Monte Carlo Tree Search and Reinforcement Learning for a Four Player, Simultaneous Move Game

Finn Archinuk
University of Victoria
finn.archinuk@gmail.com

Dana Bell
University of Victoria
danabell841@gmail.com

Leo McKee-Reid
University of Victoria
leo.tzu.mr@gmail.com

Eric Showers
University of Victoria
ejshowers@gmail.com

Nathan Woloshyn
University of Victoria
nathanwoloshyn@gmail.com

***Abstract*—This work introduces a variant of Monte Carlo Tree Search (MCTS) for the game *BattleSnake*, a 4 player adversarial grid-based game with simultaneous turns. The solution we propose solves the problem of node selection in MCTS by having each player select a subset of nodes to explore instead of deterministically selecting a node, which is how traditional MCTS works. Our implementation allows an MCTS-based Reinforcement Learning model to iteratively improve against previous generations and benchmark models.**

I. Introduction

The study of game-playing AI is as old as computer science itself. Turing, Shannon and Von Neumann were all fascinated by the prospect of algorithms which could play games such as chess at a human (or superhuman) level. For many years, chess was a grand challenge for AI researchers, which culminated in the historic victory of Deep Blue over Gary Kasparov. But this victory can be attributed to sophisticated search algorithms and advanced hardware, not machine learning. Systems like Deep Blue are highly tuned to their domain, and rely heavily on the domain expertise of their programmers.

The ambition of much of modern AI research is to instead create general, blank slate systems that can learn a strong policy from first principles. Recently there have been impressive results from the application of Reinforcement Learning (RL) from self-play using tree search as a guide [Sil+17b]. Games such as Chess, Go, and Shogi ([Sil+17a] have superhuman quality agents. An important feature is that these games are between two opponents making sequential moves, so the traversal into new states is clearly defined. Similarly, for single player games the agent acts at each layer of the tree instead of alternating with the opponent.

Developing algorithms to solve these single player tasks can lead to real-world applications; a high profile example is AlphaTensor, where the researchers found an agent that generates a series of operations to more efficiently multiply matrices [Faw+22].

In this paper we develop a variant of tree search to train a Reinforcement Learning (RL) agent in the popular programming competition *BattleSnake*. We implement this algorithm and demonstrate the iterative improvements of the learned policy.

A. BattleSnake

BattleSnake is a turn based, simultaneous action, strategy game/programming competition inspired by the classic *Snake* arcade game. The *BattleSnake* environment is designed as a starting point for programming projects. Each of four players controls a "snake" with code that selects their move in one of four cardinal directions. The 11x11 board state is updated synchronously. During play a snake can consume food to extend its body length by one and refill a slowly decreasing health bar. A snake dies when it collides with a wall, has not eaten for 100 turns, or collides with a body segment of itself or another snake. If two or more snakes have a head-to-head collision, the largest snake survives.

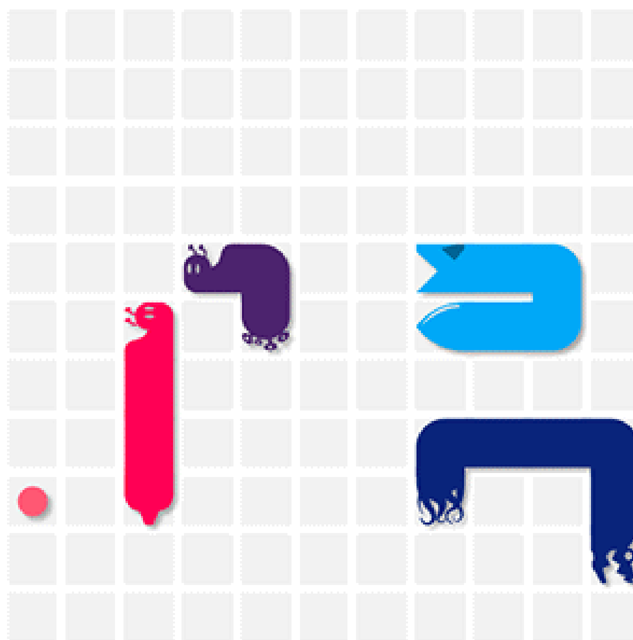

Fig. 1. A frame of BattleSnake with four agents on an 11x11 board.

