Performance of Monte Carlo Tree Search Algorithms when Playing the Game Ataxx

Leonardo F. R. Ribeiro[1], **Daniel R. Figueiredo**[1]

[1]Programa de Engenharia de Sistemas e Computação (PESC/COPPE)
Universidade Federal do Rio de Janeiro (UFRJ) – Rio de Janeiro – RJ

leoribeiro@cos.ufrj.br, daniel@cos.ufrj.br

Abstract. *Monte Carlo Tree Search (MCTS) has recently emerged as a promising technique to play games with very large state spaces. Ataxx is a simple two-player board game with large and deep game tree. In this work, we apply different MCTS algorithms to play the game Ataxx and evaluate its performance against different adversaries (e.g., minimax2). Our analysis highlights one key aspect of MCTS, the trade-off between samples (and accuracy) and chances of winning the game which translates to a trade-off between the delay in making a move and chances of winning.*

1. Introduction

In recent years, computers have become strong opponents for humans in many games. Successful and prominent examples are in games of chess and Go, where computers are able to challenge the best human players in the world [Silver et al. 2016]. Games with two players that are zero-sum, perfect information, deterministic, discrete and sequential are described as combinatorial games [Browne et al. 2012]. Combinatorial game are excellent for AI agents as they have controlled environments defined by simple rules. However, these games typically exhibit a large and deep game tree posing significant challenges in designing algorithms to play them satisfactorily.

Ataxx is a combinatorial game developed around 1990 for the Atari platform. Its name comes from the word "attack" because each player must play aggressively to be able to win. Ataxx is a relatively unknown game despite its interesting and simple rules and high game-tree complexity. Because of its large game-tree complexity, it is not possible use minimax in Ataxx. However, we can use minimax with a fixed horizon such as minimax2 (horizon of two moves).

Monte Carlo Tree Search (MCTS) has recently emerged as a promising technique to play games with large and deep game trees [Silver et al. 2016]. Its basic idea is to determine the most promising move from a given state in the game tree by performing random sampling on the remainder of the game tree. However, there are various strategies for sampling the tree such as uniformly at random, weighted by prior results, or based on multi-armed bandits algorithms.

In this paper we evaluate the performance of Monte Carlo Tree Search (MCTS) algorithms to play Ataxx. We consider different strategies for the MCTS player and opponents that play according to random choices or minimax2. We show that MCTS can consistently win over a random player by making very little effort (small number of samples). Moreover, by making more effort (larger number of samples) MCTS can win

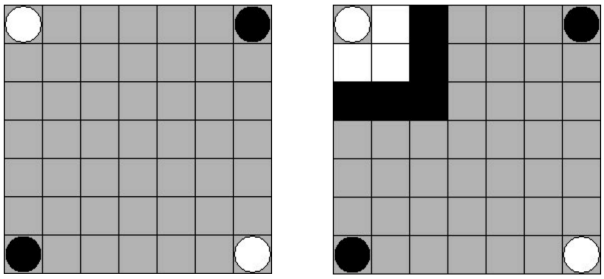

Figure 1. (Left) Starting positions. (Right) Possible moves for the upper left stone (figure from [Cuppen 2007]).

against minimax2. This highlights a fundamental trade-off of MCTS algorithms between the time required to make a move (translated into the number of samples) and the chances of winning the game.

The remainder of this paper is organized as follows. In Section 2 we present some related work. Section 3 we present the rules of the game Ataxx. We describe MCTS in detail in Section 4 with results presented in Section 5. Finally, Section 6 concludes the paper with a brief discussion.

2. Related work

Monte Carlo methods have enjoyed significant success in various AI game playing algorithms, particularly imperfect information games such as Scrabble and Bridge [Heyden 2009]. Monte Carlo Tree Search (MCTS) applies Monte Carlo to generate samples from the game tree yielding efficient tree-search algorithms. MCTS has nowadays become a fully matured search method with well defined parts and many extensions. In recent years MCTS has done remarkably well in many deterministic classic board games, such as Othello [Nijssen 2007], Amazons [Lorentz 2008] or Ataxx [Cuppen 2007]. However, it is really the success in computer Go, through the recursive application of Monte Carlo methods during the tree-building process, that has been responsible for much of the recent interest in MCTS [Silver et al. 2016].

In [Jacobsen et al. 2014] MCTS techniques are applied to control the player character in a clone of the popular platform game Super Mario Bros. They investigated the performance and behaviour of MCTS on the Mario AI benchmark, and found that standard MCTS performed relatively poorly. They then proposed a number of modifications to MCTS showing that a judicious combination of these modifications yields an agent that plays near optimally.

