Who Invented MCTS?

Tim Debets, Arun Gade, Albert Negura, and Carlos Tello

***Abstract*—*Monte-Carlo Tree Search (MCTS)* has been at the forefront of search AI research for more than a decade. There are many algorithms which claim to be proto-MCTS algorithms. We present a comparison between MCTS and a proto-MCTS algorithm called *Adaptive Multi-stage Sampling (AMS)* in the context of game AI. Despite adapting AMS to perform in the context of games and implementing several MCTS-enhancements, including MCTS-style play-outs, Move-Average Sampling Technique and N-Gram Selection Technique, AMS completely underperforms in comparison to MCTS on most of the games, achieving a similar win rate to MCTS only in Reversi/Othello, Connect Four and the multiplayer games Tic-Tac-Mo, Yavalde and Triad with only specific enhancements when using UCB1 selection.**

I. Introduction

One field of research in the context of Artificial Intelligence has focused on the development of intelligent search algorithms [1] [2]. These algorithms were then tested playing human experts in various games, such as chess and Go [2]. Within this field, the algorithms could be classified in different categories, including depth-first search methods and best-first search methods[1].

In 2006, Coulom and Kocsis & Szepesvári proposed a best-first search method, named Monte-Carlo Tree Search (MCTS) [3] [4]. MCTS utilizes the results produced by Monte-Carlo simulations and a randomized exploration over the search space to gradually build up a game tree in memory. It improves successively, leading to better estimates of the values of the most promising moves. MCTS has substantially advanced the state of the art in board games such as Go, Hex and chess, becoming one of the 'goto' algorithms for searching through the state space of board games.

There have been many Monte-Carlo algorithms proposed in the past - some of them nowadays claim to be a kind of proto-MCTS. The question is in how far this claim is true. In 2005, Chang et al. [5] proposed such a proto-MCTS algorithm in the form of Adaptive Multi-stage Sampling (AMS). Originally, AMS is implemented in the context of finite-horizon Markov-Decision Processes (MDPs) and based on the UCB1 algorithm proposed by Auer et al. in 2002 [6]. The AMS algorithm claims to produce asymptotically unbiased estimates, whilst converging to the optimal quantity in MDPs.

Comparing these two algorithms leads to a better understanding of the AMS algorithm in the field of games instead of MDPs. If AMS behaves similar to MCTS, the claim that it is a proto-MCTS algorithm can be substantiated. Furthermore, by this comparison the differences and similarities between the algorithms become clear and even how various enhancements influences the performance of either algorithm.

[1]The complete project source code can be found at https://github.com/AlbertNegura/WhoInventedMCTS

Consequently, the following three research questions guide this study:

- How much do MCTS and AMS differ and how much are they alike in the context of sequential games?
- Does one algorithm perform better than the other on specific (types of) games?
- Would the same enhancements work on both algorithms?
  - If they do, does one algorithm benefit more from said enhancements than the other and if so, which one?

The article is structured as follows. First, Section II gives the necessary background information regarding MCTS, AMS and the various enhancements implemented. Subsequently, Section III and IV discusses the experimental setup and the results respectively. The last section, Section V, contains the conclusion and a discussion on potential future work.

II. Background

This section contains an in-depth explanation of both the Monte-Carlo Tree Search algorithm in subsection II-A and Adaptive Multi-stage Sampling algorithm in subsection II-B. Several enhancements for both algorithms are explained in sections II-C (AMS with Play-outs), II-D (Move-Average Sampling Technique), II-E (N-Gram Selection Technique) and II-F (Memoization using Transposition Tables).

A. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [3] [4] is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. It has already had a profound impact on Artificial Intelligence approaches for domains that can be represented as trees of sequential decisions, particularly games and planning problems [7]. MCTS progressively builds up a tree based on previous iterations over that tree. More iterations translate into a better accuracy at estimating the best move. MCTS repeats four steps until its predefined computational budget – typically a time, memory or iteration constraint [7] - is reached:

- During the **selection** process, the tree is explored starting from the root node down, until a non-terminal state with unexplored child nodes is reached.
  The algorithm has to keep a balance between selecting the most promising nodes known so far (**exploitation**) and selecting less rewarding child (**exploration**). Exploitation ensures the accuracy of the most promising plays, while exploration allows to ensure that other branches get evaluated. Different strategies have been suggested for the selection phase of MCTS, but the most commonly

used is the Upper Confidence Bounds applied to Trees (UCT). A selected child b of a node p should maximize Formula 1.

$$b \in \underset{i \in I}{\operatorname{argmax}}\left(\frac{w_i}{m_i} + C \times \sqrt{\times \frac{ln(m_p)}{m_i}}\right) \tag{1}$$

Where I is the set of all direct child nodes of the current node p, w_i is the number of wins so far of i, m_i is the visit count of i, m_p the visit count of p and C is a constant used to adjust the exploration versus exploitation measure. For an unvisited child, the maximum value is assumed as to ensure that no children are left unexplored before expanding the tree. The constant C can be tuned to increase or lower the amount of exploration.

- A **play-out** of the game is run from the selected node until the game reaches an end.
  During a play-out, the moves are chosen according to a play-out strategy. Play-out strategies may vary, ranging from playing random to smarter, heuristics based moves. Although smarter simulations can significantly enhance the results of the MCTS algorithm, it can have a number of drawbacks:
  - If the moves made are too deterministic, the search space may becomes too narrow resulting a performance drop.
  - Making informed plays may result in more accurate simulations but might take longer to compute.
- The **expansion** of the tree. A popular expansion strategy is to add to the tree the node corresponding to the first position that was not stored for each simulation [3]. Although the expansion strategy chosen has little impact on the strength of the plays, other strategies exist, such as adding all children of a node, specially when the branching factor is small.
- Finally, **backpropagation** of the play-out's results ensures that the explored nodes statistics are updated to the parent nodes.
  Backpropagation updates the value of all nodes that have been passed through, all the way to the root node. The result is propagated as a tuple the size of the number of players, which allows MCTS to handle multiplayer games.

Algorithm 1 shows the pseudocode for the MCTS algorithm and Figure 1 shows the algorithm graphically [2].

Algorithm 1 Monte Carlo Tree Search

Input: Root
while time left **do**
  {*Selection Phase*}
  while currentNode ∈ Tree **do**
    lastNode = currentNode
    bestChild = $\operatorname{argmax}_{i \in I}(\frac{w_i}{m_i} + \sqrt{C \times \frac{ln(m_p)}{m_i}})$
    currentNode = bestChild
  end while

  {*Play-out Phase*}
  result = PlayOut(currentNode)

  {*Expansion Phase*}
  Expand(lastNode)
  currentNode = lastNode

  {*Backpropagation Phase*}
  while p ∈ Tree **do**
    Backpropagate(currentNode, result)
    currentNode = currentNode.parent
  end while
end while
return $\operatorname{argmax}_{i \in I}(Root)$

B. Adaptive Multi-stage Sampling

The Adaptive Multi-stage Sampling algorithm (AMS) approximates the optimal value of finite-horizon Markov Decision Processes (MDPs) with infinite state space, finite number of actions and bounded rewards [5]. AMS is based on the multi-armed bandit problem. The goal of a multi-armed bandit problem is to play as often as possible the action that leads to the highest reward. The idea of AMS is based on the expected regret analysis of the multi-armed bandit problem. The regret can be described as the conundrum of exploration versus exploitation, quantifiable as the loss of not performing the optimal action.