In an official competition setting a player receives a data structure containing the current board state from the server hosting the match and has 500ms to respond with their selected action (failing to respond in time results in your snake

repeating it's previous move). The time limit constrains the amount of look ahead that can be done using tree search algorithms, such as minimax. Previous algorithms that have proven to be successful are flood-fill (an algorithm that finds connected positions) and A* (an algorithm that efficiently moves to a target).

The theoretical complexity of *BattleSnake* is high in terms of branching factor, which is the number of possible state transitions from any given state. However, this can be significantly pruned in practice. If all agents are alive, each one has four possible moves. Additionally, one item of food may or may not spawn randomly on any empty tile each turn. This gives us about $4^4 * (121 + 1) = 31232$ potential transitions. But we can ignore most of them by assuming that snakes will avoid moves that make them lose instantly (such as hitting their own neck or a wall), and that food spawns are irrelevant for search purposes. This reduces the search space to a much more manageable $3^4 = 81$ at most, and often even less (for example, when some agents are dead or have only one viable move). It is worth noting however that even this curtailed space is often still significantly higher than that of chess (≈ 35), and that the time control is much stricter than is usually the case in chess.

These constraints make this an interesting problem for RL; multiple agents, synchronous moves, and a high branching factor make exhaustive tree search computationally impossible. The complexity is further increased by the stochastic addition of food at random locations at random intervals, and some tree search methods are unable to handle randomness.

B. Problem Definition

The implementation of MCTS from *AlphaZero* is not immediately capable of handling *BattleSnake* due to multiple agents jointly defining a state transition. Variants of MCTS for multiple agents have been proposed before, and will be further outlined in section I-C. Our proposed solution recognizes that the quality of a move of a snake will depend on the moves of other snakes, and that each snake selects a reduced set of states that benefits them the most. During a real game (i.e., not for training), once each snake has made a selection, the intersection of the sets of states is reduced to one, which is the transition to the new state. However, when exploring the game tree with MCTS for training purposes, choosing a set of moves to transition to the next game state is far more complicated.

Creating an expert agent in *BattleSnake* is an important problem to solve, as many problems in the physical world can be thought of as multi-agent, simultaneous action games (either zero sum like *BattleSnake*, or with elements of co-operation), such as traffic on a road or mobile robots in a warehouse.

The goal of MCTS-based RL is to train a neural network (NN) to approximate the resulting MCTS tree. This paper outlines and implements a variation of MCTS that is applicable for multiple agents with simultaneous moves that is appropriate for *BattleSnake*.

C. Related Works

With the recent renaissance in deep learning, there have been many new ways of applying RL to problems such as computer Go, which have branching factors that limit the effectiveness of traditional tree search algorithms, such as minimax. Deep Blue's defeat of Kasparov in 1997 showed that sufficient amounts of compute applied in clever ways can surpass the strategic planning abilities of the strongest human players, but it used inflexible domain specific systems. *AlphaZero*, however, uses a general framework, using a deep convolutional network to approximate the move probabilities of a Monte Carlo search at each position.

Tak et al [TLW14] proposed Simultaneous Move Monte Carlo Tree Search (SM-MCTS) for a variety of two player games. They evaluate multiple formulations of the problem including as *Decoupled Upper Confidence Bounds for Trees*, *Exp3*, *Regret Matching*, and *Sequential Upper Confidence Bounds for Trees*. Their goals differ from what we are proposing in that we intend to approximate the resulting tree with a NN for iterative improvements.