The Game Ataxx has been explored little in the context of AI game playing algorithms. In particular, Cuppen [Cuppen 2007] developed some heuristics using Monte Carlo techniques to play Ataxx, but not MCTS algorithms to explore the game tree. Instead of using pure Monte Carlo techniques as described by Cuppen [Cuppen 2007], in this work we apply different MCTS algorithms to play the game.

3. Ataxx game

Ataxx, also known by such names as Infection, SlimeWars and Frog Cloning, is a two-player board game that first appeared in 1990 as an arcade video game. It is an abstract

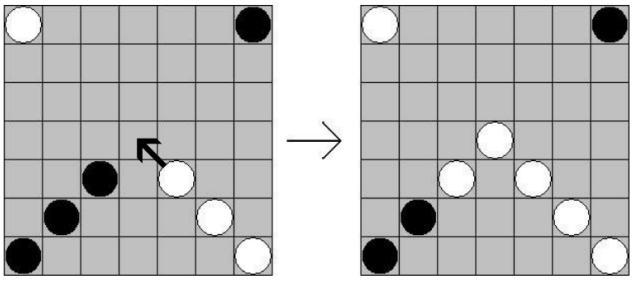

Figure 2. (Left) Copying a piece. (figure from [Cuppen 2007])

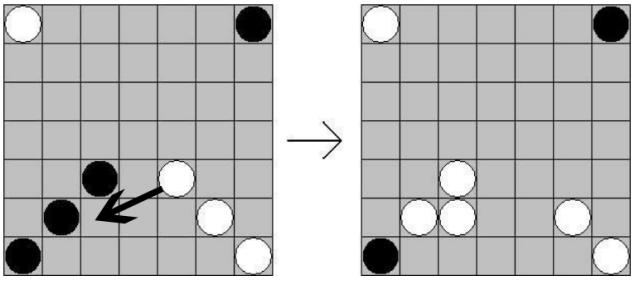

Figure 3. Jumping with a stone (figure from [Cuppen 2007])

strategy board game that involves sequential and alternating play by two opponents on a seven-by-seven square grid.

3.1. Game rules

Each player begins with two pieces, white and black, for the first and second players respectively. The game starts with four pieces on the four corners of the board (see figure 1-left), with white pieces in the top left and bottom right and black pieces on the other two. White moves first. The object of the game is to make your pieces constitute a majority of the pieces on the board at the end of the game by converting as many of your opponent's pieces as possible.

At each turn, a player must make a move, if possible. There are two types of moves: *copy* and *jump*. Both moves are only possible when moving to a empty space. In a *copy* move the current player can copy the chosen piece for one of its adjacent empty squares (white squares for the upper left stone in figure 1-right). In a *jump* move, the chosen piece will be moved to an empty square adjacent to the copying squares (black squares for the upper left stone in figure 1-right), that is, you can go further, but the piece is not copied. After a move, all of the opponent's pieces adjacent to the destination square are converted to the moving player. This is illustrated in figures 2 and 3.

The game ends when all spaces have been filled or one of the players has no remaining pieces on the board. The player with most pieces wins the game.

We have designed and implemented an efficient algorithm for playing Ataxx using MCTS and other strategies, such as minimax and random, that can be parametrized accordingly. We make the code publicly available at: `https://github.com/leoribeiro/mcts-ataxx`

