Monte Carlo Search algorithms for Order & Chaos

Mariem Aalabou[a] **and Emmarius Delar**[a]

[a]*M2 IASD*

April, 2025

Abstract—For this project, we focused on the game Order and Chaos, which is very challenging due to its unique rules. Like in tic-tac-toe, players choose between two symbols, O or X. However, the particularity of this game lies in the fact that both players are allowed to play either symbol. The objectives of the two players differ: Player 1 (Order) aims to create a winning line of five identical symbols in a row, while Player 2 (Chaos) tries to prevent this from happening, aiming for a draw instead. In this project, we experimented with various Monte Carlo Tree Search algorithms, such as UCT, Enhanced Playout Methods, RAVE, GRAVE, NRPA, and NMCS. Due to the unbalanced advantage favouring Order, we experimented with a MCTS method in Reinforcement Learning, called Sarsa-UCT. However, the results showed that this algorithm fails to reproduce Chaos's gameplay effectively.

1. Introduction

Monte Carlo Tree Search (MCTS) is a powerful approach to create AI agents capable of playing various games at a high level, notably board games such as **Order & Chaos**. MCTS methods use Monte Carlo simulations to explore multiple possible move sequences starting from the current game state, and select the one that maximizes the chances of winning. This process can be represented as a tree (see Fig.1), where each node corresponds to an action taken at a certain point, from which different sub-branches are simulated. The algorithm then chooses the action that leads to the most promising path.

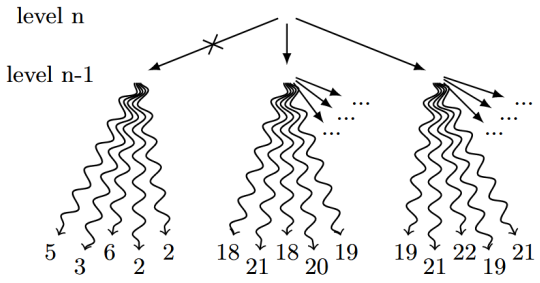


Figure 1. Level n LNMCS pruning a search subtree and launching n-1 LNMCS on surviving search subtrees by Roucairol[2]

This report is organized as follows. In Section 2, we introduce the game and its implementation. In Section 3, we present the different algorithms used. Section 4, we analyze the results and compare the performance in terms of speed, resource requirements, and head-to-head results. Finally, in Section 5 dedicated to the Sarsa-UCT algorithm before conclude in Section 6.

2. Order & Chaos

2.1. Game description

Order & Chaos[1] is a two-player board game introduced in 1981 by Stephen Sniderman, in which Order and Chaos face off. The game begins with an empty board, and Order plays first. As a more complex version of tic-tac-toe, it is played on a 6×6 board, consisting of 36 different cells, where players can choose to play either an O or an X on each turn. Order's objective is to create a horizontal, vertical, or diagonal line of five identical symbols in a row, while Chaos aims to block Order and force a draw. Once a symbol is placed, it cannot be removed.

[1] https://en.wikipedia.org/wiki/Order_and_Chaos

Order seems to have a significant advantage in this game, so tried to develop algorithms that can improve Chaos's chances of winning.

2.2. Implementation

The game has been implemented using Python and common libraries such as `numpy` and `deepcopy`. The `Move` class has been defined according to the rules of the game. It controls the symbol used and checks whether a move is valid (i.e., whether the player selects a valid symbol and places it on an empty cell of the board).

The `Board` class was defined to manage the overall game logic. It handles the player's turn, the current state of the board, initialisation, the number of moves played, and generates a hash code at each move to uniquely identify the board state.

The `legalMoves()` function lists all possible moves available to the current player on the current board. This function is essential for the simulations performed by the various algorithms, especially the playout strategy integrated into the `Board` class, which randomly selects a move from the `legalMoves()` list using a uniform distribution.

Two functions, `win()` and `draw()`, are used to determine whether Order or Chaos has won the game.

Our implementation includes an interactive testing framework within the notebook that allows to evaluate our algorithms across various parameters. The interactive GUI provides comprehensive tools for algorithm selection, parameter configuration, and result visualisation through three main components. See Figure 2.

