Maastricht University

Coding an Abalone Game-Playing Agent

Detry Clément, Imparato Adèle, Podevyn Loris,
Poliakov Ivan, Schapira Aaron, Thirion Guillaume, Yap Mathias

August 27, 2023

Abstract

This paper explores game-playing agent approaches for the board game Abalone. Using some heuristics evaluation, the agent can perform well. It consistently beats human players in spite of the complexity of the game. The paper primarily focuses on optimization and comparison of two tree search algorithms: MiniMax with Alpha-Beta pruning (called Alpha-Beta Tree Search - ABTS) and Monte-Carlo Tree Search (MCTS).

1 Introduction

Since computers were created, people have tried to make them play various kinds of games. Board games in particular are of interest to many people, thus it is not surprising that they find a huge amount of applications in the domain of Artificial Intelligence (AI).

The first AI programs were written by the University of Manchester in 1951[1]. Using the Ferranti Mark 1 machine, Checkers and Chess programs were created. It was only the beginning of AI applications in games. Since then, many improvements have been made in this field and AI programs have been developed for almost every popular board game.

Abalone is a relatively young game. Although it has seen some AI implementations, it is not as well studied as its more famous counterparts in chess and checkers. This paper will explore game-playing agents that play the Abalone game as well as possible.

2 Game

2.1 History

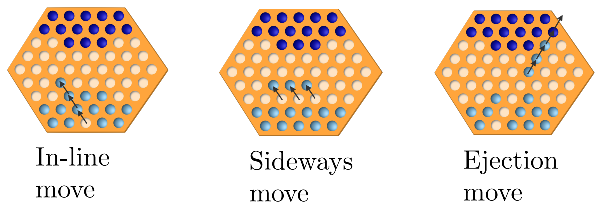
Abalone is a two-player strategy game. It was designed by Michel Lalet and Laurent Lévi in 1987 and is inspired by Japanese Sumo. The game was published in 1990 and received a Mensa Select award the year it was released[2]. In 1998 the game was ranked 'Game of The Decade' at the Cannes International Game Festival[3].

2.2 Rules

Abalone is played on a hexagonal board containing sixty-one circular spaces plus six spaces on each side. The game opens with twenty-eight marbles on the board. Fourteen light blue marbles are disposed on the bottom rows and fourteen dark blue marbles on the top rows. Player one controls the light blue marbles, player two controls the dark blue marbles.

Players are allowed to move between one and three marbles connected in a straight line. Marbles can be moved sideways and in-line. In-line moves allow a player to push marbles belonging to another player. Pushes are legal if the marbles are pushing a lower number of marbles than the pushed marbles. The two players alternate, making one move each turn.

The goal of the game is to push the opponent's marbles off the board. The game is won by pushing six marbles belonging to the opponent.



3 Game Tree

3.1 Implementation

A game tree is necessary to run the MiniMax algorithm on it. It will search through the game states in the tree to find the best available moves.

The game tree consists of nodes that represent potential game states and edges that represent moves transforming one state into another. They are initialised using a Breadth-First Search algorithm. To build the game tree it takes the current board state as the root and takes all possible moves from this state. For each move it adds the state it leads to as a child of the root. This is the first generation. For each child c of the first generation, it computes each possible move from the position at c again. All of these moves are added as children of node c. It continues the process for every node of generation one until the second generation is complete. The process continues until a goal depth is reached. The following figure is a model of the game tree.