Lanctot et al [Lan+13] applied MCTS to *Tron*, a game that resembles *BattleSnake* in some aspects. In this game, players move simultaneously and win by surviving longer than their single opponent while avoiding the walls they leave behind on the board. They used two implementations of MCTS: one that models the game sequentially during tree traversal but plays Monte Carlo simulations simultaneously, and another that uses SM-MCTS. They experimented with different algorithms for growing the tree for SM-MCTS, such as DUCT, Exp3 and Regret Matching. Decoupled UCT (DUCT) selects moves sequentially but hides the other agent's choice until both agents have chosen, simulating simultaneous play. Exp3 samples moves from a probability distribution based on their expected reward. Regret Matching generates a policy by comparing the regret values of moves. Our work is most similar to Exp3, but we differ in using a learned policy (for both tree construction and simulations) with a neural network.

There was a previous effort by a UVic team to apply the *AlphaZero* architecture to *BattleSnake*, but they eschewed the use of MCTS in favor of a pure value-based RL approach, citing issues with mustering the necessary compute for an MCTS based training pipeline [Sid+20]. In the domain of *BattleSnake*, current SOTA methods are heavily reliant on alpha-beta search methods with clever heuristics to maximize the amount of look-ahead that can be done in the 500ms response window.

II. Methodology

A. Coordinate System

The data structure provided at each turn comes in the form of a dictionary. We have represented the information in the dictionary as a series of layers, with 3 layers per snake (head location, body segment locations, and health) and a food layer. This results in an (11,11,13) array for a 4-player game on the standard (11,11) game board.

The number of board states can be reduced by a factor of 4 by converting this array into a relative coordinate system. This is inspired by [Sid+20] and is done by padding the board state to a $2n-1$ square grid (where n is the width of the board), translating the head of the selected snake to the center, and rotating so that the snake faces up. Padding the board increases the dimensionality of the board, but the complexity of the task is reduced since "nearby" obstacles are always towards the center of the board. Rotating the board reduces a snake's action space from the 4 cardinal directions into 3 relative directions (L=left, F=forward, R=right) since a downwards move will always collide with it's neck. To ensure the snake doesn't run off the board, we also add a channel indicating which locations are walls, so the final board dimensionality is (21,21,14).

B. Traditional MCTS

Monte Carlo Tree Search was introduced in [Abr86] to solve the problem of creating trees for an intractable problem. The challenge with other tree methods are that the number of nodes of the tree may expand beyond what memory can handle. The MCTS method iteratively builds a tree using simulations from leaves to estimate the quality of the state. While the MCTS method does not create a "true" tree, it will converge to the optimal tree as more samples are taken. Furthermore, by selecting the promising moves instead of an exhaustive search of the tree, the algorithm indirectly incorporates pruning.

Algorithm 1 Traditional MCTS

```
while within time limit do
    currentNode ← rootNode
    while currentNode ∈ searchTree do
        lastNode ← currentNode
        currentNode ← SELECT(currentNode)
    end while
    lastNode ← EXPAND(lastNode)
    Reward ← SIMULATION(lastNode)
    while currentNode ∈ searchTree do
        currentNode.BACKPROPAGATE(Reward)
        currenNode.visitCount ← currentNode.visitCount+1
        currentNode ← currentNode.parent
    end while
end while
```

MCTS can be broken down into 4 phases: selection, expansion, simulation, and backpropagation. Given the current tree, a node is *selected* that balances exploitation of previous rewards and exploration of interesting states. Selection is continued until the algorithm finds a leaf. The leaf is *expanded* to include all possible children. One of those children has a *simulation* run to approximate the quality of that state. Finally, the quality of the state is sent back up the tree through *backpropagation*. The MCTS algorithm is more thoroughly outlined in [Bro+12] and [Świ+22].

The primary challenge of implementing MCTS for *BattleSnake* is the *selection* phase, which we discuss in greater detail in the following section. The specific challenge is in selecting a node that greedily satisfies all 4 agents while also exploring the game tree for very good moves which are difficult to find.

C. Multiagent Simultaneous MCTS

With Multi-Agent MCTS [ZY19] and Simultaneous Move MCTS [Lan+13] having both been developed, we propose a combination of the two in order to effectively learn simultaneous multi agent environments: Multi-Agent Simultaneous MCTS (MAS-MCTS). The algorithm supports any number of agents simultaneously selecting moves.