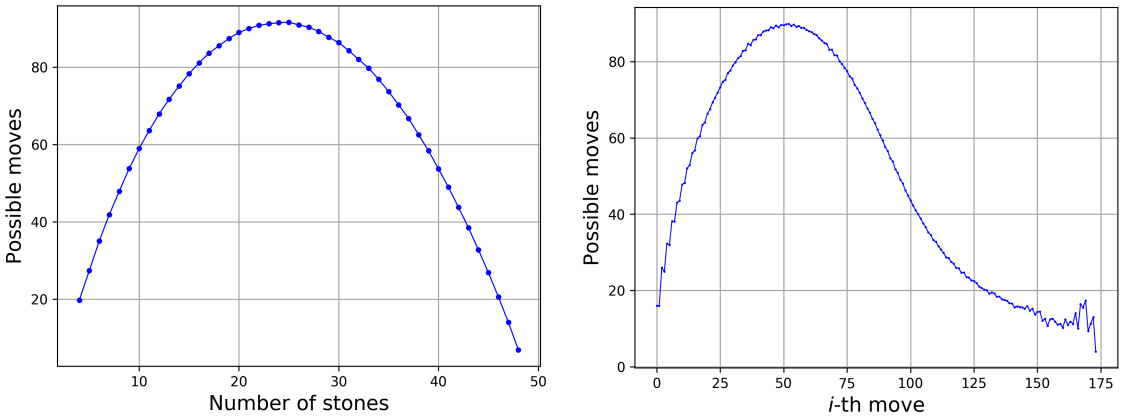

Figure 4. (Left) The average number of possible moves by the number of current stones on the board. (Right) The average number of possible moves by each ply in a game. Both graphs were generated using 10000 game simulations.

3.2. Game-Tree Complexity

A game tree is a tree where nodes are states of the game (e.g., configuration of the board) and edges are possible moves from that state. The complete game tree for a game is the tree starting at the initial configuration and containing all possible moves from each state.

The game-tree complexity is the number of different games which can be played. It indicates the total number of terminal nodes in the game tree. In many games often it is impossible to calculate the game-tree complexity exactly. However, we can estimate it using two values: the branching factor and the game length.

The branching factor indicates how many different moves a player can make from on average and the game length indicates the average number of plies performed until the game is over. A ply is a move taken by one of the players. The game-tree complexity can be estimated by raising the average branching factor (average possible moves) b to the power of the average game length d (depth of the tree):

$$GTC \approx b^d$$

In table 1 we show estimated values for the branching factor, game length and game-tree complexity calculated for Ataxx, using 10,000 simulations generated by the simulator we developed.

Branching factor	game length	Game-tree complexity
61.6	110.9	2.91×10^{198}

Table 1. Estimation of branching factor and game length.

Beyond the game-tree complexity, it is interesting to characterize how the number of moves depends on the round of play. This provides an indication of the complexity of the game as it unfolds during play. Figure 4(left) shows the average branching factor for the number of stones on the board. The average branching factor of the first ply is 20 and there is a peak on ply 25, in the middle of the game, where the branching factor is 92. Figure 4(right) shows the average branching factor by the i-th ply in a game. There is a peak on the 52nd ply with branching factor of 90 possible moves.

Finally, using a deterministic and exhaustive approach to play Ataxx can be challenging because of the number of possible games that can be played.

4. Monte Carlo Tree Search

In this section we present different algorithms that use Monte Carlo Tree Search (MCTS) techniques to play Ataxx. MCTS aims finding optimal decisions in a given domain by taking random samples in the game tree space and building a search tree according to the results [Browne et al. 2012]. Our goal is to select the move that is most likely to win the game in a given state of the game tree.

4.1. Monte Carlo Methods

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Abramson [Abramson 1990] demonstrated that this sampling might be useful to approximate the game-theoretic value of a move. The Q-value of an action is simply the expected reward of that action:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{1}(s_i, a)\, z_i$$

where $N(s,a)$ is the number of times move a has been played from the state s, $N(s)$ is the number of times a game has been played out through state s, z_i is the result of $i-$th simulation played out from s, and $\mathbb{1}(s_i, a)$ is 1 if move a was picked up from state s on the i-th play-out from state s or 0 otherwise. Monte Carlo approaches in which the actions of a given state are uniformly sampled are described as flat Monte Carlo [Browne et al. 2012].

4.2. Bandit-Based Methods

Bandit problems are a class of sequential decision problems in which one needs to choose between K actions (e.g. the K arms of a multi-armed bandit slot machine) in order to maximize the cumulative reward yielded by the actions [Browne et al. 2012]. In general, each action provides a random reward from a specific probability distribution. Choosing the best action is difficult because the underlying distributions are unknown and potential rewards must be estimated based on past observations. The crucial trade-off that a player faces at each trial is between *exploitation* of the action that has the highest expected reward currently believed to be optimal and *exploration* to get more information about the expected reward of the other actions, that currently appear sub-optimal but may turn out to be superior in the long run.