To allow AMS in the context of tree search, a reformulation of AMS is needed. While it can be proven to converge in such situations [5], convergence is only guaranteed when there are no accompanying time or resource constraints. In the context of game tree search, this is no longer the case. By applying an additional time constraint on the problem, the algorithm can no longer loop over the entire action space of the MDP [8].

- During the **initialization** phase of AMS, the immediate actions from the root node are each sampled for a given depth or until a terminal node is reached in order to first approximate a value for each child of the root node. The algorithm will consider all children at each depth and make successive recursive calls to each child node to compute the value. If it does not reach a terminal node, the algorithm will evaluate the leaf node with an heuristic function and backpropagate this value. This is essentially different compared to the AMS algorithm for MDPs, where a value of 0 is backpropagated for the selected leaf node.
- During the **loop** phase, the algorithm selects children nodes which satisfy the UCB1 bound rather than UCT, the method used by MCTS, where a given child b of a node p maximizes the following formula.

$$b \in \underset{i \in I}{\operatorname{argmax}}\left(\frac{w_i}{m_i} + \sqrt{\frac{2ln(m_p)}{m_i}}\right) \tag{2}$$

Where I is the set of all direct child nodes of the current

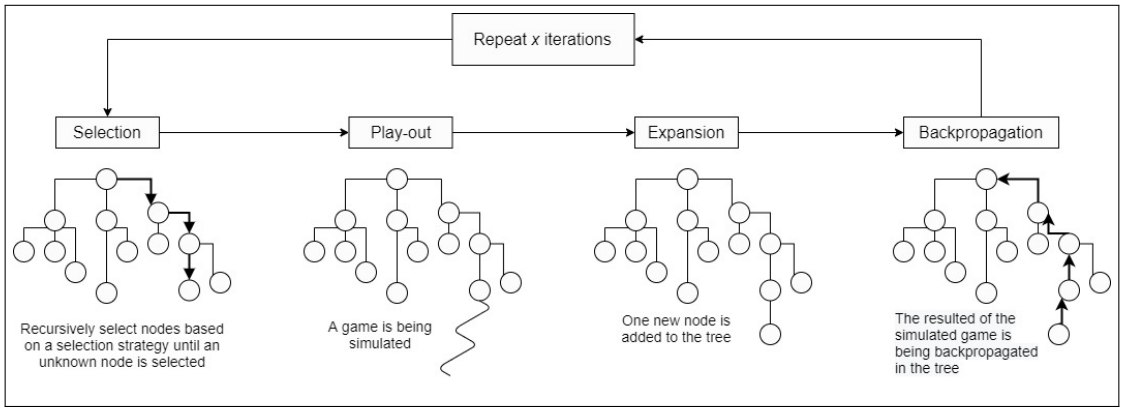


Fig. 1: Four phases executed in Monte-Carlo Tree Search (adapted from Tak *et al.* [1])

node p, w_i is the number of wins so far of i, m_i is the visit count of i, m_p the visit count of p.

Algorithm 2 Adaptive Multistage Sampling

Input: $p, M_i \geq |I|$, depth d
{*Initialization phase*}
for all i $\in$ I **do**
  if d = 0 **or** terminal state **then**
    return Evaluate(i)
  end if
  w_i = AMS(i, M_i, d - 1)
end for

{*Loop phase*}
$\bar{m} = |I|$
while $\bar{m} \leq M_i$ **do**
  $b = \text{argmax}_{i \in I}(\frac{w_i}{m_i} + \sqrt{\frac{2ln(m_p)}{m_i}})$
  Update w_b with AMS(b, M_i, d - 1)
  $\bar{m} = \bar{m} + 1$
end while
return $\sum_{i \in I} \frac{m_i}{M_i} w_i$

The pseudocode for AMS is given in Algorithm 2, where M_i is the number of iterations at a certain depth and $\bar{m}$ represents the number of iterations done thus far. For the algorithm to evaluate the children properly, there is a need for two recursive calls. The first recursive call is in the initialization phase to get a first estimate for the child. The other recursive call is made during the loop phase to continuously update the child with the new estimated value. At the end of the algorithm, a weighted average over all the children is returned for the parent node. When a decision has to be made in the root node, to determine which action should be taken, the move with the highest value will be chosen. In Appendix A a graphical explanation of AMS can be found.

C. AMS Play-out

As discussed in Section II-B, AMS was proposed in the context of MDPs, based on the multi-armed bandit problem [5]. In this problem the goal is to play as often as possible the machine that provides to the highest (expected) reward. In the context of board games the reward is winning or loosing a game when reaching a terminal state, or the heuristic value of a certain board state, when this is non-terminal. It is in this last situation where AMS shows difficulties when sampling the same not-terminal board state. Whereas, the expected reward of this node is the same heuristic value every time it is sampled, thus sampling over the same node does not provide new information. At the same time this stops the algorithm from selecting moves with any randomness, because it retrieves the same values.

In order to deal with the above mentioned situation, this work proposes the play-out enhancement for AMS. Instead of using the heuristic function, the algorithm incorporates a **play-out** phase where it performs a simulation of the game from the sampled node and backpropagates the result of the simulated game. With these play-outs the algorithm obtains new information about the nodes when it samples them. It also adds randomness to the selection process.

In the context of board games AMS might also have difficulties when only applying time constraints. In the case that the maximum number of iterations that the algorithm is allowed to perform per sampled action is set to infinite, the algorithm is likely to spend most of the given time sampling on a certain branch of the tree. A **prioritization** enhancement is proposed to prevent this issue. Only the maximum number of iterations is set to infinite in the root node, while in all its descendants this value is set to a fixed value. This distributes the sampling time of the algorithm equitably. In Appendix A a graphical explanation of this specific enhancement can be found.

D. Move-Average Sampling Technique

The Move-Average Sampling Technique (MAST) [9] is based on the fact that when a move is good in one position, this move is often also good in another position [2]. A global average $Q_h(x)$ is kept in memory for each move x. $Q_h(x)$ is the average of the results of the play-outs in which x has occurred. These values are used to bias the next unexplored

move to investigate, mainly in the play-out phase, but also as a tie-breaking rule for unexplored moves in the selection step.

The original version of MAST uses a softmax-based Gibbs measure to calculate the probability of a move x to be selected in a certain state [9]. Here the moves that are more likely to be selected are the ones with a higher $Q_h(x)$. However the probability of selecting the move with the highest $Q_h(x)$ value is unknown and not fixed with the Gibbs measure. The ϵ-greedy exploration technique, also based on the $Q_h(x)$ values, has been proposed [2]. In this paper, an ϵ-greedy strategy is being followed in the play-out phase. With a probability of ϵ, with $0 \leq \epsilon \leq 1$, a random action is chosen. With a probability of $1 - \epsilon$, an action is chosen based on the $Q_h(x)$ value. This enhancement has been successful in General Game Playing [9] and Lord of the Rings: The Confrontation [10] regardless of its simplicity.