3. Monte-Carlo search algorithms

3.1. UCT

Upper Confidence bounds for Trees (UCT) is almost surely the most siplest exploration trees algorithm. The algorithms uses playouts for simulate random games at differentes after play random first move and return the best move according that maximise the function $\frac{w_i}{n_i} + c\sqrt{\frac{\log t}{n_i}}$, where w_i denote the number of wins after have played the i-th move, n_i the number of simulations after have played the i-th move, c an exploration parameter, and t the total number of simulations for the parent node.

3.2. RAVE

Rapid Action Value Estimation (RAVE) enhances the standard UCT algorithm by using the All-Moves-As-First (AMAF) heuristic. This approach is especially useful for Order & Chaos where the value of a move often remains consistent regardless of when it is played during a game.

Our RAVE implementation uses a weighted combination of standard UCT statistics and AMAF statistics:

$$V_{RAVE}(s,a) = (1-\beta)\cdot V_{UCT}(s,a) + \beta \cdot V_{AMAF}(s,a) \tag{1}$$

Where $V_{UCT}(s,a)$ is the standard UCT value (wins/visits ratio plus exploration term), $V_{AMAF}(s,a)$ is the AMAF value (AMAF wins/visits ratio), and β is a weighting parameter that balances between the two estimates.

The parameter β is dynamically adjusted based on the relative confidence in the UCT and AMAF statistics:

$$\beta = \frac{n_{AMAF}}{n_{UCT} + n_{AMAF} + 4\cdot\beta_0^2\cdot n_{UCT}\cdot n_{AMAF}} \tag{2}$$

where n_{UCT} is the number of UCT visits, n_{AMAF} is the number of AMAF visits, and β_0 is a constant parameter (set to 0.5 in our implementation). This formula gives more weight to AMAF statistics early in the search and gradually shifts to standard UCT statistics as more information is gathered.

The key innovation in our RAVE algorithm is in how we track and update the AMAF statistics. For each simulation, we maintain a record of all moves played during the playout and the nodes they affected. After the simulation, we update the AMAF statistics for all siblings of each visited node, effectively sharing information across different branches of the search tree.

3.3. GRAVE

Generalized Rapid Action Value Estimation (GRAVE) extends RAVE by addressing a key limitation: in large search spaces, many nodes have sparse AMAF statistics. GRAVE solves this by using AMAF statistics from ancestor nodes when a node's own AMAF statistics are insufficient.

Our GRAVE implementation maintains a global table of AMAF statistics indexed by state-action pairs:

$$\text{amaf\_wins}[(s, a)] \text{ and } \text{amaf\_visits}[(s, a)] \tag{3}$$

Where (s, a) represents a state-action pair, with the state encoded as a string representation of the board and the action as a tuple of $(x, y, symbol)$.

During node selection, GRAVE modifies the RAVE formula by using the following criteria:

$$V_{GRAVE}(s, a) = (1 - b) \cdot V_{UCT}(s, a) + b \cdot V_{AMAF}(ref(s), a) \tag{4}$$

Where $ref(s)$ is a reference state that has accumulated sufficient AMAF statistics.

The weighting parameter b is calculated similarly to RAVE but using the reference state's AMAF statistics:

$$b = \frac{n_{AMAF}(ref(s), a)}{n_{UCT}(s, a) + n_{AMAF}(ref(s), a) + 4 \cdot \beta^2 \cdot n_{UCT}(s, a) \cdot n_{AMAF}(ref(s), a)} \tag{5}$$

Our experimental results showed that GRAVE consistently outperformed both UCT and RAVE, especially when playing as Chaos. The global perspective provided by GRAVE allows for more effective knowledge transfer between different parts of the search tree, which is particularly valuable in a game with high strategic complexity like Order & Chaos.

A key technical contribution in our GRAVE implementation is the efficient state encoding using string representation, which allows for fast lookups in the global AMAF table while maintaining the board position's structure. Additionally, our implementation carefully manages the trade-off between using local and reference AMAF statistics, ensuring that both long-term strategic patterns and short-term tactical opportunities are properly valued.

