of the game is 87 [6], thus the expected value for the move count after the game ends is 174.

III. Game Implementation

The game implementation itself is done mostly by using a hash table mapping. Every small hexagon, as shown in figure 1a and figure 1b, is a "tile" on the board. They are all different instances of a separate class called Hexagon, which is a Polygon of a predetermined size (depending on the size of the game), potentially storing a player's marble. Every hexagon has its own fixed location, and is linked to a code, which is used as a key in the hash table. This certain code consists out of a letter and a number, as shown in figure 4.

The hash table contains all the keys of the hexagons in the board. Whenever a code that is in the board gets called, the hexagon is found. This way, the marble can be retrieved using this code as well, as these are stored in hexagons. Marbles are ellipses that contain the player number of the player they belong to. Furthermore, they contain a code in a similar way as the hexagons. It is used to determine their location, and in which hexagon they are stored. Whenever the code inside of a marble gets changed, its position will be changed automatically as well. If a marble and a hexagon have the same code, it means that the marble is inside of that hexagon; it is at this certain position.

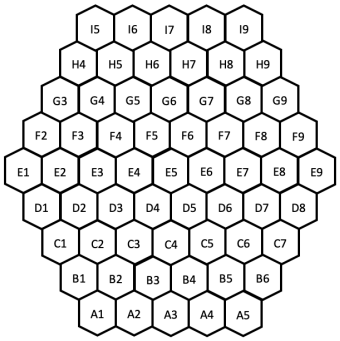


Fig. 4: The board and codes used

A. Game Moves

A move is performed by firstly selecting the marbles that need to be moved. In case the player wants to select marbles, there are a few conditions, such as the marbles being in the same row and the marbles being adjacent, that must be respected. These conditions get checked before a marble gets selected, as the selection needs to be reset when they do not hold.

When a selection is completed, and either one, two or three marbles are selected, a hexagon to be moved to

needs to be selected. This hexagon needs to be adjacent to the first marble of the selection. After the selection process is finished, a check is performed to determine whether a move is valid or not. If the move is not valid, as is explained in the game rules section, then it cannot be performed. In that case, the selection process needs to start again.

B. Game Tree

The tree is implemented using a standard tree implementation; [13], [5] it contains a root node and all its children nodes. Furthermore, the data stored in these nodes could be of any type, although in the tree used in the game, it stores a certain game state.

A game state consists of different components. The most important, which is used most in the evaluation function, is the board after a certain move is performed. In the section about game implementation, the way the board is stored is described: Using a hash table in a normal implementation [5]. Therefore, a hash table is also stored in a state.

Moreover, the performed move to get from the previous game state (in the layer before) to this game state is stored, along with the player who performs the move and the scores of every player after this move. Therefore, as long as the nodes will get added properly, meaning that every valid move gets added as a child node to its parent node, it is possible to perform a tree search. The algorithms to perform tree search are described in section IV. In figure 5, an example of a tree is shown, in which every node represents a game state.

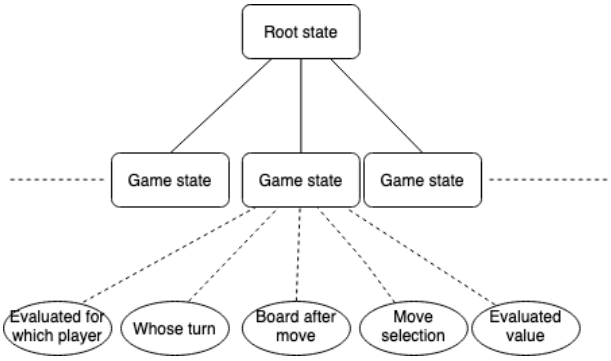


Fig. 5: An example of a tree with one layer.

The way these nodes get added is similar to a brute force algorithm described in [5], [13]. Firstly, a node gets passed as a parameter. After this is done, it tries every move using the board state in the node and the player whose turn it is. If it is a valid move, meaning if the new board state is different from the board state of the node that is being evaluated, then it gets added to its parent node as a child. Therefore, when this method is used, every node that is possible is also in the tree.

To further improve the tree to take up as little memory as possible, the tree gets created only when it is needed for the greedy or alpha-beta search. When it's the AI player's turn, the current board state becomes the root node, and its one (for greedy) layer or two (for alpha-beta) layers are created. After the search has been done and the move has been chosen, the tree gets deleted automatically. This will not be harmful for the tree search techniques, as the deleted tree does not need to be used anymore.

To reduce the size of the tree, and thus space and time complexity, as it takes more time to build the whole tree if the number of layers is larger, the tree sizes are kept small. For the greedy algorithm, one layer needs to be used, which is explained further in the algorithm section.

A description about how alpha-beta tree pruning is implemented for this application is given in section V.

IV. Agent Framework

A. Strategies

In order to develop a 'good' Abalone agent, some key strategies need to be identified according to which the agent can determine the value of a given board state. The strategies are selected heuristically. Their importance and the role that they play as integral part of the evaluation component of the agent is further explained in IV-B.

1) S_1 : Closing Distance:

* Goal of the strategy: Getting as close to the center as possible.
* Calculation: Average euclidean distance between the center of the player's marbles and the center of the board.

2) S_2 : Closing Distance Opponent:

* Goal of this strategy: Pushing opponent away from the center
* Calculation: Average Euclidean distance between the center of the opponent's marbles and the center of the board.

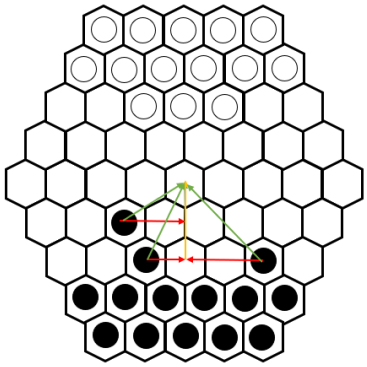


Fig. 6: Calculating Euclidean distance for three marbles; the same process is done for the remaining marbles of the player

3) S_3 : Cohesion:

* Goal of the strategy: Keep the agent's own marbles close to each other.
* Calculation: Sum of number of neighbouring team-mates for each marble of the agent.

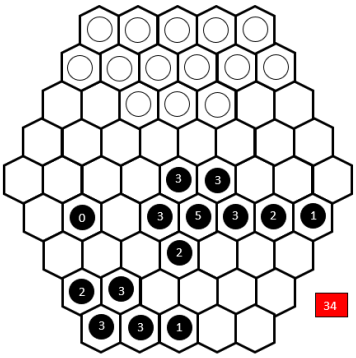


Fig. 7: In this case *Cohesion* returns 34

4) S_4 : Breaking Groups:

* Goal of the strategy: Splitting accumulations of opponent marbles into smaller groups.
* Calculation: Sum of places, where the player has an opponent marble on opposite sides.