E. N-Gram Selection Technique

MAST remembers the merit of moves from one state to another, without remembering the order of the moves made. The N-Gram Selection Technique (NST), keeps track of the sequence of moves made [1]. The offered context of this method is more favorable in terms of added benefits, using a more realistic sequence of moves, versus the increased computational cost [1]. NST is based on the N-Gram selection technique which is used in natural language processing [11].

This technique remembers the grams, move sequences, of length 1, 2 and 3. The average returned rewards of the play-out is accumulated, which is similar to MAST. This average reward is remembered for all of the grams instead of a single move.

The N-Grams are formed as follows. After each play-out, starting from the root node, for each player, all the move sequences of length 1, 2 and 3 that appeared in the play-out are extracted. Let $\langle x_0, x_1, ..., x_n \rangle$ be the sequence of moves in a specific play-out. x_0 will denote the first action at the root node and x_n will be the final move. Then a N-Gram of length 3 will be any subsequence of this move sequence. For example, both subsequences $\langle x_1, x_2, x_3 \rangle$ and $\langle x_{n-3}, x_{n-2}, x_{n-1} \rangle$ are 3-grams. Figure 2 gives an example for a move sequence of length 6. After the extraction of the grams, the result from this play-out is then backpropagated to the N-Grams. This method does not take into account that move subsequences can appear more than once in the play-out. Thus, if a subsequence appears n times, the N-Gram is updated n times.

The same ϵ-greedy strategy is being followed as in MAST for the play-out phase. Let $\langle x_0, x_1, ... , x_{s-1} \rangle$ be the move sequence in the current play-out and x_s be one of moves currently being considered. Then $\langle x_s \rangle$ is the corresponding candidate for the 1-gram, $\langle x_{s-1}, x_s \rangle$ is the candidate for 2-gram and $\langle x_{s-2}, x_{s-1}, x_s \rangle$ for the 3-gram. This action x_s is only chosen if and only if it maximizes the average of the averages of the candidate N-Grams.

A candidate N-Gram, for sequences of length 2 and 3, is only being considered if it has been visited *k* times [1] [12]. When during the play-out phase, a N-gram has not been visited

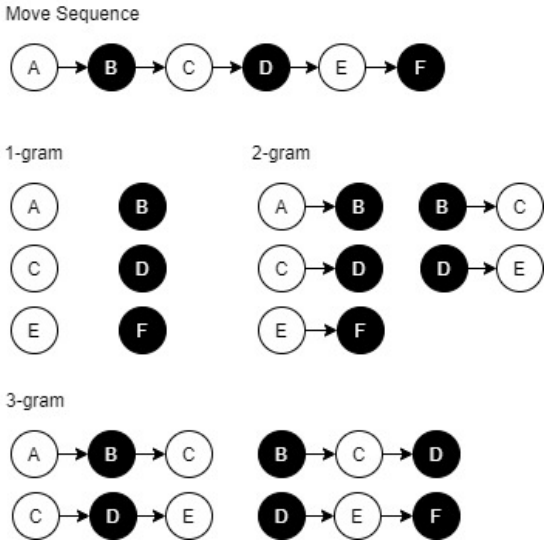


Fig. 2: N-Gram Selection Technique for move sequence of size 6 (adapted from Tak *et al.* [1]).

k times, only the value of the 1-gram is returned. If a move has never been played before, no move sequence exists and a value is assigned to bias towards unexplored moves. Furthermore, when considering nodes close to the root, only smaller N-Grams are being considered. For example, when selecting an action in the root, no previous move exists, so only 1-grams are being considered.

F. Memoization

As it has been mention in subsection II-C, using a heuristic function to estimate the value of a certain board state always returns the same value for this state. In this case the algorithm spends computation resources to obtain values for states that have been previously visited with the recursive calls of the **loop** phase. Transposition tables [13] can be use in this situation to store the heuristic values obtained by the algorithm and decrease this inefficient recursive call.

III. Methodology

A. Implementation Specifics

From the concepts and strategies discussed above, Agents are implemented based on the following algorithms:

- **MCTS** - Regular MCTS.
- **MCTS-MAST** - MCTS with Move-Average Sampling.
- **MCTS-NST** - MCTS with N-Gram Selection.
- **AMS** - Regular AMS adapted for game search trees.
- **AMS-Play-outs** - AMS with random sampling at specific depths.
- **AMS-Priority** - AMS without branch prioritization at the root node.
- **AMS-MAST** - AMS-Play-outs with a MAST-like implementation with branch prioritization.
- **AMS-NST** - AMS-Play-outs with a NST-like implementation with branch prioritization.

The implementation parameters can be found in Table I. These parameters were used for the main experiments between Adaptive Multistage Sampling agents (AMS) and the Monte-Carlo Tree Search agents (MCTS).

TABLE I: The hyperparameters used in the experiments

Parameter	**Explanation**	**Value**
seconds	Time limit per move	1
depth	Search depth of the tree for AMS	2
C	Exploration parameter in UCT	0.4
iterations for prioritized AMS	Number of iterations in the loop phase with exception of root node	50
ϵ	Probability t0 act randomly in MAST and NST	0.1
k	Minimal number of visits before using the N-Gram score for grams of length 2 and 3.	7
value unexplored node	Value assigned to 1-gram for unexplored moves	∞
weight decay	How much the grams should be remembered between turns	0
Number of games	How many games to run for a specific experiment	100

B. The Ludii Game Engine

The experiments have been done using Ludii, a program developed for the ERC-funded Digital Ludeme Project hosted by Maastricht University. This project aims to study how games were played and spread though history using modern Artificial Intelligence techniques [14]. This framework includes a broad library of games that can be played by computer programs and humans, it is able to run custom artificial intelligence agents in the application and it offers a ludeme-based game description language which allows to easily model new games [15]. The first two listed features are the main reasons why this general game system has been chosen over other game systems.

C. Experiment Setup

All the agents played various games among themselves on the Ludii game engine. Table II contains the list of games used for experiments. Tic-Tac-Mo, Yavalade, Triad are multiplayer games that require 3-player to play. Hence, three agents are participating in the various trials with various combinations of player positions amongst themselves.

TABLE II: The list of games on which the experiments are conducted. The games in bold are used for tuning and for comparing the MCTS agents.

Games	**Players**	**Symmetry**
Connect Four	2	Yes
Breakthrough	2	Yes
Knightthrough	2	Yes
Yavalath	2	Yes
Skirmish	2	Yes
Tic-Tac-Chess	2	Yes
Ultimate Tic-Tac-Toe	2	Yes
Reversi/Othello	2	Yes
Hex	2	Yes
Tic-Tac-Mo	3	Yes
Yavalde	3	Yes
Triad	3	Yes

Table III shows the setup of the MCTS agents in their corresponding matchups to compare their individual performances.