3.4. Enhanced Playout Methods

In standard Monte Carlo Tree Search algorithms, the playout (or simulation) phase typically uses random move selection to estimate the value of a position. However, this approach often leads to unrealistic play, especially in strategic games like Order & Chaos. To address this limitation, we implemented two enhanced playout methods that incorporate domain-specific knowledge:

3.4.1. Heuristic Playout

The heuristic playout method prioritizes winning moves for Order and blocking moves for Chaos through the `find_tactical_move` function. Before making a random move, the algorithm checks:

- For Order: If there is an immediate winning move (creating a line of 5 identical symbols)
- For Chaos: If Order has a potential winning move in the next turn that needs to be blocked

Our implementation first looks for immediate winning moves when playing as Order. For Chaos, it temporarily changes the board state to simulate Order's turn and checks for potential threats, storing the positions that need to be blocked. This tactical awareness significantly improves the win rate compared to purely random playouts.

3.4.2. Pattern-Based Playout

For even more sophisticated simulation, we developed a pattern-based playout strategy that evaluates moves based on pattern recognition. The pattern-based playout leverages two key components:

1. **Move Evaluation Function**: Each candidate move is scored based on:
    - Formation of consecutive symbols in all directions (horizontal, vertical, and diagonal)
    - Higher scores for longer sequences (e.g., 100 points for 5 in a row, 10 points for 4 in a row)
    - Bonus points for moves adjacent to existing symbols (promoting clustered play)
2. **Exploration vs. Exploitation**: A tunable `exploration_rate` parameter (default: 0.2) balances between random exploration and exploitation of the highest-scoring moves

The pattern-based playout achieved a perfect 100% win rate in our tests when playing as Order, compared to the 95% of the heuristic playout and 80.6% of the random playout. This significant improvement demonstrates the value of incorporating domain knowledge into the simulation phase of MCTS algorithms.

Both enhanced playout methods can be used with any of the tree search algorithms (UCT, RAVE, GRAVE, etc.), providing a flexible way to improve performance regardless of the specific tree search strategy employed.

3.5. NRPA

Nested Rollout Policy Adaptation (NRPA) represents a significant departure from traditional MCTS approaches. Instead of building and traversing a search tree, NRPA iteratively refines a policy (a probability distribution over moves) through nested self-play. Our implementation of NRPA for Order & Chaos follows this recursive approach.

The NRPA function initializes a default policy where all moves are equally likely, then calls the nested search function. The nested search function implements the recursive structure of NRPA. At the base level (level = 0), it performs a playout using the current policy. At higher levels, it repeatedly calls itself at the next lower level, updating the policy based on the best sequence found.

The key aspects of our implementation include:

- **Policy Representation**: The policy is represented as a mapping from move keys (x, y, symbol) to numerical weights
- **Move Selection**: During playouts, moves are selected probabilistically according to their weights using softmax:

$$P(a|s) = \frac{e^{policy(a)}}{\sum_{a' \in A(s)} e^{policy(a')}} \tag{6}$$

- **Policy Adaptation**: After each simulation, the policy is updated based on the moves in the best sequence:

$$policy'(a) = policy(a) + \alpha \cdot result \cdot \mathbf{1}_{a \in sequence} \tag{7}$$

where α is the learning rate (set to 0.1 in our implementation)

A notable advantage of NRPA is its ability to discover and exploit patterns across different board positions, which aligns well with the strategic nature of Order & Chaos. However, NRPA's performance is sensitive to the number of iterations and nesting level. In our experiments, we found that a level of 1 with 20 iterations provided a good balance between computational efficiency and playing strength.

3.6. NMCS

Nested Monte Carlo Search (NMCS) is a recursive algorithm that explores the game tree through nested levels of search. Unlike MCTS algorithms that build an explicit tree, NMCS performs depth-first exploration with increasingly sophisticated playouts at each level.