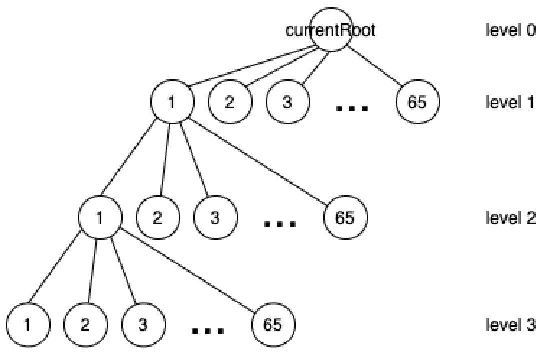


Figure 1: Game Tree visualisation

3.2 Complexity Analysis

Due to the complexity of Abalone, it is not feasible to build a complete tree of all moves. From a board position, there are on average 65 different ways to move one/two/three marbles. Computing the entire game tree will take a long time since we know that the creation of a tree is logarithmic. To enable the computer to compute the game tree (the shorter one), it is necessary to give it a depth limit. For the MiniMax Algorithm, 3 seems to be a reasonable choice. This way, the entire game tree contains about 274625 ($= 65^3$) nodes and its complexity is 270965. This number corresponds to the amount of leaf nodes in the tree which is the number of possible different ways the game can be played (note that the tree contains only starts of games, the actual number of possible ways to play the game would be the complexity of the complete tree).

3.3 Use in MiniMax

The MiniMax Algorithm will use this game tree in order to decide which move to make. Since it is not complete (i.e. the leaves of the tree are not end positions), the game tree nodes first need to be "evaluated" using an evaluation function. Its purpose is to give a score to each possible state, which will define how good a move is. If a move leads to game states with good evaluations, it is a good move. The evaluation is very important since the bot aims to choose the move that gives the highest performance.

3.4 Transposition table

In the Abalone game tree data structure repeated states can occur frequently because of transposition[4]. By taking different paths in the game tree, it can end up with exactly the same board position and player turn in both states regardless. A transposition table takes advantage of this fact.

Its implementation consists of a hash table that keeps track of the evaluation scores of the boards visited previously. If a currently visited node $n2$ has the same key for the hash table as a previously visited node $n1$, it means they are the same. Instead of evaluating the current position, it can take the evaluation score of the previously visited note.

4 Evaluation Function

In order to have an efficient game tree, the nodes need to give a good evaluation of the current board state. To do so, three different evaluation functions have been implemented. Together, they are based on 10 different heuristics[4]

4.1 Heuristics Evaluation

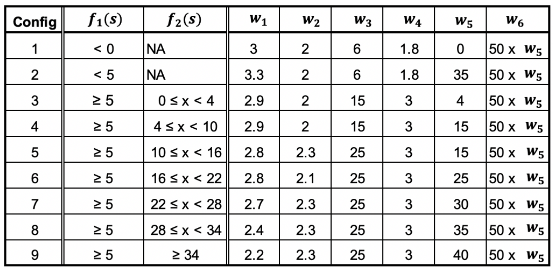
- $f_1(s)$ corresponds to the difference between the Manhattan center distance of both player marbles.
- $f_{1bis}(s)$ corresponds to the Manhattan center distance of the current player marbles.
- $f_2(s)$ stands for the cohesion strategy. It determines the difference between neighbouring of each player marble.

- $f_{2bis}(s)$ determines the neighbouring distance of the current player marbles
- $f_3(s)$ is the heuristic that again computes the difference between the amount of 'break-strong-group' pattern of each player. This can be recognised as an opponent marble at both adjacent sides of a current player marble.
- $f_4(s)$ is calculated in the same way as $f_3(s)$ however, it searches for a current player marble that has an opponent one and an own one at both adjacent side. It is used to know how many possible contact pushing marble both players have.
- $f_5(s)$ compares the number of opponent marbles still in the game between the board at the beginning of the search and the current state.
- $f_6(s)$ is equivalent to $f_5(s)$ but deals with the current player marbles.
- $f_7(s)$ counts the number of sumito positions for the current player's marbles.
- $f_8(s)$ counts the number of pushing positions for the current player's marbles.

4.2 Neutral Evaluation Function

$$E(s) = (\sum_{i=1}^{5} w_i \cdot f_i(s)) - w_6 \cdot f_6(s)$$

The Neutral Evaluation Function is the most efficient one since it adapts its weights depending on the state of the board it is evaluating. It gives a well balanced score which can be either offensive and defensive based on the situation.