TABLE III: MCTS Agents brackets in Breakthrough.

Bracket 1	Bracket 2
MCTS	MCTS-MAST
MCTS	MCTS-NST
MCTS-MAST	MCTS-NST

Table IV and Table V show the detailed setup of the matchups between the MCTS and AMS agents and multiplayer agents respectively.

TABLE IV: Two player agent brackets.

Bracket 1	Bracket 2
MCTS	AMS
MCTS	AMS-Play-outs
MCTS	AMS-Priority
MCTS	AMS-MAST
MCTS	AMS-NST
MCTS-MAST	AMS
MCTS-MAST	AMS-Play-outs
MCTS-MAST	AMS-Priority
MCTS-MAST	AMS-MAST
MCTS-MAST	AMS-NST
MCTS-NST	AMS
MCTS-NST	AMS-Play-outs
MCTS-NST	AMS-Priority
MCTS-NST	AMS-MAST
MCTS-NST	AMS-NST

TABLE V: Multiplayer play brackets.

Bracket 1	Bracket 2	Bracket 3
MCTS	MCTS	AMS
MCTS	MCTS	AMS-Play-outs
MCTS	MCTS	AMS-Priority
MCTS	MCTS	AMS-MAST
MCTS	MCTS	AMS-NST
MCTS	AMS	AMS
MCTS	AMS-Play-outs	AMS-Play-outs
MCTS	AMS-Priority	AMS-Priority
MCTS	AMS-MAST	AMS-MAST
MCTS	AMS-NST	AMS-NST

In all MCTS versus AMS experiment setups, each match is simulated with a maximum time limit per move of 1 second and the win rate is observed over 100 games. The agents are setup and compared in every possible start configuration. For the two-player games, there are two possible configurations. Likewise, for three-player games, there are eight possible configurations per agent. However, two of those possible configurations consist of the same agent. Therefore, only six of these configurations are significant.

IV. Results

In this section, the results from the experiments are presented and discussed. Section IV-A contains a comparison between the three MCTS agents implemented. In section IV-B, the win rates of the algorithms with various enhancements on the presented test games are discussed. Section IV-C contains the results for the corresponding games in relation to the

average number of iterations the algorithms performed on that particular game during the testing. Section IV-D contains the results on the three multiplayer games, discussed separately.

For all results, a win is counted as a point and a draw is counted as half of a point for either of the players, with the exception of the 3-player games, where a draw counts as a third of a point for each player. All results contain a 95% confidence interval where applicable. Note that, in the case that an agent achieves a 0% winrate, then an additional single point is used to calculate the confidence interval to account for cases where the agent could still win given enough simulations.

A. Monte-Carlo Agents Comparison

In order to ensure that the MCTS agents behaved as expected, they were played against each other on Breakthrough as described in III. They were all given 1, 5 and 10 second decision times respectively. The results can be seen in Figure 3.

As is expected and from the results, MCTS-NST performs slightly better than MCTS-MAST the more time it is allocated per move. Both algorithms also perform significantly better than regular MCTS in Breakthrough.

B. Win Rates

Tables VI - XIV show the results of the corresponding AMS agents against all MCTS agents for all the two-player games. The results were obtained with 1 second per move for either agent in a matchup. Note that, despite having a win rate of 0%, the AMS agents could always win a game given enough simulations, and as such an error of ± 0.65 was assigned to every such case.

TABLE VI: Win rate (in %) for the AMS agents against all three MCTS agents in Connect Four.

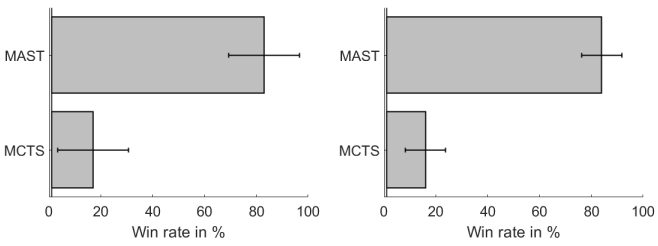
Connect Four	MCTS	MCTS-MAST	MCTS-NST
AMS	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Play-outs	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Priority	17.5 ± 2.94	42.3 ± 11.61	40.0 ± 2.77
AMS-MAST	0.0 ± 0.65	6.0 ± 4.93	2.0 ± 4.93
AMS-NST	2.8 ± 2.94	16.0 ± 7.84	7.0 ± 4.93

TABLE VII: Win rate (in %) for the AMS agents against all three MCTS agents in Breakthrough.

Breakthrough	MCTS	MCTS-MAST	MCTS-NST
AMS	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Play-outs	3.0 ± 3.64	1.0 ± 1.13	2.0 ± 0.65
AMS-Priority	6.0 ± 7.84	2.0 ± 0.65	1.0 ± 1.13
AMS-MAST	35.0 ± 1.96	5.0 ± 3.64	5.0 ± 3.62
AMS-NST	33.0 ± 4.80	7.0 ± 9.80	7.0 ± 1.96

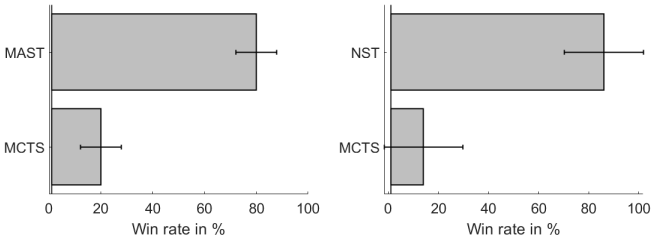
TABLE VIII: Win rate (in %) for the AMS agents against all three MCTS agents in Knightthrough.

Knightthrough	MCTS	MCTS-MAST	MCTS-NST
AMS	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Play-outs	21.0 ± 13.72	0.0 ± 0.65	0.0 ± 0.65
AMS-Priority	11.0 ± 1.96	0.0 ± 0.65	0.0 ± 0.65
AMS-MAST	36.0 ± 15.68	5.0 ± 3.64	4.0 ± 1.31
AMS-NST	40.0 ± 7.84	6.0 ± 5.58	4.0 ± 4.93



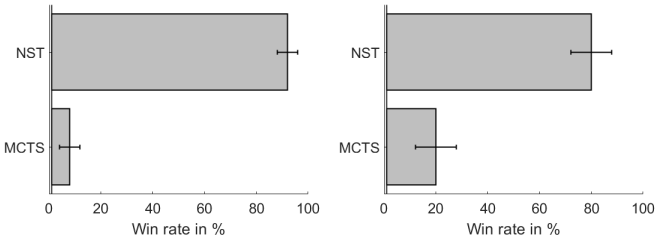
(a) MCTS vs MCTS-MAST, 1s per move in Breakthrough.

(b) MCTS vs MCTS-MAST, 5s per move in Breakthrough.