Our NMCS implementation works recursively. If the board is in a terminal state, it returns the score. If at level 0, it makes a random move and returns the result of a random playout. At higher levels, it tries each possible move, recursively calls itself at the next lower level for each move, and selects the move with the highest score.

The key advantages of NMCS for Order & Chaos include:

- **Efficient Exploration**: NMCS is particularly effective at finding winning sequences for Order, as it can systematically explore promising lines of play
- **Low Memory Requirements**: Unlike MCTS algorithms that store a search tree, NMCS only needs to track the current best sequence at each recursive level
- **Deterministic Results**: At a given search level, NMCS will always return the same result for a given board position, making it more predictable than randomized approaches

Our experiments showed that NMCS performed exceptionally well when playing as Order, achieving a near 100% win rate even at modest search depths (level=1). The algorithm's systematic approach to exploring winning sequences makes it particularly suited to the offensive objectives of the Order player. For Chaos, however, the defensive nature of the role makes NMCS less effective than algorithms like GRAVE that better handle strategic blocking patterns.

4. Experimental results

4.1. Methodology

We tested the performance of the algorithms by running 100 iterations of 50 games, where each algorithm took the role of Order and then Chaos. Various metrics were collected, such as the total running time of the algorithm to play each of its moves during the game, and the win ratio as Order and as Chaos. In the next subsection, we analyze the performance of each algorithm against a random player, followed by a comparison against a heuristic player. The final section is dedicated to the performance of the algorithms against each other. Additional figures for this section are provided in Appendix A.

All the experiments were run on an Acer Swift 3 laptop with an AMD Ryzen 7 4700U CPU and 16GB of RAM over a period of 8 hours.

4.2. Algorithms vs Random

According to Fig. 3, against a random player, the algorithms playing as Order perform well, with a minimum win rate of 80%, and the strongest being NMCS and the Heuristic algorithm. However, as Chaos, the results are significantly different. Even against a random player, the algorithms cannot achieve more than a 25% win rate, likely due to the inherent advantage of playing as Order in this game.

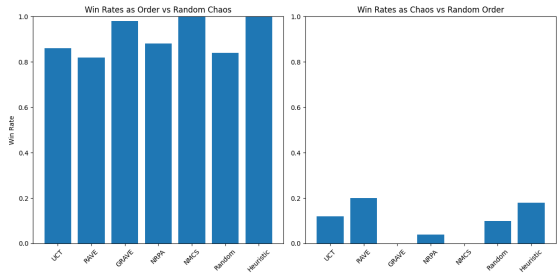


Figure 3. General comparison of algorithms against Random play

4.3. Algorithms vs Heuristic

As expected, according to Fig. 4, against a stronger player, the performance of each algorithm decreases, except for NMCS, which maintains a 100% win rate when playing as Order. As Chaos, the GRAVE algorithm manages to win a few games

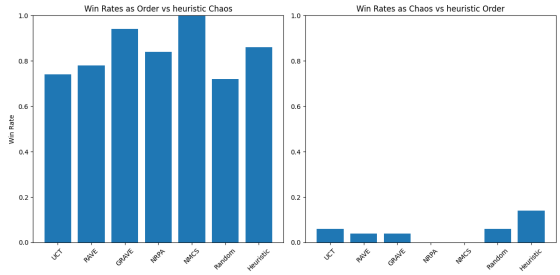


Figure 4. Comparison using heuristic opponent

4.4. Algorithms vs Algorithms

As Chaos role no one of the algorithm out perform Heuristic algorithm. That mean Heuristic is the better algorithm to play as Chaos against the other algortihms. For more, against GRAVE and NMCS as Chaos role, each algorithms have a win rate neir to 100%. On the other hand, for NMCS as Order role, the win rate for each algorithm as Chaos role is less than 10% and atmost 0 for UCT, GRAVE, NRPA, NMCS, and Random. That mean NMCS is the better algorithm to play as Order against the other algortihms. The figures is disponible in Appendix A.