Config	$f_1(s)$	$f_2(s)$	w_1	w_2	w_3	w_4	w_5	w_6
1	< 0	NA	3	2	6	1.8	0	50 x w_5
2	< 5	NA	3.3	2	6	1.8	35	50 x w_5
3	≥ 5	0 ≤ x < 4	2.9	2	15	3	4	50 x w_5
4	≥ 5	4 ≤ x < 10	2.9	2	15	3	15	50 x w_5
5	≥ 5	10 ≤ x < 16	2.8	2.3	25	3	15	50 x w_5
6	≥ 5	16 ≤ x < 22	2.8	2.1	25	3	25	50 x w_5
7	≥ 5	22 ≤ x < 28	2.7	2.3	25	3	30	50 x w_5
8	≥ 5	28 ≤ x < 34	2.4	2.3	25	3	35	50 x w_5
9	≥ 5	≥ 34	2.2	2.3	25	3	40	50 x w_5

Figure 2: Neutral strategy weights conditions[4]

The table shows that the aim of the agent is to get the center first. As the cohesion increases in comparison to the one of the opponent, it reinforces offensive strategies and weakened the need to occupy the centre. Without losing the determination to preserve its own marbles which is always crucial.

4.3 Offensive Evaluation Function

$$E(s) = w_1 \cdot f_1bis(s) + w_2 \cdot f_2bis(s) + w_5 \cdot f_5(s) + w_7 \cdot f_7(s) + w_8 \cdot f_8(s)$$

The offensive strategy focuses on ejecting the opponent's marbles rather than protecting its marbles. For that reason, w_5 is set to an especially high value, this will engage the agent to take more risks by trying to push more often opponent marbles out.

4.4 Defensive Evaluation Function

$$E(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + w_4 \cdot f_4(s) + w_6 \cdot f_6(s)$$

The defensive strategy focuses on protecting its marbles rather than attacking the opponent. To do so it gives a very low value to w_6 for the purpose of avoiding getting one of its marbles ejected.

5 Algorithms

5.1 MiniMax

Since Abalone is a deterministic two players game, the MiniMax algorithm suits it well[5].

In the MiniMax algorithm one player will be the Max player, and the other will be the Min player. The algorithm occurs whenever it is the Max player's turn to play. Its goal is to find a path that will lead to a favorable outcome. To do so, it will visit all the leaves the game tree has computed. Every leaf contains a state of the board as well as an evaluated score. The Max player will aim to find a sequence of moves that will lead it to a leaf that is evaluated highly. In Abalone it is impossible to compute the game tree until the end of the game. Therefore the Max player can only look at 3 moves in advance (depth=3) and thus make a move with only the information of those leaves. Since the Max player does not know which move the Min player will perform, it will choose the most promising move regardless of what the Min player does.

5.1.1 Alpha-Beta pruning

One of the disadvantages of the MiniMax algorithm is; to find the best solution, it has to search as deep as possible in the game tree. The reason why the whole tree structure can not be created is because of the time requirements that grows exponentially and restricts the depth of the tree. Alpha-Beta pruning can improve tree search time, increasing the depth in a time window.

Alpha-Beta pruning works in such a way that it will eliminate branches of the tree that are useless and thus the search algorithm will not have to check each branch. The result will always be the same as MiniMax, so it saves time but maintains accuracy.

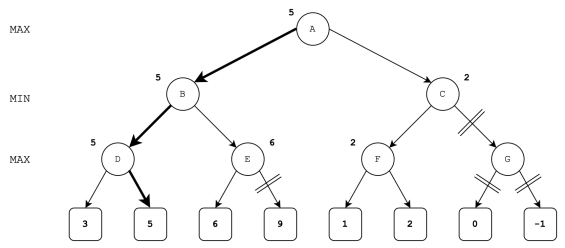


Figure 3: MiniMax Tree after Alpha-Beta pruning

5.1.2 Complexity Analysis

Estimate the average number of possible moves from each game state as M and average number of plies needed to finish the game as T. Denote the depth of the tree as D. Each move, recompute all the nodes in the current game subtree. The total time complexity of MiniMax is $\mathcal{O}(T \cdot M^D)$.