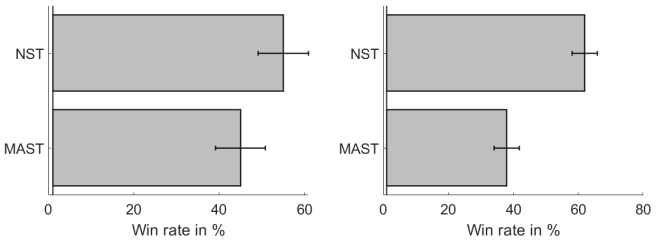
(c) MCTS vs MCTS-MAST, 10s per move in Breakthrough.

(d) MCTS vs MCTS-NST, 1s per move in Breakthrough.



(e) MCTS vs MCTS-NST, 5s per move in Breakthrough.

(f) MCTS vs MCTS-NST, 10s per move in Breakthrough.



(g) MCTS-MAST vs MCTS-NST, 1s per move in Breakthrough.

(h) MCTS-NST vs MCTS-NST, 5s per move in Breakthrough.

Fig. 3: MCTS variants matchup comparison with 1, 5 and 10 second decision times respectively.

TABLE IX: Win rate (in %) for the AMS agents against all three MCTS agents in Yavalath.

Yavalath	MCTS	MCTS-MAST	MCTS-NST
AMS	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Play-outs	1.0 ± 1.13	1.0 ± 1.13	1.0 ± 1.13
AMS-Priority	2.0 ± 3.92	6.0 ± 1.6	4.0 ± 3.92
AMS-MAST	15.0 ± 7.95	26.0 ± 11.76	26.0 ± 1.60
AMS-NST	10.0 ± 6.30	6.0 ± 1.60	14.0 ± 11.76

TABLE X: Win rate (in %) for the AMS agents against all three MCTS agents in Skirmish.

Skirmish	MCTS	MCTS-MAST	MCTS-NST
AMS	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Play-outs	0.0 ± 0.65	6.0 ± 7.53	0.0 ± 0.65
AMS-Priority	2.0 ± 2.36	10.5 ± 4.48	9.0 ± 3.58
AMS-MAST	2.0 ± 2.36	6.5 ± 0.98	2.0 ± 2.36
AMS-NST	1.0 ± 1.13	8.0 ± 3.92	9.0 ± 3.58

TABLE XI: Win rate (in %) for the AMS agents against all three MCTS agents in Tic-Tac-Chess.

Tic-Tac-Chess	MCTS	MCTS-MAST	MCTS-NST
AMS	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Play-outs	1.0 ± 1.13	0.0 ± 0.65	2.0 ± 2.36
AMS-Priority	14.0 ± 3.92	24.0 ± 15.68	19.0 ± 1.96
AMS-MAST	2.0 ± 2.61	1.0 ± 1.73	2.0 ± 2.26
AMS-NST	1.0 ± 1.13	1.0 ± 1.13	4.0 ± 7.84

TABLE XII: Win rate (in %) for the AMS agents against all three MCTS agents in Ultimate Tic-Tac-Toe.

Ultimate Tic-Tac-Toe	MCTS	MCTS-MAST	MCTS-NST
AMS	0.0 ± 0.65	0.0 ± 0.65	2.0 ± 2.36
AMS-Play-outs	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Priority	24.3 ± 5.45	40.0 ± 6.20	31.3 ± 7.48
AMS-MAST	24.3 ± 1.67	27.0 ± 10.43	31.3 ± 7.48
AMS-NST	28.5 ± 5.62	32.5 ± 7.22	30.5 ± 10.92

TABLE XIII: Win rate (in %) for the AMS agents against all three MCTS agents in Reversi/Othello.

Reversi/Othello	MCTS	MCTS-MAST	MCTS-NST
AMS	2.0 ± 2.36	1.0 ± 1.13	0.0 ± 0.65
AMS-Play-outs	5.0 ± 2.61	2.0 ± 0.65	1.0 ± 1.13
AMS-Priority	20.3 ± 1.67	12.3 ± 0.49	11.0 ± 1.13
AMS-MAST	42.5 ± 2.33	32.0 ± 6.79	33.0 ± 6.88
AMS-NST	37.0 ± 1.96	26.0 ± 1.27	35.0 ±9.80

TABLE XIV: Win rate (in %) for the AMS agents against all three MCTS agents in Hex.

Hex	MCTS	MCTS-MAST	MCTS-NST
AMS	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Play-outs	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-Priority	0.0 ± 0.65	0.0 ± 0.65	1.0 ± 1.13
AMS-MAST	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65
AMS-NST	0.0 ± 0.65	0.0 ± 0.65	0.0 ± 0.65

The suggested enhancement, using prioritization, seems to perform significantly better on most games compared to just adding play-outs to AMS. Since the main benefit of AMS-Priority is that it no longer uses all time resources on exploring a few children, this enhancement could be considered necessary for AMS when there exists a time limit.

From the results, AMS seemingly performs better in games with lower branching factor or depth, such as Connect Four, Reversi/Othello or even Ultimate Tic-Tac-Toe, while performing significantly worse in Hex, Yavalath and Skirmish. AMS with play-outs, prioritization, and MAST or NST can perform decently against the regular MCTS agent in Breakthrough and Knightthrough, but struggles to defeat MCTS-MAST and MCTS-NST. AMS-Priority alone performs decently against all MCTS variants in Tic-Tac-Chess, but all other AMS agents have under a 5% win rate on the same game.

C. Iterations

Tables XV and XVI show the average number of iterations taken during a match by each algorithm on the set of games used for experiments discussed previously, with one second per move for any agent. On average, MCTS is capable of doing up to 50 times more iterations than the AMS agents, even with similar enhancements. The only exception to this rule is Hex, where the MCTS agents perform around half the number of iterations of the AMS agents, despite MCTS dominating in Hex. A possible explanation of this is that MCTS will tend to prefer exploiting moves that lead it to a win as opposed to AMS.

Adding play-outs to AMS decreases the number of iterations it is capable of doing, up to 5 times in certain games. Further modifying AMS beyond this does not cause large differences in the number of iterations it can do, however certain games will further affect the number of iterations AMS can do with specific enhancements, such as AMS-NST having almost only half the iterations of AMS-Play-outs and AMS-Priority.

D. Multiplayer Games

For the multiplayer games, the matchups mainly consisted of one AMS agent against a standard MCTS agent in all 6 possible configurations (wherein at least 2 different agents are playing). The games used were Tic-Tac-Mo, Yavalde and Triad. The results in Tables XVII, XVIII, XIX, XX, XXI and XXII were obtained using 1 second per move per agent. Note that the results were separated into cases where there was only a single AMS agent playing against two MCTS agents and two AMS agents playing against a single MCTS agent. This is because there is a rather large difference between the win rates in those situations, as having more AMS agents playing at the same time increases the chances of the AMS agents winning. As discussed previously, AMS performs significantly fewer iterations than MCTS due to the high overhead of the algorithm in the context of game trees.