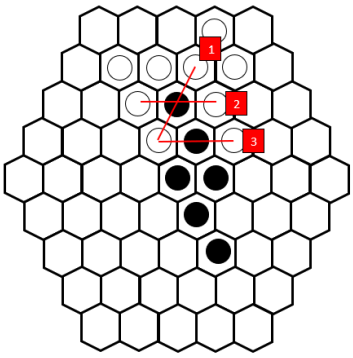


Fig. 8: In this case *Breaking Groups* returns 3

5) S_5 : Strengthened Groups:

* Goal of the strategy: Having as many possibilities to push the opponent
* Calculation: Sum of places where the player can push the opponent.

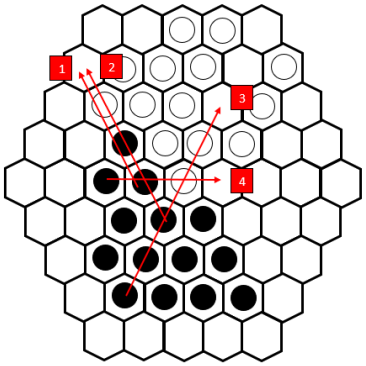


Fig. 9: In this case *Strengthened Groups* returns 4

6) S_6 : Marbles Won:

* Goal of the strategy: Pushing out an opponent marble.
* Calculation: Calculating the difference between the number of opponent marbles between two game states.

7) S_7 : Marbles Lost:

* Goal of the strategy: Keeping as many marbles as possible.
* Calculation: Calculating the difference between the number of own marbles between two game states.

8) S_8 : Marbles In Danger:

* Goal of the strategy: Protecting own marbles from getting pushed out.
* Calculation: Sum of own marbles, which are on the border of the board and that the opponent can push outside.

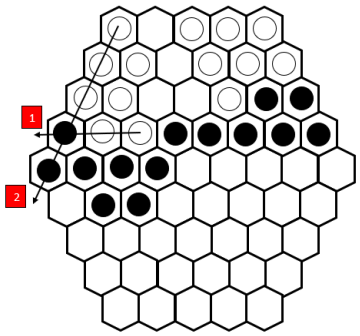


Fig. 10: In this case *Marbles in Danger* returns 2

B. Weight Matrix

To compare different game states, and to see whether one is more desirable to move to than the other, a numerical value needs to be assigned to each state. This is accomplished by the strategies which all return a number given the observed game state.

However, this value can and should be different, sometimes even for the same game state, according to how much emphasis is being put on a certain way of playing. Different ways of playing, such as attacking quickly, defending the own marbles, or breaking their opponent's strategy can be more desirable given a certain score at the time (for example if the player already pushed out 5 marbles of the opponent but only lost 1, then he/she can play relatively aggressively and putting his/her own marbles at risk without risking to loose the entire game). In the Abalone agent framework, this is represented by modes 1 to 5 that have different values for the 8 strategy values from IV-A, resulting in a 5×8 matrix. The mode is determined based on the amount of own marbles, as well as opponent marbles (ie. the score) and the distance from the center. Frameworks with a related foundation are for example proposed by [6], [7], [8] and achieve good results against human players.

1) Definition of agent behaviour declaration: In this paper, a certain type of agent behaviour is sometimes referred to as 'aggressive', 'defensive' or 'neutral' playing style. A formal way to define this nomenclature is given as follows:

- Aggressive AI: An agent whose main goal is to push opponent Marbles out. If it finds an opening, it pushes the opponent marble out, non regarding if it loses a marble of its own in the process.
- Defensive AI: Losing a marble is the biggest punishment, implying that the agent plays without threatening any of its own marbles.
- Neutral AI: Pushing out opponent marbles gets rewarded while losing marbles gets punished in an equal manner

By changing weights (if the weight is higher, there is more emphasis on a strategy), the way an AI is "thinking" is also changed. Thus, various AI agents A_n with different characteristics regarding their behaviour (as explained in the Definition of agent behaviour declaration IV-B1) can be created by altering values in the weight matrix depicted in equation 1.

The values from the strategies get mapped to a value between 0 and 1, with 0 being the minimum value possible and 1 being the maximum value possible. Since the weights in 1 are normalised, the strategy values should be as well to allow for a linear change when altering the corresponding weight in the weight matrix. Normalising is done by dividing each element in one row by the sum of all elements in that row. It follows that the weights per row in matrix (see equation 1) add up to exactly 1.

$$\mathbf{A_n} = \begin{array}{c} \begin{matrix} S_1 & S_3 & S_4 & S_5 & S_6 & S_7 & S_2 & S_8 \end{matrix} \\ \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & \dots & \dots & \dots & \dots & x_{1,8} \\ x_{2,1} & & & & & & & \\ \vdots & & & & & & & \\ \vdots & & & & & & & \\ x_{5,1} & & & & & & & \end{bmatrix} \end{array} \begin{matrix} \\ mode1 \\ . \\ . \\ . \\ mode5 \end{matrix} \tag{1}$$

, where the 5 rows of 1 correspond to the 5 different modes the agent is playing in. The 8 columns depict the weights for each strategy. For all rows 1 to m : $\sum_{n=1}^{8} x_m S_n = 1$

(The strategies in equation 1 are illustrated in the order in which they are implemented in the source code rather than in the order they are explained in section IV-A.)

C. Modes

From which row in equation 1 the AI draws its weights is integral for the play style it uses. The classification into different modes [6] given a specific game state is done heuristically given observational data from plays of human players against human players.

One promising way to improve the agent further without optimising the weights is surely to give it a more fine grained mode mapping. As it is, the mode mapping is dependent on the distance from the center of the own marbles (S_1), the count of the own marbles and the count of the opponent marbles:

Once the AI player left the first mode which is mainly concerned with moving to the middle quickly, it cannot

Mode	S_1	count Own Marbles	count Opp Marbles
1	< 0.75	10-15	10-15
2	≥ 0.75	10-15	10-15
3	≥ 0.75	$> Opp$	$< Own$
5	-	10-15	9
4	-	9	10-15

TABLE I: Mode Mapping

be reentered. Also, in the case when both players have lost 5 marbles (concerning mode 4 and 5) the AI is choosing the more defensive mode 5. How defensive and aggressive play style is defined can be read in section VI.

D. Evaluation function

The evaluation function makes use of different *modes*. Where a mode is a list of weights. These modes are being used so that the AI can play differently depending in which situation it finds itself.

$$Eval(g) = w_{1,m} * S_1(g) + w_{2,m} * S_2(g) + ... + w_{8,m} * S_8(g) \quad |m = 1, .., 5 \tag{2}$$

- where *g* is each gamestate
- *w* depicts the weights
- *S* incorporates the different strategy values
- and *m* denotes the modes which change the given the gamestate characteristics.

V. Selection Component: Methodology

A. Greedy Algorithm

The greedy algorithm uses, as its name suggests, a greedy heuristic [5]. It starts with building a tree with one layer. The initial state is the board as it is at this particular game state of the game. It then creates a root node with this state. Thus, in the layer, there are only moves that can be performed by the player whose turn it is at the moment. While the nodes are being created, their evaluation score will automatically get calculated, using the evaluation function which is described in detail in the previous section IV. When a child is added to the root node, it already contains its evaluation value. Afterwards the children are ordered and the node with the highest score will be chosen as the next game state. The move that leads to this game state is thus the one that is performed.