After each move, the game tree not only adds new nodes but also deletes those that are not reachable anymore. For this reason, space complexity is $\mathcal{O}(M^D)$. In the best move ordering case, time complexity for ABTS is $\mathcal{O}(M^{D/2})$ (since Alpha-Beta pruning can skip at most half of the nodes) and space complexity stays the same.

5.1.3 Move Ordering

For Alpha-Beta pruning to perform well, it is important that good moves are checked first. Knowing this, it is important that our game tree takes into consideration what moves could be good. By making assumptions about moves before evaluating them, an ordering could be made for possible moves. Good moves will be at the top of this ordering and therefore they will be checked first. This paper tests and considers three different orderings; random ordering, simple ordering and full ordering.

Random ordering will be a complete random ordering of the moves that are checked. It is expected to not perform well.

Simple ordering will order three marble moves first, followed by two marble moves, followed by one marble moves. This ordering is based on two basic principles in Abalone. The first is that a player will only be able to score using a move that pushes. It is possible to push in Abalone using three or two marbles, but it is not possible to push in abalone using one marble. The second is that forums of Abalone players generally agree that it is important to keep your marbles grouped [6]. Moving three marbles will more often keep the marbles grouped than moving two marbles. In turn, two marbles will more often keep the marbles grouped than one marble.

Full ordering expands on the idea that moves that capture and moves that push generally are better moves. It orders the game tree in this way:

1. 3 marbles, capturing
2. 2 marbles, capturing
3. 3 marbles, attacking
4. 2 marbles, attacking
5. 3 marbles, moving
6. 2 marbles, moving
7. 1 marble, moving

Here the move ordering prioritizes pushing moves over non-pushing moves and capturing moves over regular pushing moves. Another difference from the previous move ordering is that it will prioritize two marble capturing and attacking moves over three marble moves that do not capture or attack. A similar move ordering has been effective at improving Alpha-Beta pruning previously [5]. It is expected that this move ordering will perform similar to the simple ordering until there are many pushing and capturing moves. Therefore it is expected to outperform in the mid-game and potentially in the late game too.

5.2 Monte-Carlo Tree Search

The Monte-Carlo algorithm is a popular search technique based on statistics and probabilities[5]. Using the law of large numbers[1] concept, it appears to be a good predictor in board games.

The version that has been implemented in the game is the Monte Carlo Tree Search (MCTS). The idea of this algorithm is to simulate random games as in the Monte Carlo Search, but this time using four principles to simulate those games.Those principles are the Selection, the Expansion, the Simulation and the Back-propagation.

The **selection** is done by using Exploration and Exploitation. The biggest hurdle in this is choosing

[1]As the sample size of simulations grows, its mean gets closer to a good estimate of the effectiveness of the move

which child to explore. To do so, a formula is used that balances the exploration and the exploitation; the UCT score (Upper Confidence Bound 1 applied to trees). The formula is as follows:

$$\frac{w_i}{n_i} + c * \sqrt{\frac{\ln N_i}{n_i}}$$

with w_i being the number of wins for the node, n_i being the number of simulations for the node, c being the exploration parameter, and N_i being the total number of simulations of the parent. Once the possible moves are generated, each of them is given a UCT score. The highest scoring move will be visited until reaching a leaf node.

When the current node is a leaf, the algorithm **expands** it with its possible moves and choose a random move.