5. Advanced algorithm

Our objective in this section its to provide an algorithm who perform well against Heuristic Chaos and NMCS Order. Building a powerful algorithm for unbalanced two-player games is particularly challenging. Before exploring entirely new algorithms, we first attempted to refine the UCT approach by incorporating domain knowledge or priors about the game. For example, we reduced the set of possible moves at a given state by selecting only those that are legally close to previously played moves, or penalized actions that lead to aligning three identical symbols. However, these heuristics ultimately degraded UCT's performance. This led us to investigate more advanced methods, notably the Sarsa-UCT algorithm.

5.1. Sarsa-UCT Algorithm

Sarsa-UCT is a temporal-difference learning algorithm adapted to Monte Carlo Tree Search (MCTS), introduced by Vodopivec et al. [1]. It combines the exploration strength of UCT with the learning capabilities of Sarsa. This hybrid approach allows the agent to not only explore the decision tree efficiently but also to learn an effective policy through temporal-difference backups and eligibility traces. The structure of the algorithm is illustrated in Figure 5.

5.2. Experimental Results

This algorithm is computationally expensive, so the Sarsa component was trained over only 100 iterations, whereas the original paper used at least 1000 iterations. For training, we allowed the agent to play against NMCS, a Random strategy, and a Heuristic algorithm—the latter being our most effective approach for the Chaos player and also the hardest to beat.

After training, the algorithm was evaluated against the same set of opponents. As shown in Figure 6, the Sarsa-UCT algorithm does not outperform the Heuristic method (our best-performing algorithm for Chaos), and it even underperforms compared to NMCS (our best-performing algorithm for Order).

6. Conclusion

In this project, we explored various Monte Carlo Tree Search algorithms for the Order & Chaos game. Our experiments clearly demonstrated that NMCS excels as Order (nearly 100

Enhanced playout methods significantly improved all algorithms, demonstrating the value of domain-specific knowledge in MCTS. Our attempt to use Sarsa-UCT did not yield the performance improvements we had hoped for, highlighting the challenge in developing effective algorithms for the disadvantaged player.

While we primarily focused on Sarsa-UCT in this project, other promising directions for improving Chaos's performance could include pattern-based opposition strategies that specifically target Order's winning patterns. Future work might also explore neural network evaluation functions, hybrid MCTS approaches, and investigations of alternative board sizes for better game balance.

This research provides valuable insights into the strengths and limitations of Monte Carlo search algorithms in asymmetric game scenarios, where structural advantages significantly impact algorithm performance.

References

[1] T. Vodopivec, S. Samothrakis, and B. Ster, "On monte carlo tree search and reinforcement learning", *Journal of Artificial Intelligence Research*, vol. 60, pp. 881–936, 2017.

[2] M. Roucairol, "Monte-carlo tree search applied to structure generation", Ph.D. dissertation, Université Paris sciences et lettres, 2024.