The reason why AMS-Play-outs and AMS seems to perform slightly better than in Tic-Tac-Mo and Triad is likely due to the fact that their matchups resulted mostly in draws. Neither algorithm was able to win more than a handful of games against MCTS in any configuration. Without counting the draws and across all configurations, AMS-Play-outs has a win rate of 1.3% in both games, while AMS has a win rate of 0.0%. A similar reason can be attributed to the abnormally high win rates of the other AMS variations, where games resulting in draws constitute almost half of the win rate for all variations. As Yavalde cannot end in a draw, this is no longer the case for it. Furthermore, in Yavalde, for the case where there are two AMS agents against a single MCTS agent, the win rates across both agents on average is significantly higher than the case with only a single AMS agent.

TABLE XV: Number of iterations performed on average by each algorithm in the relevant matchups in all games for the MCTS algorithms.

	MCTS	MCTS-MAST	MCTS-NST
Connect Four	563005 ± 53882	457653 ± 62104	457712 ± 73159
Breakthrough	39452 ± 6558	57744 ± 7340	54387 ± 12840
Knightthrough	88192 ± 15489	136836 ± 16516	126400 ± 23004
Yavalath	135665 ± 2288	91295 ± 2489	83358 ± 4017
Skirmish	17121 ± 995	22697 ± 2363	21982 ± 1658
Tic-Tac-Chess	1031912 ± 302926	896522 ± 147332	1325818 ± 119011
Ultimate Tic-Tac-Toe	194446 ± 30177	176212 ± 32268	162747 ± 40096
Reversi/Othello	379532 ± 13478	354492 ± 13213	350291 ± 11648
Hex	5973 ± 443	8615 ± 776	5695 ± 636

TABLE XVI: Number of iterations performed on average by each algorithm in the relevant matchups in all games for the AMS algorithms.

	AMS	AMS-Play-outs	AMS-MAST	AMS-NST	AMS-Priority
Connect Four	21057 ± 444	13092 ± 453	10382 ± 197	7240 ± 344	13047 ± 1319
Breakthrough	7088 ± 333	1600 ± 19	2121 ± 63	1762 ± 35	1325 ± 18
Knightthrough	7023 ± 176	2287 ± 26	2666 ± 19	2521 ± 67	1893 ± 18
Yavalath	7543 ± 75	7213 ± 68	5620 ± 67	4659 ± 49	8372 ± 148
Skirmish	3996 ± 117	1803 ± 56	1675 ± 82	1580 ± 55	1764 ± 82
Tic-Tac-Chess	21192 ± 1654	15970 ± 1863	8312 ± 1143	8048 ± 689	12867 ± 1813
Ultimate Tic-Tac-Toe	14218 ± 326	5568 ± 127	4588 ± 281	3954 ± 169	5786 ± 599
Reversi/Othello	16333 ± 511	8890 ± 460	7784 ± 517	10574 ± 347	8756 ± 505
Hex	10577 ± 1212	9994 ± 1475	9324 ± 912	9198 ± 950	9532 ± 911

TABLE XVII: AMS win rate (in %) against Vanilla MCTS in Tic-Tac-Mo in all possible configurations with a single AMS agent. Average MCTS Iterations: 751735± 70734.

Tic-Tac-Mo	Win Rate	Iterations
AMS	2.2 ± 3.72	17208 ± 2870
AMS-Play-outs	1.6 ± 2.43	27517 ± 6318
AMS-Priority	21.6 ± 20.11	20486 ± 3108
AMS-MAST	11.8 ± 10.60	26984 ± 4513
AMS-NST	7.8 ± 7.13	16393 ± 4336

TABLE XVIII: AMS win rate (in %) against Vanilla MCTS in Tic-Tac-Mo in all possible configurations with two AMS agents. Average MCTS Iterations: 611278 ± 112772.

Tic-Tac-Mo	Win Rate	Iterations
AMS	4.7 ± 4.71	24548 ± 2012
AMS-Play-outs	5.1 ± 5.14	39653 ± 2436
AMS-Priority	60.7 ± 46.18	19516 ± 1769
AMS-MAST	65.3 ± 54.24	20384 ± 1907
AMS-NST	65.8 ± 48.79	18524 ± 1750

TABLE XIX: AMS win rate (in %) against Vanilla MCTS in Yavalde in all possible configurations with a single AMS agent. Average MCTS Iterations: 64883 ± 9091.

Yavalde	Win Rate	Iterations
AMS	0.0 ± 0.49	6278 ± 81
AMS-Play-outs	0.7 ± 1.31	6462 ± 230
AMS-Priority	17.3 ± 20.54	5925 ± 195
AMS-MAST	10.0 ± 2.26	5880 ± 105
AMS-NST	3.3 ± 6.53	4656 ± 151

TABLE XX: AMS win rate (in %) against Vanilla MCTS in Yavalde in all possible configurations with two AMS agents. Average MCTS Iterations: 55509 ± 7364.

Yavalde	Win Rate	Iterations
AMS	18.0 ± 14.84	6355 ± 330
AMS-Play-outs	39.3 ± 28.30	6114 ± 125
AMS-Priority	51.3 ± 31.22	6093 ± 173
AMS-MAST	43.3 ± 7.28	5637 ± 237
AMS-NST	59.3 ± 14.55	4864 ± 43

TABLE XXI: AMS win rate (in %) against Vanilla MCTS in Triad in all possible configurations with a single AMS agent. Average MCTS Iterations: 206274 ± 13000.

Triad	Win Rate	Iterations
AMS	5.3 ± 4.71	11386 ± 165
AMS-Play-outs	7.6 ± 1.15	2755 ± 338
AMS-Priority	21.6 ± 8.41	1850 ± 92
AMS-MAST	28.2 ± 18.70	2241 ± 137
AMS-NST	17.3 ± 10.15	1863 ± 209

TABLE XXII: AMS win rate (in %) against Vanilla MCTS in Triad in all possible configurations with two AMS agents. Average MCTS Iterations: 171891 ± 46460.

Triad	Win Rate	Iterations
AMS	10.7 ± 5.23	11329 ± 257
AMS-Play-outs	21.6 ± 11.74	2768 ± 317
AMS-Priority	55.3 ± 10.96	2544 ± 346
AMS-MAST	52.9 ± 21.5	2264 ± 237
AMS-NST	71.2 ± 6.13	1882 ± 183

V. Conclusion and Future Work

In this article, we explored how Monte-Carlo Tree Search (MCTS) and a tree-search version of Adaptive Multistage Sampling (AMS) perform on a series of games with several characteristics. Furthermore, we implemented some common enhancements for MCTS and adapted them for AMS, specifically Move-Average Sampling Technique (MAST) and N-gram Selection Technique (NST).

Based on our experiments in both two and three player games, we note that AMS seems to perform better in finite-depth, lower branching factor, converging cases where it can fully explore the search space. Otherwise, MCTS is the better algorithm overall in most sequential games. Optimizing AMS and improving its behaviour on game search trees could lessen the performance gap between the two algorithms, however this would require some additional modifications to the original algorithm proposed by Chang et al. [5]. For example, modifying and optimizing the AMS search depth for each game could improve its performance.