The next step is the **simulation**, it consists of simulating a certain number of games starting at the current board. The algorithm takes as a parameter the number of plays, which is the number of random plays by simulated games. This is done because one total game would be to long to compute. Another feature used for the simulation is the sample size, which corresponds to the number of simulated games by node. Once a node has been processed, it receives a score based on the result. To do so, the algorithm uses a weighting function defined by

$$\sqrt{\frac{|simulatedscore - currentscore|}{5}}$$

If the score computed by the simulation is bigger than the score from the current node, then the method will add the computed score to the actual one. Otherwise, if the computed score is less than the actual one, the method will subtract the computed score from the actual one.

The last step is the **back-propagation**. Once a node has been simulated and its score has been computed, the algorithm will back-propagate its score onto its parents, until reaching the root, thus, the UCT score will be updated.

The algorithm works with a stop condition. In this paper, it corresponds to a predefined timer. When the time is reached, the algorithm will output the move that has been simulated the most.

5.2.1 Complexity Analysis

Let's suppose that by the time MCTS finishes there will be N nodes in the tree. Then space complexity is $\mathcal{O}(N)$. We can also roughly estimate selection time complexity as $\mathcal{O}(N)$ since in the worst case we will have to check each node in the tree. Time complexity of expanding a node is $\mathcal{O}(M)$ where M is the average number of possible moves from each position. Simulation time complexity we can simply denote as S. Last but not least back-propagation time complexity also can take $\mathcal{O}(N)$. Since we repeat the process for all nodes in the tree, total time complexity is $\mathcal{O}((N + S + M) * N)$.

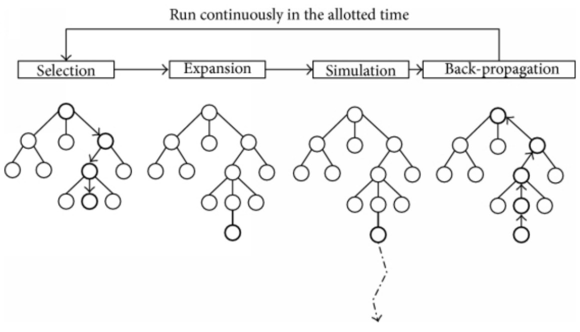


Figure 4: Monte-Carlo Tree Search process[7]

5.3 Rule-Based Algorithm

The Rule-Based algorithm is an algorithm that enables an agent to make a decision following a set of rules. In the case of Abalone, the Rule-Based bot follows a list of rules in the form of if/else statements giving it instructions on how to play. This bot is especially fast at making a move (less than 0,1 second) since it does not anticipate the opponent's moves resulting in a very short time computation.

5.4 Alternatives

As an alternative strategy to build an agent there exist multiple examples of successful agent training based on genetic algorithms[8]. The agent should be able to perform a strategy that aims towards game positions with the highest reward/number of points/evaluation function value(e.g. greedy selection of each move, MCTS). Thus, the genetic algorithms could be used for calculating evaluation function weights. In order to explore this option we developed the following teaching strategy. First of all, we were simulating two parallel evolution processes using default multi-threading provided in Java[9]. This was done for 2 reasons: first, concurrency proved to speed up two independent calculations (was working 1.25 times faster on i5 8300H), second, it could help avoid sample bias. Initially we had 2 "islands" of 100 gens. Then first island's population was evolving in accordance with Rank parent selection and the other one in accordance with Pie selection. After the offset was generated and placed instead of weaker genes, both populations would exchange 50% of their total population randomly between each other. Afterwards, the best candidates would compete in accordance with tournament selection principles and generate offset consisting of pairwise combination of the best candidates. However, this turned out to be quite challenging to train genetic algorithm effectively using the given in the research evaluation function. Since it is essentially changes its weight sets as the game progresses it turned out that using only one weight set leads to a huge number of positions ending in a draw.

5.4.1 Complexity Analysis

Time complexity of the generation update is $\mathcal{O}(M \cdot N \cdot K)$, where M is the average number of moves from each game state, N is the average number of plies made in a single game and K is the total number of games.