4.2.1. Upper Confidence Bounds (UCB)

The Upper Confidence Bound (UCB) is a bandit-based method which has an expected logarithmic growth of regret uniformly over time without any prior knowledge regarding the reward distributions [Auer et al. 2002]. For each round, this strategy chooses the action i which maximizes the following formula:

$$UCT = \bar{X}_i + 2\, C_p \sqrt{\frac{2 \ln n}{n_i}}$$

where $\bar{X}_i$ is the average reward from arm i, n_i is the number of times arm i was played and n is the total number of plays across all arms. The reward term $\bar{X}_i$ encourages the exploitation of higher-reward choices, while the right hand term encourages the exploration of less visited choices. C_p controls the trade-off between exploitation and exploration in the game tree, namely, a larger C_p encourages exploration allowing the method to seek for other choices. In this work we use $C_p = 1$. UCB has the property that your regret, the difference between what you would have won by playing solely on the actual best move and your expected winnings under the strategy that you do use, grows only as $O(\ln n)$.

4.3. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) determines the best move from a given state by making random samples on the game tree. The algorithm progressively builds a partial game tree, that can depend on the results of previous exploration of that tree.

The value of a move (the chance of winning) may be estimated using random simulation of the game. So the tree is used to estimate the values of moves, with these estimates (particularly those for the most promising moves) becoming more accurate as the tree is built with more samples [Browne et al. 2012].

For each state, the algorithm builds the remainder of the search tree until some predefined computational budget (e.g. time, memory) is reached, at which point the search is stooped and the best move is returned.

Each iteration of MCTS consists of four steps [Chaslot et al. 2008]:

- **Selection:** Starting at the root node, a child is recursively selected through the tree until the most urgent expandable node is reached. A node is expandable if it represents a non-terminal state and has unvisited (i.e. unexpanded) children.
- **Expansion:** One (or more) child nodes are added to expand the tree, according to the available moves.
- **Simulation:** A simulation is run from the new node to produce an outcome.
- **Backpropagation:** The outcome of simulation is "backed up" (i.e. backpropagated) through the selected nodes to update their statistics.

Figure 5 shows an iteration of the MCTS algorithm. During the selection phase, starting at the root, child nodes are recursively selected according to some function until a node is reached. The reached node describes a state that can be terminal if the game is over or not if tree is not fully expanded. At the figure, the second node at depth 3 was reached (the node with statistics 3/3).

In the expansion phase, the reached node that represents state s is expanded. A move a from state s is randomly selected and a new leaf node is added to the tree, which describes a new state s' reached from applying move a to state s. After creating s', we initialize the simulation phase. A random simulation of the game is started from s' and stops when the computational budget is reached or when the game reaches its end. At the end of simulation a result is generated.

The last phase is a backpropagation. The result generated by the simulation is used to update statistics of nodes in the tree. We update statistics of all nodes on the path between the node at the end of the simulation and root. Many iterations from root

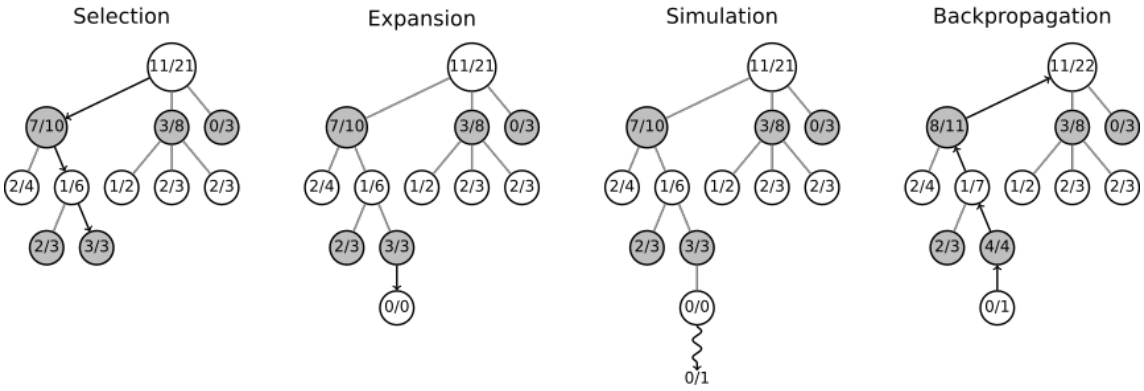

Figure 5. One iteration of MCTS (figure from [Wikipedia 2018]).

node are executed, each passing through the four phases, until a computational budget is reached.

4.4. Using MCTS in Ataxx

We assume the computer plays black in Ataxx, and thus plays first. The main goal of MCTS is to determine the best move for the black player given the current game state. Figure 5 represents a partial game tree built. The root node represents the current game state. Tree levels with black nodes represent possible states after black player's moves and levels with white nodes represent possible states after the white player's moves. For each node, we consider all possible moves from the current game state, the corresponding game state (after the move), a win count (number of times that playing this move led to a win) and a visit count (number of times this move was chosen).