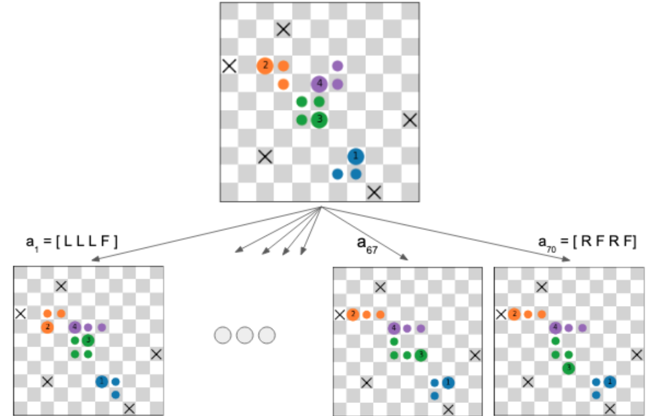

Fig. 2. Transitions from a parent to a child are in the form of a 4 element action.

In the case of *BattleSnake*, we support 4 agents selecting one of up to 3 moves each turn, which collectively define an action and state transition. Therefore our maximum non-trivial branching factor from any state is: $\#\ moves^{\#\ agents} = 3^4 = 81$. Agents select a move by considering what state transitions could possibly result from that move, which is the set of all possible actions where that agent chose that move. For example, for an agent to consider moving left they would consider the set of state transitions where they move left, and every other agent can make any of their valid moves. In the simple case of two agents, this would be the set $\{(L,L),(L,F),(L,R)\}$, where our agent moves left and the other agent may move left, forward or right. Figure 2 shows possible state transitions for *BattleSnake*. The arrows indicate state transitions as a combination of moves of the snakes. If a snake chooses to move left, there are 27 possible resulting states due to the moves of the opponents.

The selection of which child node to visit is collectively decided by the agents using a combination of *exploitation*—how likely an agent is to win from that node—and *exploration*—a measure of how novel the node is. The exploitation component is defined in equation 1 as the total reward collected from travelling to that node (W_a) divided by the number of visits to that node (N_a), repeated for each agent. This results in a reward summarization of the set of moves for each snake. The reward summarization is mapped to the 81 child nodes by

iterating over all permutations. This value is then normalized to sum to one. Note that in the following equations, a is an action, where an action is defined as a combination of all the agents moves. Actions therefore represent the edge from a node to a child node in the game tree.

$$Q_a = \frac{W_a}{N_a} \tag{1}$$

The exploration component, U, is defined in equation 2. The exploration coefficient (c) multiplies the NN probability of selecting a given action, $P(s, a)$, which is the product of the probability that the NN assigns to that move for each snake (equation 3). We therefore call the NN, $f_\theta(s)$, at every state when calculating the exploration component. For example, given the NN probability of each move for each snake at state s in Table I, if some action a = [L, L, R, F], then $P(s, a)$ = $m_{1,1}$ * $m_{2,1}$ * $m_{3,3}$ * $m_{4,2}$

TABLE I
NN OUTPUTS USED FOR CALCULATING $P(s, a)$

Alive Snakes	Move Left	Move Forward	Move Right
$f_\theta(s)$ for Snake1	$m_{1,1}$	$m_{1,2}$	$m_{1,3}$
$f_\theta(s)$ for Snake2	$m_{2,1}$	$m_{2,2}$	$m_{2,3}$
$f_\theta(s)$ for Snake3	$m_{3,1}$	$m_{3,2}$	$m_{3,3}$
$f_\theta(s)$ for Snake4	$m_{4,1}$	$m_{4,2}$	$m_{4,3}$

In this table, $m_{i,j}$ is the probability that the NN assigns to the ith snake to move in the jth direction

The remainder of equation 2 biases exploration of under-visited nodes by dividing the number of visits of the children ($\sum_b^{81} N(s, b)$) by the number of all visits from that state ($1 + N(s, a)$).

$$U_a = cP(s, a)\frac{\sqrt{\sum_b^{81} N(s, b)}}{1 + N(s, a)} \tag{2}$$

$$P(s, a) = \prod_i Pr(f_\theta(s) = a_i) \tag{3}$$
$$\forall i \text{ such that } snake_i \text{ is currently alive}$$

The selection of the node to visit is done by finding the maximum Action Value $V_A = Q + U$. As with traditional MCTS, if the selected node has child nodes, this process is repeated. If the selected node does not yet have children, the node is expanded, and the Action Value of the child nodes are calculated.