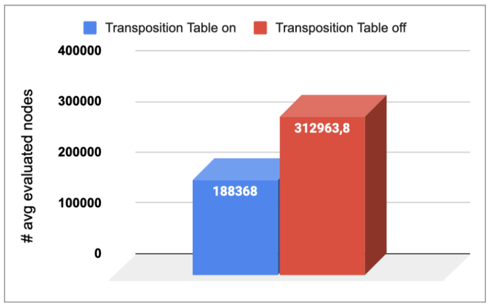
6 Experiments

The experiments that have been carried out in this scientific paper are about comparing different type of AI's configurations and various pruning techniques. Based on the collected information's, questions have been asked to which interpretations have been developed. The data is stored in a google sheet available through the link in the appendix.

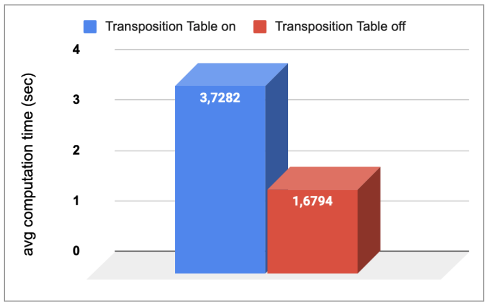
6.1 Research and results

6.1.1 Game Tree

The game tree data structure for Abalone is huge due to the average branching factor as already explained before in the paper. Because of this, using a transposition table is a good way to avoid evaluating transposed node several times. But how useful is it for our game tree implementation? To answer this question some tests with and without the transposition table have been done for the game tree construction.



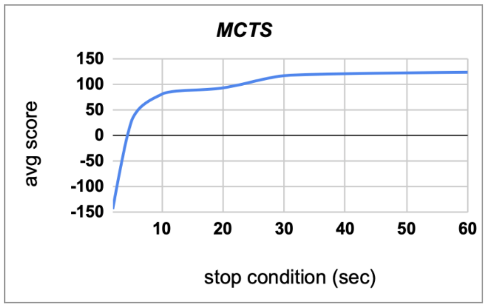
This graph shows the gain of computation when the transposition table is on for a game tree structure of depth 3. The amount of transposed nodes is high which supports its usefulness.



Unfortunately, the game tree computational time is a bit higher when the transposition is active. It is due to some missing optimisations of the hash table/function in the code. Evaluating more nodes is shorter in time than checking each time in the table. But when the evaluation function contains more heuristics and is more computationally expensive, the transposition table is a good way to save time.

6.1.2 Monte-Carlo Tree Search (MCTS)

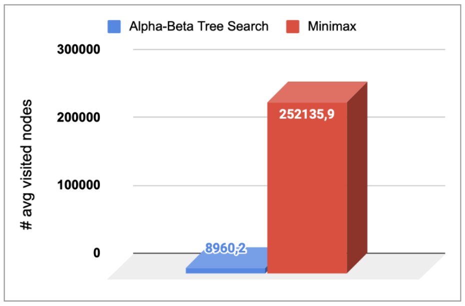
The MCTS algorithm is not able to run without having a stop condition predefined. In this paper, it corresponds to a fixed amount of time. What happens if this condition is changed? How is the MCTS output move influenced by it? What is the best stop condition to use?



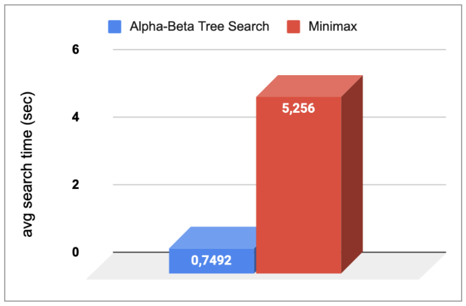
This graph is relevant to find an answer to the previous questions. It clearly shows that the MCTS becomes usable from a stop condition of 10 seconds. Before that, the average score of the output moves are not very reliable. Because the algorithm has not been able to simulate enough times to find the best action. But the more the condition is increased is not related to a way better performance. From 30 seconds, it seems to converge to a constant score. The best stop condition to set appears to be 10 seconds since this gives good results in general and it is a reasonable amount of time too.