For each iteration, starting at the current game state (denoted by root) child nodes are sampled until a terminating criteria, such as a maximum number of iterations. In this work, we use two strategies to select child nodes: uniformly at random or using UCB. Once a child node is selected, we start a simulation of the game with random play (plays selected uniformly at random) until the game is over. Note that random play will not select end game states uniformly at random, as there is a bias towards earlier end games (understanding and removing this bias is subject of future research). Once the game finishes, we update all nodes on the path from the root to the end game state, as follows.

Let v_i denote the number of visits to node i, which is equivalent to the number of times node i has been played. Let w_i denote that number of times that after playing node i an end game with a win for black was reached. Let f_i denote a sum of the fraction of black pieces for every end game with a win for black after playing node i. Note that every end game has a fraction of black pieces, independent of being a win or loss for black (and this value is added to f_i when black wins).

We build the tree executing many iterations from the current game state. The algorithm then chooses the child node (the next move) with the highest score. We define two scores: *binary* and *fractional*. The binary score of a node i is given by w_i/v_i while the fractional is given by f_i/v_i. We will consider these two methodologies for selecting the next move.

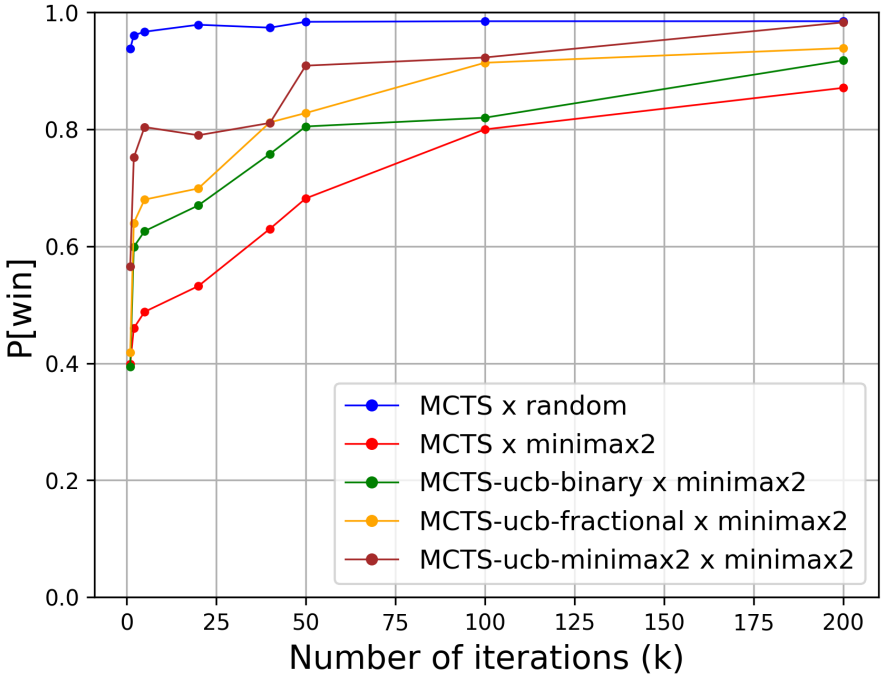

Figure 6. Fraction of wins between the MCTS player and others by the number of iterations (samples) used in each MCTS move. These values were calculated using 1000 different games.

5. Experimental Setup

We evaluate our approach of using MCTS to play Attaxx using different scenarios and present the results concerning winning the game. In simulator developed for playing Ataxx we can choose between the strategy for different players. In particular, the following strategies are available:

- **random:** This player randomly and uniformly chooses the next move from all possible moves given the current state.
- **minimax2:** Minimax [Russell and Norvig 2016] is used to compute the value of a game state in a two-player deterministic game. It generates all possible game states after playing a certain number of plies. This means that minimax is a full-width tree search. The player chooses the move with the maximum game value and the opponent chooses the move with the minimum game value. The value of a game will be the fraction of black pieces on the board in the end game state. However, since it is not possible to expand the complete tree for Ataxx, we will use a fixed the depth of 2, and thus, minimax2.
- **MCTS:** This player uses MCTS and selects the child nodes with the highest binary score.
- **MCTS-ucb-binary:** This player uses MCTS and selects child nodes using UCB based on binary score.
- **MCTS-ucb-fractional:** This player uses MCTS and selects child nodes using UCB based on fractional score.
- **MCTS-ucb-minimax2:** This player uses MCTS and selects child nodes using UCB based on binary score. However, it uses the minimax2 strategy during the simulation phase, as opposed to random play.