At the end of each MCTS loop for a given state, the selected action to advance one turn in the real game is calculated by taking a probability distribution of π 4. π is a 4x3 array, where each element is proportional to how often a given snake made a certain move during MCTS. This selection method makes sense, since the number of visits to a node during MCTS, N_a, is determined by both the exploitation and exploration components.

$$\pi_{i,j} = \sum_{a:a_i=j} \frac{N_a}{\sum_b N_b} \tag{4}$$

The following algorithm 2 describes the repeated process of generating data and training the NN via self-play.

Algorithm 2 MCTS Training Loop

```
Generate initial set of training games using trivial policy
while NN improving do
    while Games played < 200 do                ▷ Generate data
        Initialize new game with 4 identical NN agents
        while Game is not over do
            Use MCTS and the NN to calculate the best next
              move probability for all agents, π
            Save current game state and associated π
            Move all agents 1 turn as a probability
              distribution of π
        end while
    end while
    Train new NN on generated games
    Ensure new NN out-performs previous NN
    NN ← new NN
end while
```

III. Results

A. Initial Training Data

To generate the initial training data, 200 games were played using MCTS simulations at each state to calculate how each snake would move. For the first set of games, we use a trivial agent that was classically coded to avoid immediately moving into obstacles, but had no other planning routine. For this initial set of games, each turn was simulated 2000 times with a maximum depth of 10 steps or until only one snake remains, whichever comes first. The intention here is to provide high quality training data for the first NN. These games resulted in approximately 270,000 samples for training.

B. Training

The NN is a simple convolutional neural network (CNN) with two convolutional layers (16 filters, 3x3 kernel, ReLU activation), followed by two fully connected layers with 256 neurons each and ReLU activation. There are then 3 output neurons with a softmax activation. This model was trained for 100 epochs with a batch size of 1024. The optimizer was AdamW [LH17] with a learning rate of 5e-5 and weight decay of 5e-5. The output of the NN is a 3 element probability distribution determined by the MCTS summary, so we selected Kullback-Leibler Divergence (KLD) as the loss function.

C. Training Loop

Once the NN was trained on the initial games, it becomes an approximation of MCTS. The next set of games were trained using the fixed weights of the NN. Each turn of these games were simulated only 1000 times, since the planning of the NN could produce higher quality exploration than a random policy. This training loop is outlined in Algorithm 2.

Training the NN requires converting the board state into a relative coordinate frame and reducing the predicted loss using the MCTS generated tree summarization as the ground truth.

D. Architecture Modification

The most recent NN update includes an adjustment to the architecture and a preprocessing step to the inputs. The architecture has an additional fully connected layer of 256 neurons with a ReLU activation. The health of snakes were reduced by a factor of 100. Finally, the snake bodies have an inherent order to them (from neck down to tail) which the network can access. The previous training data had these body segments clipped to either zero or one, indicating either the absence or presence of a body obstacle, respectively. This alternative architecture and preprocessing was selected due to a significant decrease in validation loss observed during training.

E. Evaluation

Here we evaluate the quality of the learned policy with special attention paid to improvements between iterations. The evaluation metric we use is the percentage of wins against three opponents. Since *BattleSnake* has four agents, our learned policy is expected to win 25% of the time if it is performing at a similar level. This is an imperfect metric since snake behaviours may be non-transitive, with snakes winning more or less frequently depending on the behaviours of their similarly matched opponents. As a first approximation, and to demonstrate a trend in improvement, win percentage is sufficient.