6.1.3 Alpha-Beta Tree Search (ABTS)

Alpha-beta pruning is a well known technique to run the minimax tree search faster. But how much does the efficiency change if we activate the alpha-beta pruning through the minimax search? Does it really improve the speed of the game tree search?

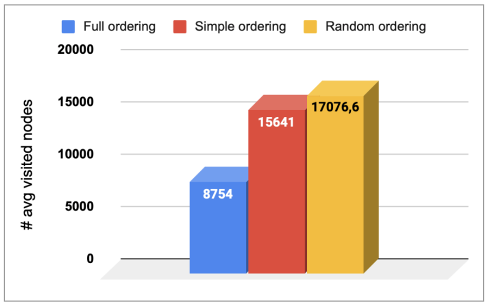


This graph gives details about the average investigated nodes by the minimax and ABTS. The difference is huge as it can be observed. In general, ABTS saves up to 96% nodes to visit. What about the computational time improvement?

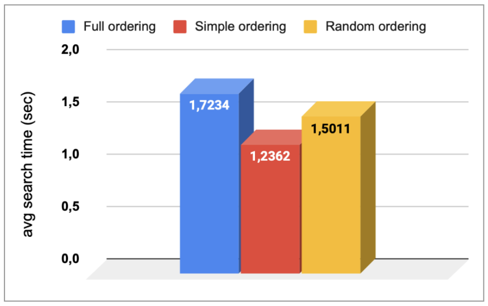


As could be expected, alpha-beta pruning allows to gain a lot of time. This proves the crucial importance of using it each time the game tree is explored.

Another pruning technique for the ABTS is the move ordering. It has already been introduced and explained previously but here is a quick recall. It is basically used to improve the possible alpha-beta pruning in the game tree through the 3 different move ordering already stated. Using this information, questions may arise. How helpful is it to use them in the game?



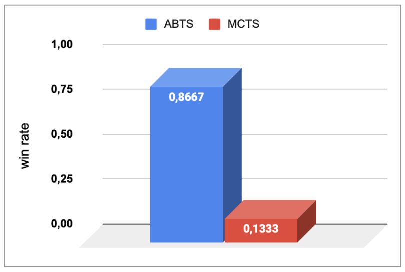
On average, the full ordering gives better result in terms of visited nodes. It clearly improves alpha-beta pruning. Data from the simple and random ordering are similar. The simple one still remains better which seems logical because random data does not have a linear performance.

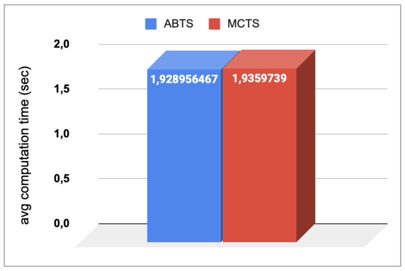


The average search time results of ABTS using the three different ordering is not correlated to the previous graph. As it can be observed, the full ordering is the one that takes the highest amount of time to compute (but still reasonable). It is certainly due to the ordering complexity. The fully ordered list contains the possible sumito and pushing move in addition to the other possible moves. The simple ordering is the computationally cheapest because it only needs to find the triple, double and single marbles moves. While the random one also needs to compute them and then randomize it.

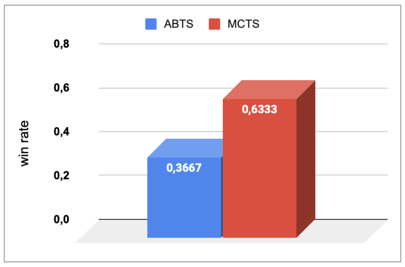
6.1.4 Evaluation Function

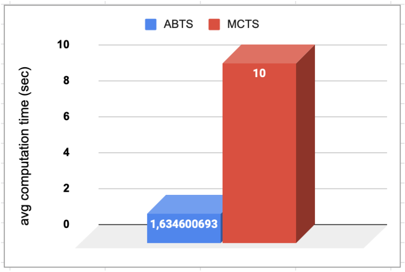
The evaluation function is basically used to run our AI's. The more efficient it is, the better the AI's will be. The neutral strategy is known to be our best evaluation function. But which is the most efficient bot between ABTS and MCTS when they both use the neutral evaluation function?