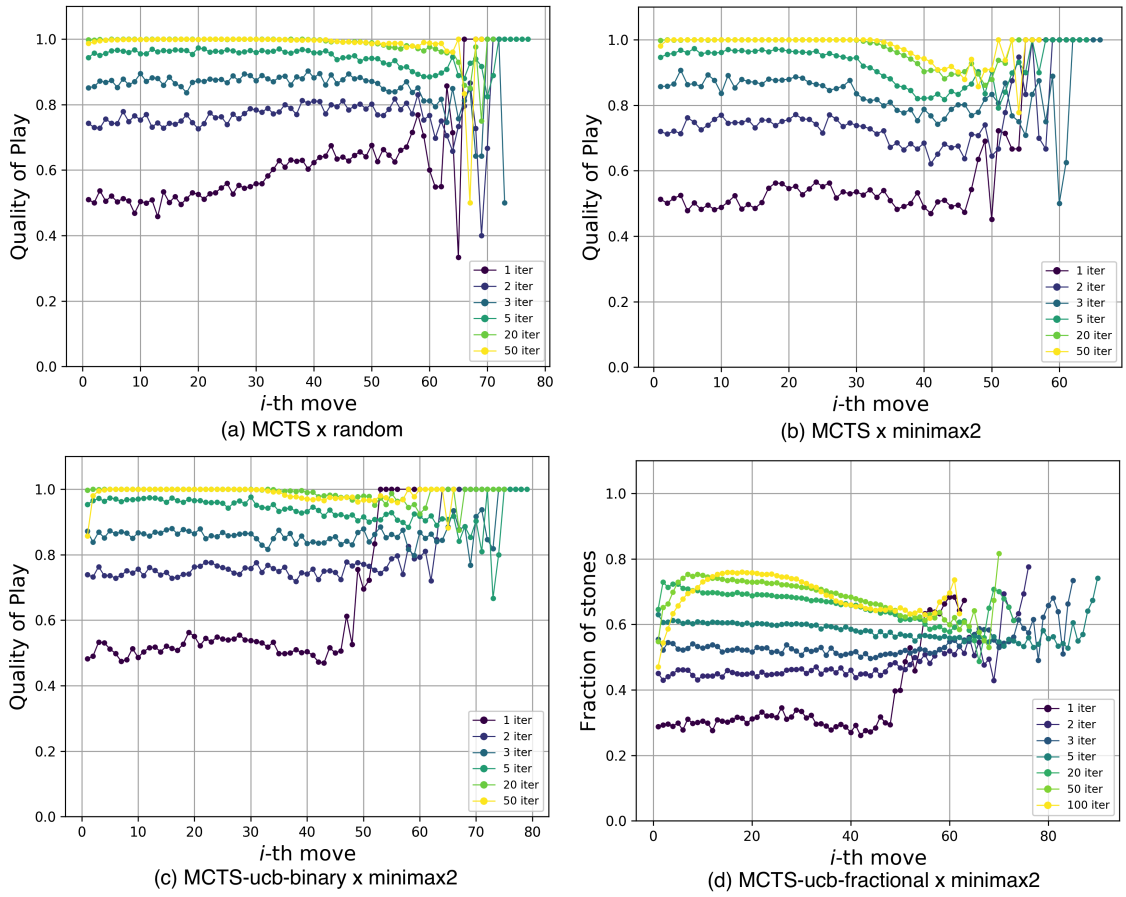

Figure 7. (a)(b) Average score of the move chosen by the MCTS player against random and minimax2 players, respectively, by the i-th move at the game. (c) Average score of the move chosen by the MCTS-ucb-binary player against minimax2 player by the i-th move at the game. (d) Average fraction of MCTS-ucb-fractional player's stones in the board during the i-th move on the game. These values were calculated using 1000 different games.

5.1. Results

Figure 6 shows the fraction of time that the MCTS player won the game when playing against other strategies as a function of the number of iterations used by MCTS to determine the best play. Note that the MCTS player can beat the random player easily. Even with a single iteration, it can beat the random player in more than 90% of the time. Playing against minimax2 player is more difficult and with few iterations the MCTS player can win in 50% of games. However, as we increase the number of iterations, MCTS player wins a larger fraction of the games. With 100 iterations for each MCTS move, MCTS player wins in 80% of games. Although it is intuitive that increasing the number of interactions improves the quality of play, the improvement is super linear (Figure 6). The player using MCTS with UCB has a better performance than the MCTS player, but as we increase the number of iterations the difference between them decreases. The orange curve describes the MCTS player using UCB and scoring the moves according to fraction of pieces on the board. This player performs better than previous MCTS players when playing against the minimax2 player. Indeed, using the fraction of pieces on the board is more informative for a given move than using a win/loose signal.