B. Alpha-Beta Pruning Tree Search

The alpha beta algorithm works with a tree of depth two. Two layers means it can still make use of alpha beta's strength: foreshadowing what can happen in the next move [7], [10]. The reason why alpha beta is chosen instead of normal Min-Max tree search, is that they will both return the same outcome, the same best move, but alpha beta does not visit every node, i.e, prunes subtrees that cannot possibly influence the final outcome. Therefore, the time complexity for alpha beta is better (see VII).

Among the techniques used in move ordering which the alpha-beta pruning can capitalise on, in a sense of a narrower search space [7], [10], [13], [14], the nodes in which more marbles are moved are prioritised in the child list. The probability that moves in which three marbles and two marbles are used to push an opponent's marble (out of the board) is higher and thus have higher expected scores estimated by the agent's evaluation component than the other moves.

For alpha-beta search two layers are used. So alpha-beta still takes both players in account (in case of a two-player game), but does not predict more than one move of the opponent. Alpha-beta pruning would work with as many layers as is desired, but two is used as it is found that the more thorough search of three layers increases the time as well as space complexity in a way that makes it infeasible to evaluate the whole tree. This is for using 'conservative' pruning methods such as transposition tables [10], move ordering and cutting moves below a certain threshold value. It is to be noted that a heuristic approach for pruning the tree like proposed in [7], where the 'smoothness' of the evaluation values (compared to chess) for the game states in Abalone is exploited, is not implemented with this version of alpha-beta.

As stated in section IV, different weight matrices are initiated to create different AI players. This is the case for both the alpha-beta based and greedy variant of the agent; for example: an AI player which has a high weight for keeping marbles together will work differently than an AI player which has a high weight for breaking opponent's groups, even though they work with the same selection algorithm. The way of playing depends on the evaluation component. In VI, where an optimisation method for the weight matrix is proposed, this fact is exploited in order to investigate how much an AI player can be improved, given the previously described static framework .

C. Monte-Carlo Tree Search (MCTS)

While searching for a path in a game tree, there is a chance that the move that would be decided to be the

current best, is not necessarily the most desirable move in the long run [15], [16]. In such situations, the MCTS algorithm is beneficial as it tends to test various moves on a periodic basis during its learning phase, instead of the existing assumed optimum approach [15], [17]. It uses probability instead of the evaluation function that is discussed before.

MCTS's implementation can be described in four steps. There are variations of all of these steps, which can make Monte Carlo more efficient, but the base steps will remain the same. The first one is called selection. In this step, a leaf needs to be found. The tree gets traversed from the root node; it keeps selecting the node in the next layer of the tree which has the biggest value. The value of the nodes gets determined by a the UCB1 [15], Upper Confidence Bound, which can be seen in equation 3 [15].

$$UCB = v_i + C \times \sqrt{\frac{\ln N}{n_i}} \tag{3}$$

- where v_i is the estimated value of node i.
- where n_i represents the total number of the times the node has been visited .
- where N represents the total number of visits of the parent node.
- where C is a constant which is used as a bias parameter. This could be determined heuristically, but the one most commonly is the square root of 2, which is used here as well.

At the beginning of the search, the values n_i, N and v_i are zero (note that if n_i is equal to zero, the UCB is zero as well, which makes sure that many different nodes get explored). If multiple nodes have the same value, a node will be selected randomly, as to make it as fair as possible. Therefore, at the start of the game when every node still has value infinity, the node will be chosen randomly as well.

During traversal, when the tree cannot be searched any further and a leaf node L is reached, the MCTS firstly checks whether the node is a terminal leaf node. If this is not the case, the MCTS goes into the second step, which is called the expansion step. [15] Otherwise, it would back-propagate, which is explained later.

In the expansion step, child nodes get added to the leaf node which was reached. In the version of MCTS used in this paper, every possible node is automatically added to the leaf. So, if there are sixty moves possible after for the leaf node that is being traversed, all of these sixty children will get added to the node.

Afterwards, a child C is chosen according to the selection step. MCTS prioritizes the child with the maximum UCB (3). These steps (selecting and expanding) will continue until a terminal leaf node is reached. Together, this is called the simulation step. The simulation step chooses moves until a result is reached; meaning, until one of the players wins.

After a result is reached, the UCB value of every node needs to be updated. The terminal node and every node that precedes it in the tree get an increased number of visits, every of these values increases by one. Their score can get increased by one as well, if the move performed in the node is done by the winning player. [15]

As there are many different moves possible at every step, reaching the game state in which the game ends can be quite difficult, especially if the node chosen to expand is chosen randomly (which would happen in the first few stages of the game). Therefore, the MCTS needs to have a depth at which the game automatically starts back-propagation. For this research, depth one hundred is used. It would be possible to end a game in this amount of moves. In these case, it is seen as a tie and both the players will get half a point added to their scores. In case it back-propagates after a cutoff, every node gets 0.5 added to their original number of wins, and one added to their number of visits.

The move that will ultimately be performed after Monte Carlo tree search, needs to be one that has already be searched. So, it will be the move from the first layer of nodes that has the highest UCB score that's not infinite [15].

As the Monte Carlo could run until it converges to a full minimax tree search, it needs to be cut off at some point . Therefore, a time limit needs to be set to search the tree [15]. In this implementation, a Monte Carlo tree search of a certain amount of seconds is performed whenever an AI player needs to move. The cutoff means that after this, after the following back-propagation, it will stop traversing the tree.

The tree which was found before is being stored and used again every time an AI player wants to perform a move, except for the branches of the tree which can not be reached anymore, which will be deleted. Thus, every move an AI player does it will perform this search, while keeping the nodes which were already searched before. This way, the information that has already been found will not be lost.

One big problem with the Monte Carlo tree search is its space complexity [17], [16]. As this implementation will add all of a leaf's nodes when the leaf is reached, the tree will get bigger quickly. Therefore, it is possible that the memory of a computer runs out before the end of the game is reached. Further testing about this is done in the experiment section of this paper.