A. Algorithm Comparison

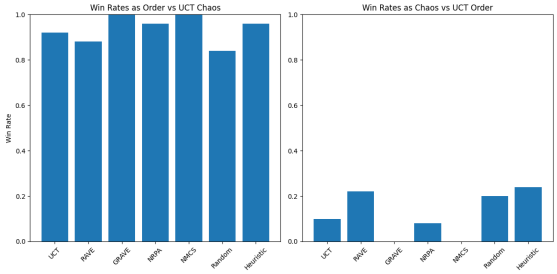


Figure 7. UCT algorithm performance

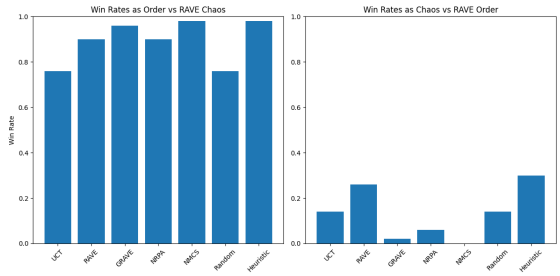


Figure 8. RAVE algorithm performance

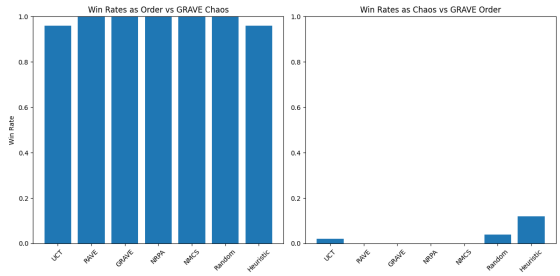


Figure 9. GRAVE algorithm performance

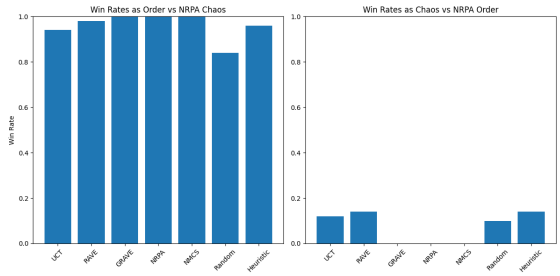


Figure 10. NRPA algorithm performance

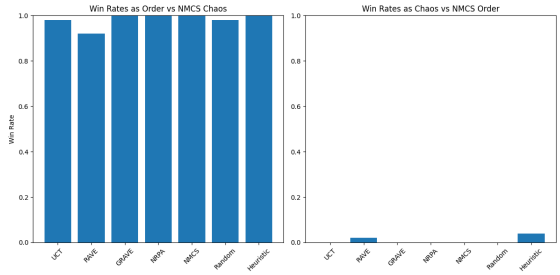
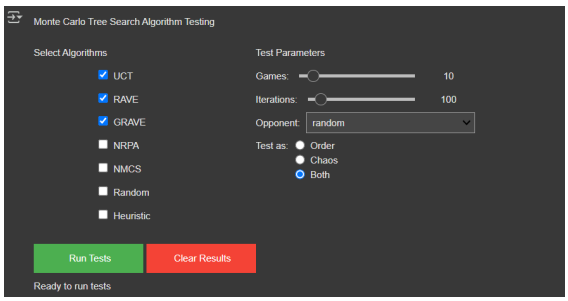
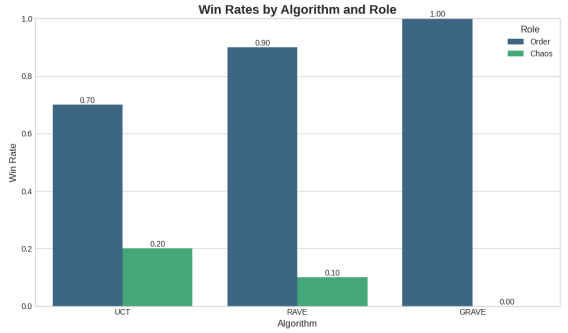


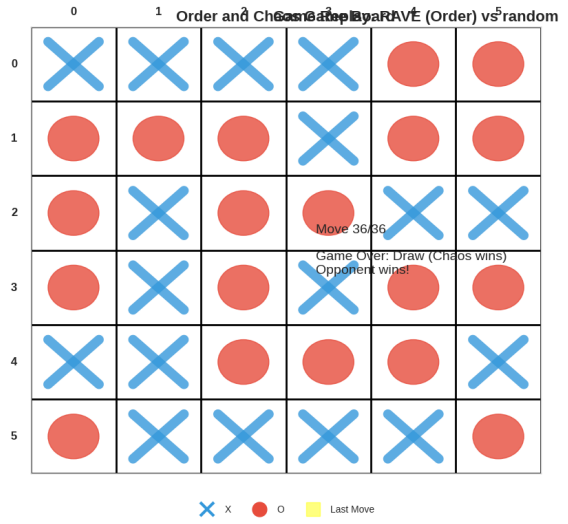
Figure 11. NMCS algorithm performance



(a) Algorithm selection and parameter configuration panel



(b) Performance visualization and comparative analysis



(c) Game replay and board visualization

Figure 2. The interactive testing GUI implemented in the Jupyter notebook

Algorithm 1 An iteration of a tabular offline Sarsa-UCT(λ) algorithm that builds a tree and evaluates actions with afterstates.