In terms of enhancements, we attempted to modify AMS to utilize MCTS-style play-outs as well as making it prioritize exploration of unknown branches. Based on this, we further explored how AMS and MCTS differ with MAST and NST utilizing the exact same hyperparameter values with no optimization and no move decay. Optimizing these values for the specific algorithm for specific games could further provide valuable inisght into which algorithm benefits the most from said enhancements.

To conclude, in the context of game trees, AMS cannot be matched to MCTS in a one-to-one fashion. AMS is, in fact, unsuitable as a game playing algorithm with large depths and large branching factors. The games in which AMS performed significantly better were games where either the depth was low, or the branching factor decreased significantly in the latter half of the game. However, this was only after adding MCTS-style play-outs to AMS. The main exception to this rule was Hex, which is likely due to the fact that MCTS keeps expanding the tree and can quickly find possible solutions while AMS is stuck trying to explore inefficient moves.

There exist many enhancements for MCTS, both for domain-independent and domain-dependent, which could be adapted and implemented into AMS. Whether one algorithm performs better than the other with other enhancements is an interesting path to focus on in future research.

References

[1] M. J. Tak, M. H. Winands, and Y. Bjornsson, "N-grams and the last-good-reply policy applied in general game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 73–83, 2012.

[2] M. H. Winands, "Monte-carlo tree search in board games," *Handbook of Digital Games and Entertainment Technologies*, pp. 47–76, 2017.

[3] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[4] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[5] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, "An adaptive sampling algorithm for solving markov decision processes," *Operations Research*, vol. 53, no. 1, pp. 126–139, 2005.

[6] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[7] C. Browne, E. Powley, D. Whitehouse, S. 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*, vol. Vol. 4, no. No. 1, pp. 1–43, 2012.

[8] H. S. Chang, J. Hu, M. C. Fu, and S. I. Marcus, *Simulation-based algorithms for Markov decision processes*. Springer Science & Business Media, 2013.

[9] Y. Bjornsson and H. Finnsson, "Cadiaplayer: A simulation-based general game player," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 4–15, 2009.

[10] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Bandits all the way down: Ucb1 as a simulation policy in monte carlo tree search," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. IEEE, 2013, pp. 1–8.

[11] C. Manning and H. Schutze, *Foundations of statistical natural language processing*. MIT press, 1999.

[12] D. J. Soemers, C. F. Sironi, T. Schuster, and M. H. Winands, "Enhancements for real-time monte-carlo tree search in general video game playing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–8.

[13] D. M. Breuker, "Memory versus search in games," 1998.

[14] M. Stephenson, E. Piette, D. J. Soemers, and C. Browne, "An overview of the ludii general game system," in *2019 IEEE Conference on Games (CoG)*. IEEE, 2019, pp. 1–2.

[15] É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne, "Ludii – the ludemic general game system," in *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, ser. Frontiers in Artificial Intelligence and Applications, G. D. Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, Eds., vol. 325. IOS Press, 2020, pp. 411–418.

APPENDIX

A. Adaptive Multi-stage Sampling

The Adaptive Multi-stage Sampling algorithm (AMS) operates in various stages. As mentioned in the subsection about AMS, Section II-B, there exist an initialization and a loop phase. Figure 4 shows the various stages of the algorithm, with a tree of depth 2 and a branching factor of 2. The first step of the algorithm is to initialize every move once, as is shown in step 2,3 and 4 of the figure. A bold node is an initialized node. Hereafter, the algorithm enters the loop phase of the left action of the root node. In this loop phase, a move is selected according to the selection strategy and it enters a node again that has been initialized. While entering this node, it repeats all the necessary steps once more. In this case, evaluating the node. This loop phase is repeated for a number of iterations x.

After finishing the loop phase, the algorithm moves one layer up and performs the remaining move in the initialization phase of the root node. Steps 6, 7, 8 and 9 are similar to steps 2, 3, 4 and 5. After finishing the loop phase of the right action with respect to the root node, this node is now initialized, as shown in step 10 of the figure. In step 10, the root node enters the loop phase. Either the left or right action is selected according to the selection strategy and that steps 2,3,4 and 5 or steps 6,7,8, and 9 are repeated for x iterations.

During the experiments, a time limit of 1 second has been put per move for an agent. For an AMS agent, this means that it can do as many iterations as it want on one move, until the time lime has been met, because the number of iterations has been put to infinity. Therefore, as an enhancement, AMS-Priority prioritizes to explore all the moves. To do this, the number of iterations has been limited to 50 iterations in the loop phase for all the nodes descending from the root node. In the root node, still the old constraint of infinite many iterations hold.

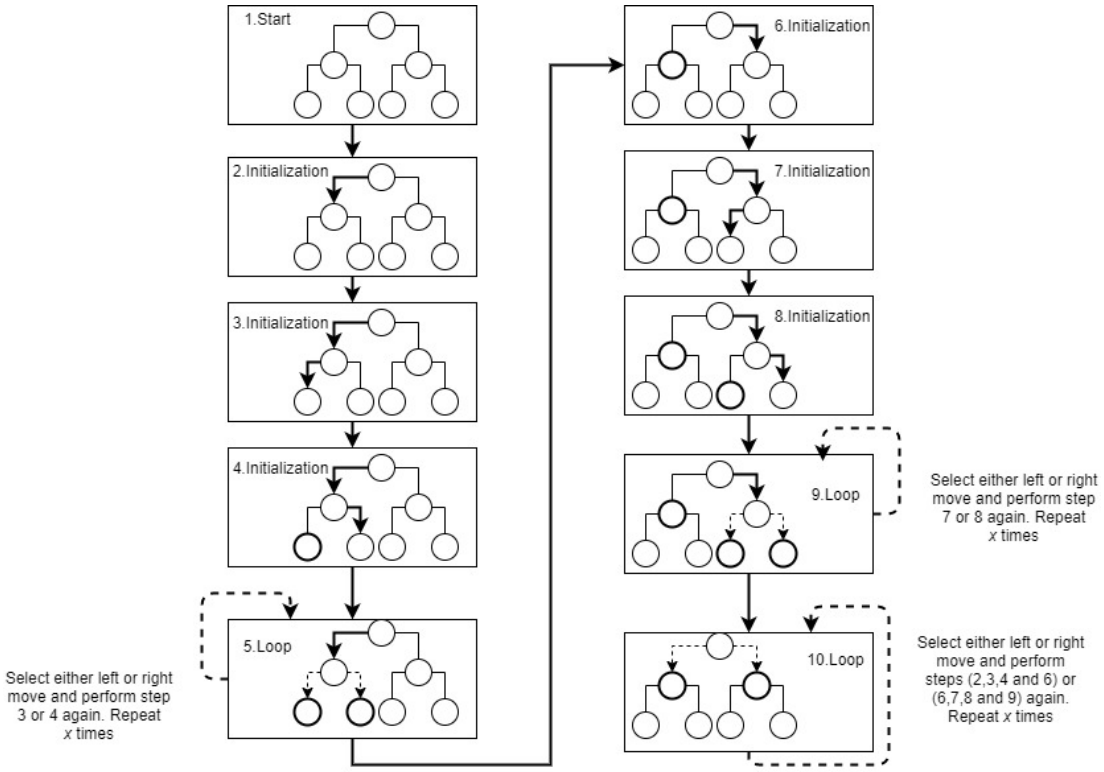


Fig. 4: Adaptive Multi-stage Algorithm explained, bold arrows are the moves being considered right now, whereas bold nodes are initialized nodes.