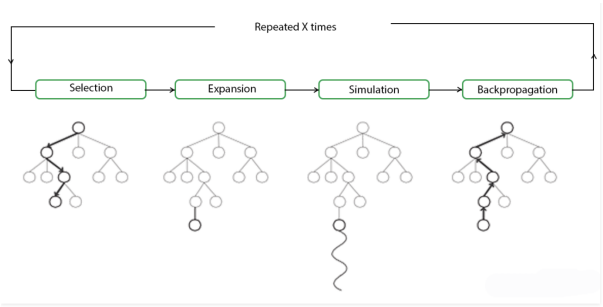


Fig. 11: The fundamental steps of MCTS
https://www.geeksforgeeks.org/ml- monte- carlo- tree- search- mcts/

VI. Evolutionary Optimisation

An interesting question that naturally arises when developing the agent framework and determining the heuristically set baseline weight matrices from section IV is whether and by how much the agent can be improved. A process of genetic optimisation seems suited to change the weights in the weight matrix (equation 1) according to some well-defined fitness function [18], [19] which includes in this case the performance metrics of average moves to end a game and the difference of marbles after the game ended. It is to be expected that due to the restrictions that the framework and scope of this work impose on the yet to be optimised agent, a global optimum, meaning the 'best' possible AI, is an infeasible outcome to achieve, last but not least also under the premise that genetic algorithms have a tendency to converge to a local optimum [18], [20]. This can be mitigated by carefully determining whether an initial population should be "seeded"[21] and by exploring the impact of various mutation rates on the outcome of the evolutionary optimisation [19]. However, exploring the impact of these variables in detail is limited by the computationally expensive fitness function evaluation, which needs to be conducted repeatedly in most cases for several generations [18], [19], [20], [21].

Similarly to most evolutionary optimisation algorithms [18], the process for optimising the weight matrix for the Abalone player involves three steps per generation of instances: Selection, Crossover and lastly Mutation [18], [19].

A. Selection

For the given problem, an initial population pool of size 10 with random weight matrices, where each element is drawn randomly from a $Unif(0, 1)$ is initialized. The elements of each row are normalised by the same process as described in section IV. This means that no initial seed is given which would affect the region (or direction) of optimisation. Whether an initial population should be "seeded" or, as in this case, drawn randomly is open to debate and the answer might change given a specific problem [18], [21].

Each instance of the 10 to-be-optimised matrices/agents (ϵ_n) is then pitted against 3 so called 'trainer' AI's. These training instances stay the same in one generation, so that the performance against them is comparable between the initial population, but change from generation to generation.

The training instances are 3 base seed-AIs: They essentially extend permutations of 3 heuristically set baseline players that display aggressive, neutral and defensive playing (see IV-B1). It is said that the trainer A_t extends the corresponding baseline algorithm A_b, because the heuristic value x_h for each weight in the matrix of A_b is the midpoint of an interval $[x_h - 0.15, x_h + 0.15)$.

$x_n \in [x_h - 0.15, x_h + 0.15)$, where x_n is randomly chosen from the range of values in the interval (which implies that x_h can also $= x_n$). x_n is then the modified value in A_t . The weights in each row of A_t are then normalised. The reason for the permutation process of the training instances is to prevent overfitting of the 3 baseline algorithms which would make the optimised AI an expert against the 3 training instances, but at large cost of generality when dealing with other opponent behaviour that might be for example displayed by human opponents. Genetic algorithms have a strong tendency for overfitting the given learning examples; in-depth analysis and possible workaround solutions (involving special crossover procedures VI-B) for this pitfall are proposed by [22].

Each (ϵ_n) instance plays against one type of trainer AI A_t 4 times and the results are stored in a priority list [5]. In 2 of the 4 games the (ϵ_n) instance starts the game and vice-versa (to account for possible skewing of performance metrics regarding the impact of who starts the game).

After the $10 \cdot 3 \cdot 4 = 120$ games of one generation, 4 candidate instances for the crossover procedure are selected for by their average performance over the given training set regarding the two metrics 'expected move count' and 'marble difference'.

B. Crossover

In order to retrieve the next generation $\epsilon_{1,n}$ of genetically optimised agents, the crossover is an integral step. For this, the weight matrices of the $\binom{4}{2} = 6$ possible combinations of the 4 fittest agents from the current generation are added and the average value (dividing by 2) is taken as the weight value of the newly created

instance. As in all previous steps, the rows are again normalised to add up to 1.

The intermediate generation $\epsilon_{*,n}$ of size 10 is then made up of the 4 fittest agents from the previous generation and the $\binom{4}{2} = 6$ possible combinations of these which need to undergo the mutation step next before formally constituting the 'new' generation $\epsilon_{1,n}$.

C. Mutation

Mutation is an important part in the evolutionary procedure that enables the genetic algorithm in some cases to leave local optima, so that the trade-off between short and long term gain is managed in an adequate and sufficient way given the problem for which an approximation needs to be found [18], [21], [22]. For this implementation of an evolutionary algorithm, the mutation process works like a pointer that goes from the first to the last element of each of the 10 weight matrices from $\epsilon_{*,n}$ and changes the weight with probability 0.1. If a weight is mutated, then the process is similar to the permutation process of the training instances A_t described in VI-A, where the initial weight w_* is set as midpoint of an interval, from which then the new value for the weight is selected randomly. The rows of the matrices are again normalised.

After the mutation process, the creation of the new generation $\epsilon_{1,n}$ is completed. The new generation then plays against new trainers, entering the selection step and thus a new cycle of the genetic algorithm.

In total, 10 generations are created in the above explained manner, each building on the fittest instances from the previous population. The findings for some experiments with this set up are presented in section VIII.

VII. Complexity analysis

To state the complexity of the different algorithms that are being used to perform a move, firstly a few variables need to be denoted.

* n means the number of nodes that are currently in the tree.
* d is the current tree depth.
* b is number of branches of the current tree.
* s is number of marbles that belong to one to one player.
* h is the number of hexagons in the board.

Now, for the greedy algorithm and the alpha beta algorithm, the complexity can be found. As the optimised greedy also works with the greedy algorithm, the complexity of the greedy algorithm is applicable to the optimised as well.

Firstly, the tree needs to be built. The complexity to add one single node is $O(h)$, as the most computationally heavy action is the copying of the hashtable that represents a board. For this action, it will loop through every hexagon on the board once. Valid move checking, so checking whether this move can be performed, has a complexity of $O(1)$, as it is only composed of if-statements, without there being any loops present.

Evaluating these nodes has a complexity of $O(s^7)$, which is due to a strategy which could have up to seven for-loops of size s. Its a rare case, which is only met when multiple different if-cases hold, making its best complexity $\Omega(s^2)$.

Secondly, for the greedy algorithm, the complexity is $O(n)$. As the tree built for greedy only contains one layer, it will be the nodes attached to the root node, meaning all the nodes in the tree.

Lastly, for the alpha beta algorithm, its worst case complexity, when it can not cut off multiple nodes, would be $O(b^d)$, while its best case, when it can cut off most nodes, would be $\Omega(b^{\frac{d}{2}})$.

VIII. Experiments

A. Performance Metrics

To compare the different AI's, some metrics that define success need to be set.

Firstly, the accuracy rate of an AI is chosen to be one measure of success: what is the probability that the AI wins the game against another player. The game is meant to be challenging, and to be able to beat a fairly good-playing human player. Therefore, having a higher accuracy is better.

Secondly, how many moves does an AI need to win a game. If one AI can win a game in less moves than another AI, it could mean this one performs generally better. It make for a more challenging for the player.

Lastly, the time needed to perform one move. As human players would want to be able to actually play the game, having an AI that responds to a move fast will make it more easily playable, as a small delay means that the human does not have to wait for a long time.