parameters: C_p, α (optional), $\gamma, \lambda, V_{\text{init}}, V_{\text{playout}}$ ▷ V_{init} and V_{playout} can be constants or functions
global variable: $tree$ ▷ the memorized experience

procedure SARSA-UCT-ITERATION(S_0)
  $episode \leftarrow$ GENERATEEPISODE(S_0)
  EXPANDTREE($episode$)
  BACKUPTDERRORS($episode$)

procedure GENERATEEPISODE(S_0)
  $episode \leftarrow$ empty list
  $s \leftarrow S_0$ ▷ initial state
  while s is not terminal
    if s is in $tree$ ▷ selection phase
      $a \leftarrow$ UCB1TREEPOLICY(s)
    else ▷ playout phase
      $a \leftarrow$ RANDOMPLAYOUTPOLICY(s)
    $(s, R) \leftarrow$SIMULATETRANSITION(s, a)
    Append (s, R) to $episode$
  return $episode$

procedure EXPANDTREE($episode$)
  $S_{\text{new}} \leftarrow$ first s in $episode$ that is not in $tree$
  Insert S_{new} in $tree$
  $tree(S_{\text{new}}).V \leftarrow V_{\text{init}}(S_{\text{new}})$ ▷ initial state value
  $tree(S_{\text{new}}).n \leftarrow 0$ ▷ counter of updates

procedure BACKUPTDERRORS($episode$)
  $\delta_{\text{sum}} \leftarrow 0$ ▷ cumulative decayed TD error
  $V_{\text{next}} \leftarrow 0$
  for $i =$ LENGTH($episode$) **down to** 1
    $(s, R) \leftarrow episode(i)$
    if s is in $tree$
      $V_{\text{current}} \leftarrow tree(s).V$
    else ▷ assumed playout value
      $V_{\text{current}} \leftarrow V_{\text{playout}}(s)$
    $\delta \leftarrow R + \gamma V_{\text{next}} - V_{\text{current}}$ ▷ single TD error
    $\delta_{\text{sum}} \leftarrow \lambda\gamma\delta_{\text{sum}} + \delta$ ▷ decay and accumulate
    if s is in $tree$ ▷ update value
      $tree(s).n \leftarrow tree(s).n + 1$
      $\alpha \leftarrow \frac{1}{tree(s).n}$ ▷ example of MC-like step-size
      $tree(s).V \leftarrow tree(s).V + \alpha\delta_{sum}$
    $V_{\text{next}} \leftarrow V_{\text{current}}$ ▷ remember the value from before the update

procedure UCB1TREEPOLICY(s)
  for each a_i in s
    $S_i \leftarrow$ afterstate where a_i leads to from s
    if S_i is in $tree$
      $V_{\text{norm}} \leftarrow$ NORMALIZE$(tree(S_i).V)$ ▷ normalization to $[0, 1]$
      $Q_{\text{UCB}}(s, a_i) \leftarrow V_{\text{norm}} + C_p\sqrt{\frac{2\ln(tree(s).n)}{tree(S_i).n}}$
    else
      $Q_{\text{UCB}}(s, a_i) \leftarrow \infty$
  return $\text{argmax}_a\left(Q_{\text{UCB}}(s, a)\right)$

Figure 5. Sarsa-UCT algorithm.

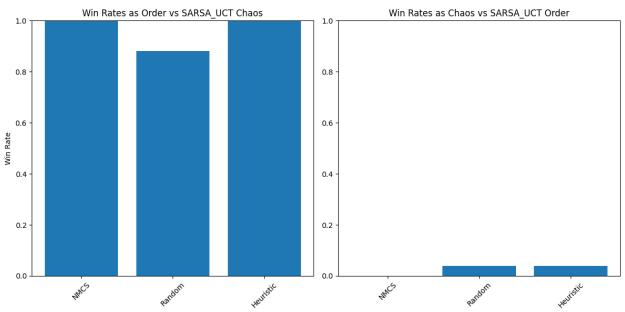


Figure 6. Sarsa-UCT algorithm performance