B. Game Description

This section contains the Ludii descriptions for the used games as found on the Ludii website / engine [14].

- **Connect Four** - Played on a vertically placed grid of 7x6, where colored disks are dropped from the top of the grid. Players alternate dropping discs, which fall to the bottom of the column in which they are dropped. The first player to create a row of four disks in their color wins.
- **Breakthrough** - Played on an 8x8 board with a double contingent of chess pawns. Pieces move forward one orthogonally or diagonally. Pieces can capture by moving diagonally. The first player to reach the opponent's edge of the board wins. A player also can win if they capture all of the opponent's pieces.
- **Knightthrough** - Almost the same as Breakthrough, but pieces move as knights in Chess. The goal is to be the first player to reach the opposite side of the board from the starting position.
- **Yavalath** - Played on a hexagonal board with five spaces per side. It can be played by two or three players. Players alternate turns placing pieces on one of the spaces. The first player to place four in a row without first making three in a row wins.
- **Skirmish** - The rules are the same as in Chess, without checkmate. The game is won either after 100 turns or when one player can no longer move, by the player with the most pieces.
- **Tic-Tac-Chess** - Players take turns placing a piece of theirs at an empty cell. When all pieces have been placed, players take turns moving one of their pieces. The pieces move like the equivalent Chess pieces but do not capture. The pieces used are the King, Queen and Rook. Any piece can hop over an adjacent enemy piece to an empty cell beyond (without capturing it). First to make a line of 3 of their pieces, at any time, wins the game.
- **Ultimate Tic-Tac-Toe** - Each small 3x3 Tic-Tac-Toe board is referred to as a local board, and the larger 3x3 board is referred to as the global board. The game starts with X playing wherever they want in any of the 81 empty spots. This move 'sends' their opponent to its relative location. For example, if X played in the top right square of their local board, then O needs to play next in the local board at the top right of the global board. O can then play in any one of the nine available spots in that local board, each move sending X to a different local board. If a move is played so that it is to win a local board by the rules of normal Tic-Tac-Toe, then the entire local board is marked as a victory for the player in the global board. Once a local board is won by a player or it is filled completely, no more moves may be played in that board. If a player is sent to such a board, then that player may play in any other board. Game play ends when either a player wins the global board or there are no legal moves remaining, in which case the game is a draw.
- **Reversi** - Reversi (also known as Othello) is played on an 8x8 board. Pieces are double-sided, with each side distinct in some way from the other. Each side designates ownership of that pieces to a certain player when face-up. Play begins with the players taking turns placing pieces into the central four squares until they are full. Then players may place their pieces next to an opponent's piece, a long as a straight line can be drawn between the new piece and an existing piece belonging to that player that goes through the opponent's piece. The opponent's pieces between the new piece and the old piece are then flipped and now belong to the player who just played. If a player cannot make a legal move, they pass. Play continues until the board is full or neither player cannot make a legal move. The player with the most pieces on the board wins.
- **Hex** - Players take turns placing their pieces on one space on the board, with the goal of connecting the opposite sides of the board corresponding to the color of their pieces with a line of pieces.
- **Tic-Tac-Mo** - Tic-Tac-Mo uses the same rules as Tic-Tac-Toe, but the third player plays with a "y". The first player to make three in a row wins.
- **Yavalde** - Yavalde is the three player version of Yavalath. White player takes a red piece and places a white piece on top of it, placing the stack before them. Black player takes a white piece and places a black piece on top of it, placing the stack before them. Red player takes a black piece and places a red piece on top of it, placing the stack before them. Starting with White, players take turns placing a piece of their colour on an empty space of the board until the victory condition is reached. The aim of the game is to create a 5-in-a-row containing both and only both colours of your stack.
- **Triad** - Triad is an abstract board game for three players which features an interesting move mechanic; the players dictate move order as the game progresses. The game is played on a tricoloured hexagonal grid with five cells per side. The three players are called Red, Green and Blue, and each player owns the cells and the pieces that bear their colour. Each turn the current player must move, capture and drop. The current player must move one of their pieces in a straight line in any of the six hexagonal directions to land on a vacant foreign cell; any intervening cells must also be empty. The opponent who owns the landing cell becomes the candidate and the other opponent becomes the bunny. All opponents' pieces immediately adjacent to the landing cell are captured and removed from the board. The current player must make the move that captures the most pieces each turn, but may choose amongst equals. This is called the max capture rule. The moving player must then drop a bunny piece on any empty cell, unless a player has just been eliminated. The candidate becomes the next player to move. Goal: Play stops the moment any player is eliminated. The game is won by the player

with the most pieces left, else is a tie between the two remaining players if they are both left with the same number of pieces.

C. C-Tuning

C-tuning is performed in order to determine and utilize the best value for the standard MCTS agent as a baseline to compare the other algorithms to. The C parameter represents how much weight should be assigned to the exploration vs exploitation factor in the UCB1 and UCT formulas. Figure 5 shows the results for the experiment. The results were taken as an average between Breakthrough, Hnefatafl and Reversi/Othello. Using a C parameter of 0.4 provides the best balance of exploration versus exploitation for the standard MCTS agent, such that it can be used as a baseline for comparison with the other algorithms. Other values, such as 0.6, 1 and 1.6, while performing better on average than 0.4, would provide MCTS too much of a comparative advantage against the various AMS agents.

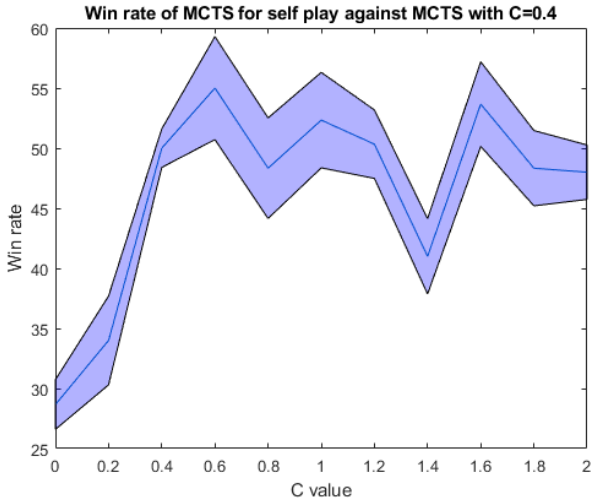


Fig. 5: C-Tuning of the Standard MCTS agent using self-play. The shaded region represents a 95% confidence interval.