B. Experiment 1: Win-rate

In order to determine which selection component in its current implementation yields a better AI player, the greedy and the alpha beta tree search play against each other, in different combinations of their baseline matrices. As the baseline 'defensive', 'aggressive' and 'neutral' evaluation functions are used, whenever the same type of AI with the same selection component

plays, the outcome is the same. It is thus not necessary to play more than one game per combination of AI's.

Therefore, with these games, it is possible to determine which way of playing performs better against which other type. Moreover, when two of the same AI's play against each other, it is possible to see if it offers an advantage to be the player who needs to move first. As also the variants of the greedy are pitted against each other, to observe which of the three performs best.

C. Experiment 2: Moves needed to win

This experiment is concerned with the same games as the previous experiment. The difference is that the number of total moves needed to win the game is measured. So, again, the greedy and the alpha beta tree search play against each other in different combinations of their baseline matrices, as well as the three greedy implementations, measuring the number of moves they need to either win or lose.

Therefore, the experiment offers insight into which AI is fastest in winning a game. Since the expected move count is one of the two performance criterion according to which the genetically optimised AI is selected for as described in VI, it is also useful to know which game involving which AI has the lowest average *ply* and thus the lowest average move count in the end of a game.

D. Experiment 3: Time needed to perform one move

One critical factor is the time it takes to perform one move. As one of the goals is to have a game that can play against a human, it is important that performing a move is done quickly, as mentioned in the performance metric section.

Therefore, the average time needed to perform one move for a type of AI needs to be found. By running multiple moves for greedy, Alpha-Beta tree search and Monte-Carlo tree search, it can be calculated what is the average of the times needed to search for the best move and perform it. As Monte-Carlo cuts off after the first back-propagation that takes place after the ten second mark, it is already known that this algorithm takes longer than ten seconds. However, it can differ how long the algorithm actually takes, as its back-propagation still needs to be done after it is cut off.

The time is measured in milliseconds, and the results are noted like this as well.

E. Experiment 4: Evolutionary algorithm

With this experiment, it can be determined if the final generation resulting from the process described in section VI is always the strongest and best performing by plotting the sum of every move needed per generation and the sum of the marbles scored. This experiment is done on the fifth to seventh time the genetic algorithm was run.

F. Experiment 5: The amount of nodes Monte Carlo tree search traverses and adds in a certain amount of time

There are multiple things that can investigated about Monte Carlo tree search; how many nodes can be stored in total (before the computer runs out of memory), how long it takes to search through and add a certain amount of nodes and whether it makes a difference to let it run for a different amount of seconds if this difference is rather small.

As stated earlier, in the section about Monte Carlo tree search, the amount of nodes that are traversed is decided by the amount of time that is given to train the tree. As a way to test how many nodes it traverses and adds on average during different amount of seconds, the Monte Carlo tree search algorithm is run for different amounts of time. Every time it is noted through how many nodes it back-propagates, giving the number of nodes it traverses. Besides that, it is also noted how many nodes there are added in total.

The times chosen to test it for are: two seconds, five seconds, ten seconds, twenty seconds and forty seconds. The MCTS is run five times for each of these times, and the average results are noted in the result section of this paper. As stated in the Monte Carlo section of this paper, the number of nodes it maximally searches in a row is one hundred, after which it will back-propagate as if it's a tie. If there is an exception, such as a usage of too much memory, this will also be noted. The maximum amount of time an move is given, is two minutes. After this time, it is cut off and the experiment is ended.

G. Experiment 6: Multiplayer experiment

Let three different AI players play against each other: the genetically optimised player is kept constant and plays against all versions of the greedy algorithm. Moreover, the starting sequence for the games are alternating so that inferences about the advantage or disadvantage about the starting positions can be made.

IX. Results

In this section the results of the experiments, explained in the previous section VIII, are presented. Moreover, also graphs and tables illustrating these results are provided.

The abbreviations 'Agg', 'Def', 'Neu' and 'Opt' that are displayed in the result tables, stand for 'Aggressive'-, 'Defensive'-, 'Neutral'- and 'Optimised'-agent ,respectively. The abbreviations reference baseline AI's that are explained in IV-B1, or an optimised AI that is retrieved from the genetic optimisation.

A. Results experiment 1 and 2

In the below tables 2 and 3, outcomes of multiple games and how many moves it took to complete the game are given. Greedy1 is the player who has the first move. The scores are denoted as two numbers, divided by a '-', the first number being the score of the first player, the second number the score of the second player.

Game	Greedy1	Greedy2	Score	Number of Moves
1	Agg	Agg	5-6	183
2	Agg	Def	3-6	145
3	Agg	Neu	6-3	146
4	Agg	Opt	5-6	125
5	Def	Agg	4-6	175
6	Def	Def	2-6	93
7	Def	Neu	6-5	118
8	Def	Opt	2-6	143
9	Neu	Agg	6-4	62
10	Neu	Def	5-6	253
11	Neu	Neu	6-3	44
12	Neu	Opt	1-6	81
13	Opt	Agg	6-4	68
14	Opt	Def	4-6	65
15	Opt	Neu	6-5	92
16	Opt	Opt	6-2	84

TABLE II: Scores and moves needed to win, greedy vs greedy

1) Aggressive greedy as player 1: 1/4 win-rate
2) Aggressive greedy as player 2: 1/2 win-rate
3) Defensive greedy as player 1: 1/4 win-rate
4) Defensive greedy as player 2: 4/4 win-rate
5) Neutral greedy as player 1: 1/2 win-rate
6) Neutral greedy as player 2: 0/4 win-rate
7) Optimized greedy as player 1: 3/4 win-rate
8) Optimized greedy as player 2: 3/4 win-rate

The following table has the same structure as the previous one, except for the fact that the next one is for Greedy against Alpha Beta tree search.

Game	Greedy	Alpha Beta	Score	Number of Moves
1	Agg	Agg	6-0	312
2	Def	Agg	6-1	114
3	Neu	Agg	6-2	246
4	Agg	Def	6-1	158
5	Def	Def	6-1	430
6	Neu	Def	6-0	420
7	Agg	Neu	6-0	280
8	Def	Neu	6-0	354
9	Neu	Neu	6-2	272

TABLE III: Scores and moves needed to win, greedy moving first

1) Greedy win-rate as Player 1: 9/9
2) Average number of moves needed: 287 moves

Game	Alpha Beta	Greedy	Score	Number of Moves
1	Agg	Agg	5-6	147
2	Agg	Def	1-6	139
3	Agg	Neu	2-6	143
4	Def	Aggt	1-6	189
5	Def	Def	1-6	147
6	Def	Neu	0-6	483
7	Neu	Agg	0-6	205
8	Neu	Def	0-6	175
9	Neu	Neu	2-6	119

TABLE IV: Scores and moves needed to win, alpha beta moving first

1) Greedy win-rate as Player 2: 9/9
2) Average number of moves needed: 194 moves

B. Results experiment 3

The different moves performed yield the results as can be seen in the following table. The amount of time given in this table is displayed in milliseconds. The full table, with all the different results that lead up to this average, is given in the appendix.