After running a bunch of total games between them with a 100% fairness (exactly the same amount of time to output a move), the ABTS appears to be the best one with a win rate of 86%. It is certainly due to the fact that MCTS is not efficient enough under a stop condition of 10 seconds (here, on average 1.93 seconds). But what if the stop condition of the MCTS is risen to 10 seconds against the ABTS? Will it become better in terms of win rate?





The first graph proves that now, MCTS comes up with a better win rate (63%). But their fairness differ a lot because ABTS now computes the best move 6x faster. This is not really relevant then since the efficiency of the ABTS cannot be increased as its opponent do (game tree of depth 3 max). But it is still good to know that MCTS can beat its adversary by taking a reasonable amount of time though.

7 Conclusion

To improve the performance of MiniMax different heuristics have been applied in this paper. Alpha-Beta pruning proved to be very effective in the test results. It reduced both visited nodes and computational time greatly. It was even further improved when an effective move ordering was selected. A full ordering that prioritizes pushing moves proved to be most effective. A transposition table greatly reduced the evaluated nodes, but usage of the transposition table instead of evaluating the board position cost the algorithm more computational time.

MCTS proved to be efficient depending on the fixed amount of time given for the stop condition. The main threshold is located around 10 seconds. The lower the stop condition is in relation to the threshold, the more ineffective it will be. On the other hand, it will converge to a constant effectiveness when we exceed it.

Three evaluation functions were implemented in this paper; an offensive one, a defensive one and a neutral one. Offense nor defense proved to be very effective when compared to the neutral evaluation function. For that reason, the neutral evaluation function was implemented in both main algorithms and their comparison.

In completely fair matches, MiniMax with Alpha-Beta pruning outperformed MCTS 86% of the time. Further testing showed that when the stop condition of MCTS is increased, it starts performing better in the matches. With a stop condition of 10 seconds, it only loses 63% of the matches. Alpha-Beta Search does not improve with time, whereas MCTS does. From the findings it is reasonable to conclude that MiniMax with Alpha-Beta pruning is the best algorithm implemented in this paper. It outperforms MCTS where they are both in fair conditions.

References

[1] B Jack Copeland. The modern history of computing. 2000.

[2] American Mensa. `https://mensamindgames.com/about/winning-games/`.

[3] Abalone - a mundial success! `http://abalone-game.com/appli-en.html`.

[4] N.P.P.M. Lemmens. *Constructing an Abalone Game-Playing Agent.* Maastricht University, 18/06/2005.

[5] Pascal Chorus. *Implementing a Computer Player for Abalone using Alpha-Beta and Monte-Carlo Search.* Maastricht University, 29/06/2009.

[6] Ender Ozcan. *A Simple Intelligent Agent for Playing Abalone Game: ABLA.* Proceedings of the 13th Turkish Symposium on Artificial Intelligence and Neural Network, 2004.

[7] Maciej Świechowski. Four steps of the monte-carlo tree search algorithm. `https://www.researchgate.net/figure/Four-steps-of-the-Monte-Carlo-Tree-Search-algorithm_fig11_282043296`.

[8] Karol Wal¸edzik Magdalena Kusiak and Jacek Ma´ndziuk. Evolutionary approach to the game of checkers. 2007.

[9] Ausif Mahmood Ajay Shrestha. Improving genetic algorithm with fine-tuned crossover and scaled architecture. 2015.

Appendix

- Click on this **link** to access the experiments data.