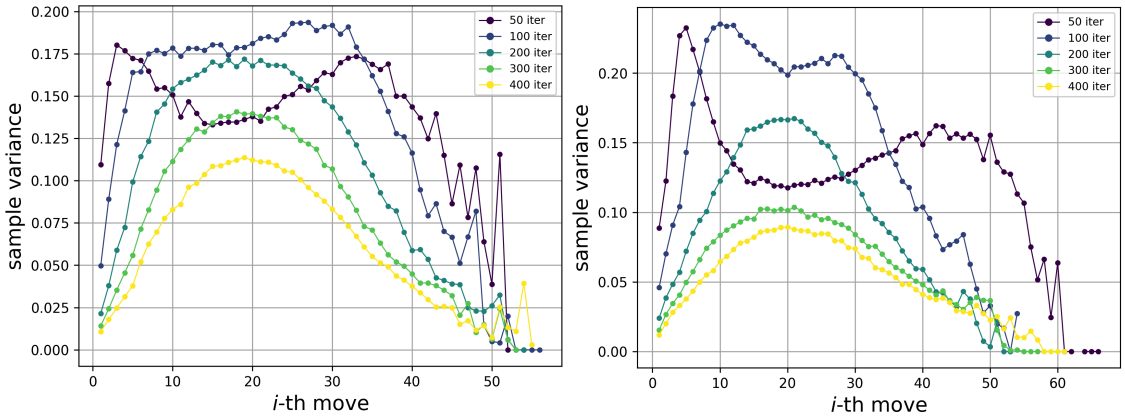

Figure 8. The sample variance of move estimators used by the MCTS (Left) and MCTS-ucb-binary (Right) players in each i-th move of a game. We averaged these values using 500 executions of different games playing against minimax2.

Lastly, the strategy that combines MCTS with UCB and minimax2 (MCTS-ucb-minimax2 player) against a pure minimax2 player has the best performance among all MCTS strategies. Note that with 200 iterations it shows the same performance as the classic MCTS playing against the random player. However, this strategy has a larger time complexity (because of simulations using minimax2), but builds a tree with many more nodes that guide the MCTS to select the best move with more confidence.

Figures 7(a,b,c) show the average score of the node chosen by the MCTS player as a function of the round of play during the game. Each curve shows the values for a different number of iterations. Interestingly, as the number of MCTS iterations increases, the scores of the chosen node also increases. This occurs because the tree used by the MCTS player has more information and hence can better decide its next move. However, for a fixed number of iterations the average score of the chosen move does not change significantly over the rounds of the game (with the exception at the end of the game).

Unfortunately, the score of the chosen node did not increase as the game developed. In some cases the score decreased in the end of the game. This occurs possibly because towards the end of a long game it becomes more difficult to find winning end game states. Thus, the chance of wining the game from such states is low, thus leading to moves that have a low score.

Figure 7(d) shows the average fraction of stones on the board that the MCTS-usb-fractional player has in each i-move as the game evolves. This fraction remains stable for most of the game, and increasing the number of iterations leads to a larger fraction of stones. This shows that increasing the number of iterations leads to a better overall strategy, since it maintains a larger fraction of stones during the game, thus increasing its chances of wining the game at the end. Note that there is noise at the end of the curves in Figure 7 due the fewer amount of data concerning these moves. This occurs because different executions of the game can have different amount of plies, and executions with many plies are less frequent.

Figure 8 shows the sample variance for the scores computed by the MCTS (left) and MCTS-ucb-binary (right) players for each move of the game. In the beginning, moves