	Greedy	Alpha Beta	Monte-Carlo
Average	3676	43	13672

TABLE V: Average time needed to perform a move.

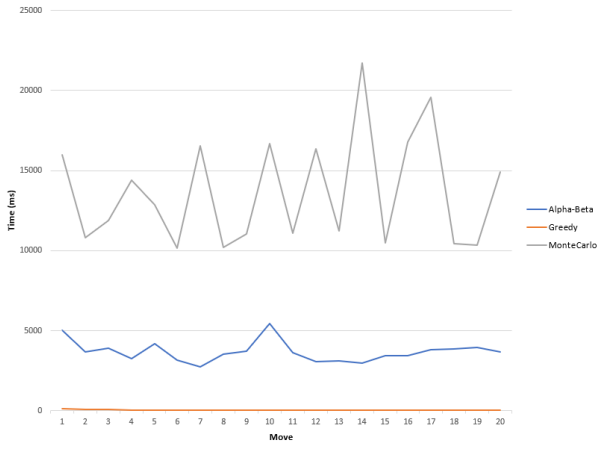


Fig. 12: Time needed

To give more insight into the data which is being displayed above, in figure 13, a graphical representation of the average time is also given.

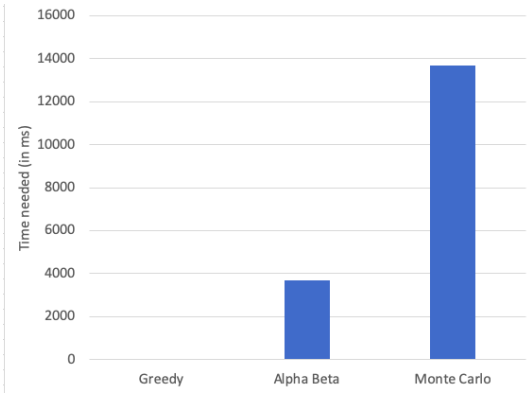


Fig. 13: Time needed to perform one move.

C. Results experiment 4

One data point represents the sum of all the AIs results belonging to the same generation against one trainer. In all the following plots, the data sets represent the results against a mutation of the Neutral Trainer(Blue), Aggressive Trainer(Orange) and Defensive Trainer(Gray).

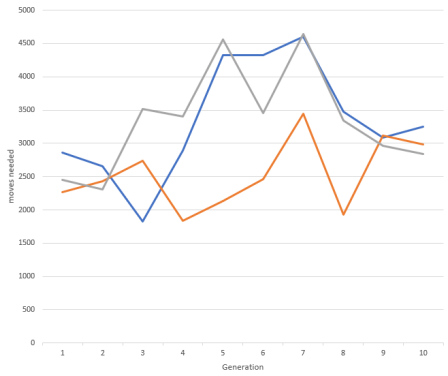


Fig. 14: moves needed for the fifth run of the genetic algorithm

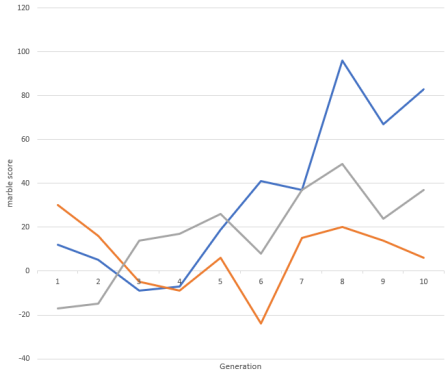


Fig. 15: marbles scored needed for the fifth run of the genetic algorithm

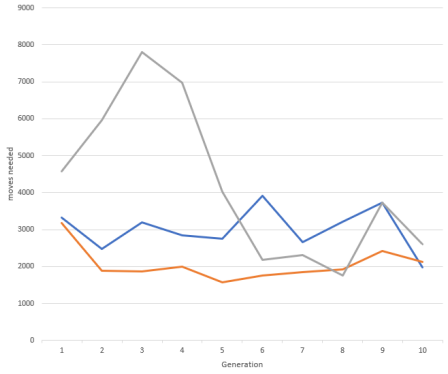


Fig. 16: moves needed for the sixth run of the genetic algorithm

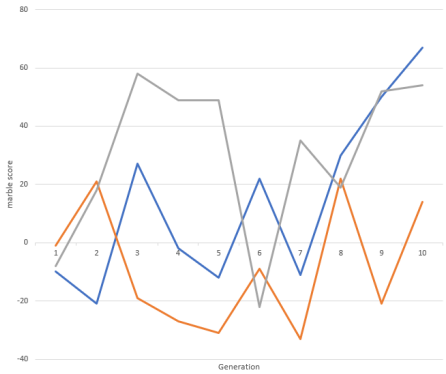


Fig. 17: marbles scored needed for the sixth run of the genetic algorithm

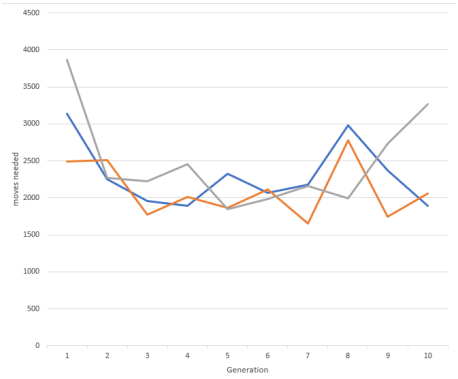


Fig. 18: moves needed for the seventh run of the genetic algorithm

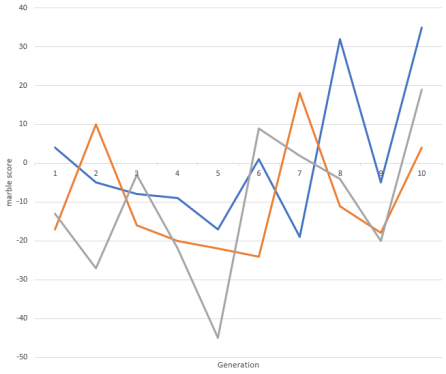


Fig. 19: marbles scored needed for the seventh run of the genetic algorithm

D. Results experiment 5

In table VI, the average time used, average number of nodes back-propagated through and average number of added nodes for each time bound of the Monte Carlo are given. In the appendix full tables based on which the averages are calculated are given.

Second-bound means the amount of seconds it is given (after this, a new selection can not be started anymore), time used is the total time the move used, back-propagated stands for the amount of nodes that is back-propagated through and added nodes means the total of nodes added to the tree.

A '-' in a table means that there were instances in which an ending back-propagation could not be reached.

Second-bound	Time used (ms)	Back-prop.	Added Nodes
2	16469	100	8334
5	20607	100	8192
10	19393	100	8835
20	24915 $\geq$ 120000	100/-	8367/11306
40	$\geq$ 120000	-	11315

TABLE VI: Second-bounds and its average results.

For the 20-second-bound, two moves can not be ended within two minutes. The results of the moves that could be finished are on the left side of the '/', the results of the moves that could not be finished are on the right side.

Note: 3 out of 5 moves performed for the 40-second-bound got an out-of-memory exception, meaning it could not go on. The other two moves for the 40-second-bound still not finished after two minutes, which meant the experiment ended.

E. Results experiment 6

Game	Gr. 1	Gr. 2	Gr. 3	Score	Nr. Moves
1	Agg	Def	Opt	2-4-6	124
2	Agg	Opt	Def	5-6-3	360
3	Def	Agg	Opt	3-2-6	133
4	Def	Opt	Agg	4-4-6	260
5	Opt	Agg	Def	5-4-6	457
6	Opt	Def	Agg	6-3-4	86
7	Agg	Neu	Opt	2-2-6	124
8	Agg	Opt	Neu	4-6-5	135
9	Neu	Agg	Opt	5-2-6	157
10	Neu	Opt	Agg	0-3-6	103
11	Opt	Neu	Agg	5-6-2	189
12	Opt	Agg	Neu	6-4-3	188
13	Def	Neu	Opt	2-4-6	121
14	Def	Opt	Neu	5-6-5	231
15	Neu	Def	Opt	2-3-6	124
16	Neu	Opt	Def	1-6-3	81
17	Opt	Neu	Def	4-2-6	95
18	Opt	Def	Neu	6-3-1	95
19	Neu	Neu	Opt	0-3-6	151
20	Neu	Opt	Neu	3-6-5	147
21	Opt	Neu	Neu	5-1-6	85
22	Agg	Agg	Opt	4-4-6	172
23	Agg	Opt	Agg	4-6-2	135
24	Opt	Agg	Agg	6-2-1	89
25	Def	Def	Opt	3-4-6	409
26	Def	Opt	Def	2-3-6	124
27	Opt	Def	Def	5-4-6	199

TABLE VII: Three player abalone, greedy vs greedy vs greedy

1) Aggressive greedy as player 1: 0/6 win-rate
2) Aggressive greedy as player 2: 0/6 win-rate
3) Aggressive greedy as player 3: 2/6 win-rate
4) Defensive greedy as player 1: 0/6 win-rate
5) Defensive greedy as player 2: 0/6 win-rate
6) Defensive greedy as player 3: 4/6 win-rate
7) Neutral greedy as player 1: 0/6 win-rate
8) Neutral greedy as player 2: 1/6 win-rate
9) Neutral greedy as player 3: 1/6 win-rate
10) Optimized greedy as player 1: 4/9 win-rate
11) Optimized greedy as player 2: 6/9 win-rate
12) Optimized greedy as player 3: 9/9 win-rate

* Average player 1 win-rate: 4/27
* Average player 2 win-rate: 7/27
* Average player 3 win-rate: 16/27

X. Discussion

What can be deduced from the first and second experiment is that the Greedy algorithm is globally performing better than the Alpha-Beta tree search, as the results show the Greedy has an overall 18/18 win-rate against Alpha-Beta. Using these results, the research question inquiring which agent performs better can be answered as followed. Greedy algorithm has an overall considerably higher win-rate than the Alpha-Beta over 18 games, regardless of which modes is chosen for each AI.

Furthermore, the results also denote that being the player who goes either first or second doesn't affect the final score of the game, as the scores in the tables III and IV are equivalent except for game 1.

However, being the player who goes either first or second seems to influence the game in some way. Indeed, the results on average number of moves (287 moves) with Greedy as player 1 is a lot higher than with Greedy as player 2 (194 moves). Hence, it can be argued that starting second in a 2-player abalone game is more advantageous.

The third experiment shows that on average, the fastest algorithm is the greedy one. Although it is slower than greedy, the alpha beta also performs its move rather quickly. The difference between these two though, is still relatively large. Whereas the greedy takes about 43 milliseconds to perform a move, it takes the alpha beta algorithm around 3676 milliseconds.

This can mostly be explained by the fact that for an alpha beta search, as the tree for greedy and alpha beta gets created at every step, more nodes need to be added and evaluated. As it has been stated before in this paper that the average number of moves of one node is 60, it means that for two layers, this would be around 3600.

For the implementation that is used in this paper, performing one move for MCTS takes at least ten seconds. Therefore, it is already clear that this would be a slow AI. The MCTS took longer than these ten, with an average of about 14 seconds. The other two both have an average that falls between the 0 and 5 which are good bounds for an AI to move. Therefore, greedy and alpha beta would be good AI's if a human were to play against it.

For the fourth experiment the question is if the last generation is performing the best. To interpret the plots, regarding the moves needed, it is better if the result is low, because that means the games are finished faster. For the marbles scored it is better to have a high score. In general, the last generation is almost never the best generation against one specific trainer which is interesting to see. Since scoring more marbles is more important than having few moves for winning, let's analyse those plots first. In the fifth run fig. 15, the best scoring generation was 8. For the sixth run fig. 17, the best generation was 10. In the seventh run fig. 19the best generation was 10. For the number of moves, fig. 14, fig. 16 and fig. 18; the sum is mostly converging to the same amount of moves in generation 10. Nonetheless, outliers can always occur

in either plot, because the trainer used is a mutation of the starting trainers. Therefore, it is possible to get a stronger or weaker trainer for a generation.

The research question 4: *How much -if at all- does the implementation of an evolutionary algorithm improve the existing selection algorithm?* can, given the results be answered as follows. The heuristic approach most definitely can be improved with the evolutionary cycle as the marble count increases while keeping the number of moves needed on the same level. However, it cannot be said by how much exactly the baseline algorithm is exceeded, a direct comparison in experiment 6 shows that the optimised algorithm beats the baseline AI in most cases. Regarding the presented evolutionary optimisation approach it needs to be noted that the layout of the algorithm inherently offers many angles apart from the mutation and crossover processes based on which further research can be conducted [21]. This quite possibly results in improving the agent more (concerning the proposed evaluation metrics) than can be displayed in the scope of this work. Variables of the genetic algorithm that are also of interest in that sense are for example the size of the permutation interval for the training agents and the trade-off between this and a higher/lower mutation rate. Simpler changes that can yield interesting results are the size of each generation and the generation count. Therefore the second part of the fourth research question *How does altering the cross-over and mutation process affect the outcome of the optimisation process?* cannot be answered conclusively, however given the relatively low generation count used, seeding the initial generation and then increasing the mutation rate as well as the intervals for the training values (explained in detail in the subsection VI-A of VI) might improve the AI player further.

The sixth experiments results show that the optimized greedy algorithm has an overall much better performance, with an average win-rate of 19/27 against every baseline AI's. This supports the idea that the genetic algorithm greatly improved the weights to create a highly-performing AI. Moreover, the average win-rate per player position also reveals that being player 3 is putting any AI at an advantage. In addition to that, it can also be concluded on a hierarchy between every game mode for the Greedy algorithm. The optimized version of it is, undoubtedly, the best one, followed by the defensive, the neutral version and then the aggressive.