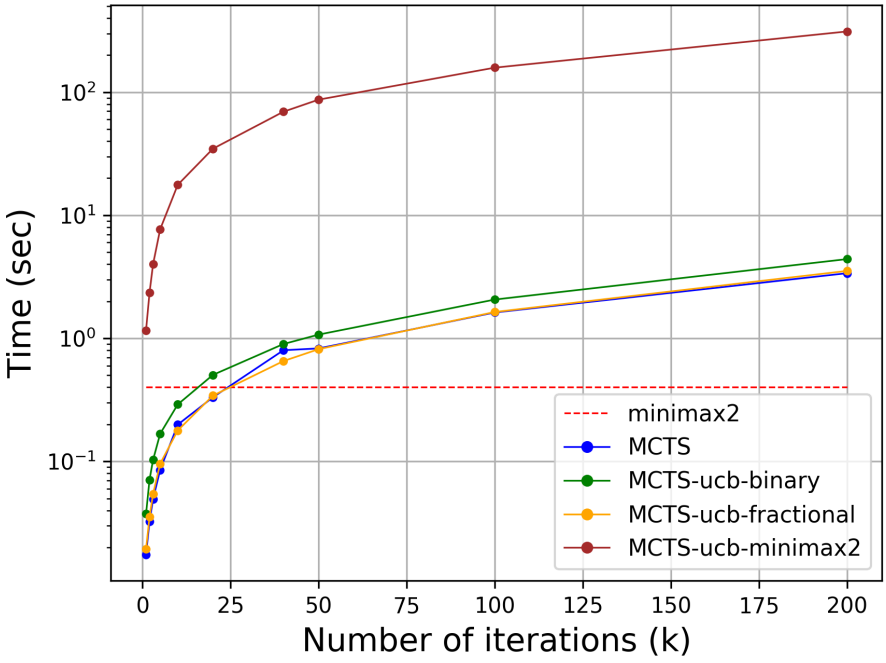

Figure 9. MCTS execution time (in seconds) to choose the best move by the number of iterations. Each curve was generated using the average running time spent by MCTS on each move of 1000 game simulations.

have a lower variance due the few possible moves available. For every i-th move of the game we use the same number of iterations (curves). As we increase the number of iterations, the variance reduces, as expected. In the middle of the game we have largest variances because the number of possible moves is much larger. In the end of the game the variance decrease as the number of possible moves available is smaller. This observation agrees with our previous analysis of the size of the game tree, illustrated in Figure 4. Note that we have used a constant number of iterations for every game state, but the sample variance could be leveraged to determine the number of iterations.

Figure 9 presents the execution time of MCTS to choose the best move, in seconds. Each curve corresponds to a different MCTS strategy using different number of iterations. As we increase the number of iterations, the execution time increases, as expected. The MCTS strategies using random and UCB have similar performance. The strategy that combines MCTS and minimax2 has the largest execution time because for each iteration, the method executes some minimax2 moves to calculate the scores. Note that the last strategy performs better among all other, but has a longer running time.

6. Conclusion and future work

Monte Carlo Tree Seach (MCTS) has emerged as a promising technique to play games that have very large state spaces, and in particular when time to make a move is limited. In this paper, we proposed and explore various MCTS algorithms to play the Game Ataxx. We evaluate our approaches against the random and minimax2 players as well as under its key parameter, the number of iterations to determine the best move given a game state.

Our findings lead to interesting observations, such as the inherent trade-off be-

tween the number of iterations and the chances of winning. Moreover, the number of iteration to win the game is also proportional to the quality of the opponent. Under a random opponent, MCTS can win a very large fraction of the games using very few iterations. However, under a minimax2 more iterations are needed to win a larger fraction of games played. Our results also highlight the role of sample variance when estimating the best move, as the game tree is not uniform as the game unfolds.

As future work, we would like to investigate the performance against minimax3 player and use domain knowledge to improve the selection step in MCTS. Note that a experienced player knows which next moves are more promising than others, and this could be leveraged to improve MCTS, targeting its iterations to estimate the score of those game states.

References

Abramson, B. (1990). Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence*, 12(2):182–193.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43.

Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. (2008). Monte-carlo tree search: A new framework for game ai. In *AIIDE*.

Cuppen, E. (2007). Using monte carlo techniques in the game ataxx. *B.Sc. Thesis, Maastricht University*.

Heyden, C. (2009). Implementing a computer player for carcassonne. *Master's thesis, Department of Knowledge Engineering, Maastricht University*.

Jacobsen, E. J., Greve, R., and Togelius, J. (2014). Monte mario: platforming with mcts. In *Proc. Annual Conference on Genetic and Evolutionary Computation*, pages 293–300.

Lorentz, R. J. (2008). Amazons discover monte-carlo. In *International Conference on Computers and Games*, pages 13–24.

Nijssen, J. (2007). Playing othello using monte carlo. *Strategies*, pages 1–9.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Pearson Education.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587).

Wikipedia (2018). Monte carlo tree search - Wikipedia. [Online; accessed May-2018].