Testing the Monte Carlo tree search, and finding averages from the MCTS algorithms bound at 2, 5 and 10 seconds show that for each of these bounds, the same number of nodes gets back-propagated and around the same number of nodes get added to the tree. This is due to the automatic back-propagating after one hundred moves, which is implemented so the tree would not go too deep. It turns out that the back-propagation only occurs after ten seconds have already passed, making the results quite similar. As is said in the previous section about MCTS, there is a small chance to actually find a game path which ends within one hundred moves, if the moves are chosen randomly (which would happen at the start of MCTS). Further research could be done to find a better way to choose the nodes to be expanded.

Besides that, testing MCTS with bounds of 20 seconds and 40 seconds, it seems that whenever a second selection is being done, it will end up being either too slow to add any more nodes or get a out-of-memory exception. Because of this, it can be deduced what the total number of nodes in a tree of the game states can be. The number of nodes in total that could be added to the tree is around 11300. As the MCTS, as implemented in this paper, adds all its nodes whenever a leaf is reached, a way to optimise this search would be to cut as many nodes as possible, only leaving the one which would be traversed after the current node.

XI. Conclusion

In the paper a static framework for developing an Abalone agent with a linear evaluation component and several selection components is proposed. It is found that in its current implementation, the greedy algorithm offers a better trade-off between the win-rate and time needed to perform a move. The second point being the main reason why it constitutes the basis for the genetic optimisation. The reason for why the greedy implementation outperforms the alpha-beta selection (in regard to the given metrics) is most likely rooted in the alpha-beta pruning methods and the sorting of its candidate tree. When using a heuristic pruning approach like proposed in [7], the alpha-beta is expected to outperform the greedy algorithm at least in regard to the win-rate (ie. marbles won per game).

Furthermore, it is found that genetic optimisation of the evaluation component offers a significant improvement of the performance of the Abalone agent as can be seen in the outcomes of Experiment 6, as well as in Fig. 14-19. . However, as noted before, the genetic algorithm most likely converged to some local optimum and hence does not show the 'best' possible behaviour. Further research can be conducted on the many variables that can be changed in the evolutionary process like mutation rate, the training pool, the initialisation of the first generation etc..

A MCTS implementation of the selection algorithm is also proposed and while in theory offers some interesting

advantages over the genetic approach such as a self-learning component, the way it is implemented here is not as efficient computationally as it can be with a more domain focused approach, where not every node gets added to a leaf when it is expanded, but rather, one node at a time.

REFERENCES

[1] O. V. Baskov, "Equilibrium payoffs in repeated two-player zero-sum games of finite automata," *International Journal of Game Theory*, Jun 2019.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, 2018.

[3] G. Tesauro, "Temporal difference learning of backgammon strategy," in *Machine Learning Proceedings 1992*, D. Sleeman and P. Edwards, Eds. San Francisco (CA): Morgan Kaufmann, 1992.

[4]

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[6] N. Lemmens, "Constructing an abalone game-playing agent," *Maastricht University, Department of Data Science and Knowledge Engineering Bachelor Thesis*, 2005.

[7] O. Aichholzer, F. Aurenhammer, and T. Werner, "Algorithmic fun-abalone," *Institute for Theoretical ComputerScience Graz University of Technology*.

[8] E. Ozcan and B. Hulagu, "A simple intelligent agent for playing abalone game: Abla," 2004.

[9] L. V. Allis, "Searching for solutions in games and artificial intelligence," 1994.

[10] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Nov.

[11] P. Campos and T. Langlois, "Abalearn: Ecient self-play learning of the game abalone," 2009.

[12] A. S. Michel Lalet, Laurent Lévi, "Abalone official rules," *https://cdn.1j1ju.com/medias/c2/b0/3a-abalone-rulebook.pdf*, 1989-2002.

[13] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. USA: Prentice Hall Press, 2009.

[14] N. M. Darwish, "A quantitative analysis of the alpha-beta pruning algorithm," *Artificial Intelligence*, vol. 21, no. 4, pp. 405 – 433, 1983.

[15] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.

[16] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, 2016.

[17] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, "Monte-carlo tree search solver," pp. 25–36, 2008.

[18] S. Thede, "An introduction to genetic algorithms," *Journal of Computing Sciences in Colleges*, vol. 20, 10 2004.

[19] H.-G. Beyer, "The simple genetic algorithm-foundations and theory." *IEEE Trans. Evolutionary Computation*, vol. 4, pp. 191–192, 01 2000.

[20] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evol. Comput.*, vol. 8, no. 2, p. 173–195, Jun. 2000.

[21] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 1, pp. 122–128, Jan 1986.

[22] G. Paris, D. Robilliard, and C. Fonlupt, "Exploring overfitting in genetic programming," in *Artificial Evolution*, P. Liardet, P. Collet, C. Fonlupt, E. Lutton, and M. Schoenauer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 267–277.

APPENDICES

A. Time experiment, full results

Move Nr.	Greedy	Alpha Beta	Monte-Carlo
1	5044	126	15989
2	3651	69	10827
3	3911	77	11863
4	3246	46	14404
5	4183	51	12843
6	3141	39	10157
7	2760	46	16543
8	3548	44	10190
9	3729	38	11027
10	5434	25	16693
11	3609	34	11096
12	3070	28	16371
13	3114	34	11244
14	2966	23	21708
15	3427	33	10470
16	3431	24	16763
17	3804	28	19561
18	3864	36	10423
19	3933	27	10357
20	3661	25	14905
Average	3676	43	13672

TABLE VIII: Time needed to perform a move.

B. Monte Carlo experiment, full results

When a number of back-propagated nodes does not have a value, it means that it got stuck without reaching a last back-propagation.

Move	Back-propagated	Added nodes	Time used (ms)
1	100	8243	12795
2	100	8294	19491
3	100	8370	17101
4	100	8795	17104
5	100	7967	15811

TABLE IX: Results of moves with second bound 2.

Move	Back-propagated	Added nodes	Time used (ms)
1	100	8502	30866
2	100	8554	20534
3	100	8492	16432
4	100	8564	18264
5	100	6846	16937

TABLE X: Results of moves with second bound 5.

Move	Back-propagated	Added nodes	Time used (ms)
1	100	8290	24319
2	100	8510	17253
3	100	9288	20361
4	100	8815	17165
5	100	9270	17871

TABLE XI: Results of moves with second bound 10.

Move	Back-propagated	Added nodes	Time used (ms)
1	100	8361	27877
2	-	11313	¿120000
3	100	8507	20249
4	-	11298	¿120000
5	100	8232	26620

TABLE XII: Results of moves with second bound 20.

Move	Back-propagated	Added nodes	Time used (ms)
1	-	11309	¿120000
2	-	11296	¿120000
3	-	11322	¿120000
4	-	11328	¿120000
5	-	11322	¿120000

TABLE XIII: Results of moves with second bound 40.

Note: in move 3, 4 and 5 of this last experiment, there was an out of memory exception quickly after the two minute mark.