At the time of writing we have three iterations of the NN policy. For each of the three iterations we are testing against two types of opponents: a Trivial snake that is only able to avoid immediately adjacent obstacles, and a Flood Fill snake. The Flood Fill snake works by moving to the best non-obstacle position according to a heuristic which calculates the best-case longest-path possible from each available position.

Each iteration plays 500 times against three of the opponents of the given type. Table II shows the learned policy is able to regularly win against 3 Trivial opponents. This table also shows that although it has a low win percentage against the more complex Flood Fill snake, the win percentage increases with each training iteration.

TABLE II
WIN PERCENTAGE VS 3 OPPONENTS

Policy	vs Trivial	vs Flood
Iter. 1	53.4%	2.8%
Iter. 2	62.4%	4.0%
Iter. 3	76.6%	6.8%

To ensure the snakes are improving relative to their previous generation, we also test all three iterations together, with the fourth spot filled with either the Trivial Snake or the Flood Filling snake. The results of this test are summarized in Table III. These values may not sum to 100% due to draws. Again, 500 games were played to limit the influence of random chance.

TABLE III
WIN PERCENTAGE VS VARIED OPPONENT

Game	Policy	Wins
1	Trivial	4.4%
1	Iter. 1	15.8%
1	Iter. 2	23.2%
1	Iter. 3	54.2%
2	Flood	66.6%
2	Iter. 1	5.2%
2	Iter. 2	8.6%
2	Iter. 3	18.8%

These results indicate that not only is each training iteration improving the model, but the improvements from Iter. 2 to Iter. 3 are greater than from Iter. 1 to Iter. 2. Based on these initial results, it is reasonable to anticipate that the model will continue to improve with additional training.

IV. CONCLUSION

In this work we have outlined a variant of Monte Carlo Tree Search for multiple adversarial agents in a sychronous environment. We have shown this algorithm can be used for training a neural network as introduced by AlphaZero [Sil+17b] in an iterative process, and that later iterations of the model outperform prior iterations.

V. FUTURE WORK

While the MCTS-based RL method we have developed is generating good initial results, this project will undergo a number of changes in the coming weeks to improve performance. In addition to experimenting with different NN architectures, hyperparameter tuning, and increasing our depth of MCTS simulation, we will also be updating our training pipeline, as described in the following algorithm.

Algorithm 3 Improved MCTS Training Loop

Generate initial set of training games using trivial policy
while NN improving **do**
  while Games played $<$ 500 **do** $\triangleright$ Generate data
    Initialize new game
    while Game is not over **do**
      Use MCTS and the NN to calculate the best next move probability for all agents, π
      Save current game state and associated π
      Move all snakes 1 turn as a probability distribution of π
    end while
  end while
  Train 3 new NNs on generated games
  Play 100 4-player evaluation games between the 3 new NNs and the previous NN
  NN $\leftarrow$ the NN which won the most test games
end while

Aside from increasing the number of generated training games for each model iteration, the difference between this algorithm and the currently implemented pipeline (Algorithm

2) is the idea of training multiple NNs each iteration, then selecting the best by running evaluation games. Generating games is the most computationally expensive step in this pipeline, so better utilizing the generated games may decrease the number of training iterations needed to achieve an equivalent skill level.

Finally, we will experiment with different rewards during MCTS simulations. Our current model is only rewarded for survival (with less reward if opponents are also alive). Giving explicit reward for eliminating opponents, finding food, or ending games quickly may significantly affect training. Since *BattleSnake* is an incredibly complex and stochastic game, there may be different playing styles that correlated with different peaks of optimal play, and reward engineering may be the key to unlocking certain styles of play.

References

[Abr86] Bruce Abramson. “Thesis Proposal: The Expected-Outcome Model of Two-Player Games”. In: 1986.

[Bro+12] Cameron Browne et al. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012), pp. 1–43.

[Lan+13] Marc Lanctot et al. “Monte Carlo Tree Search for Simultaneous Move Games: A Case Study in the Game of Tron”. In: 2013.

[TLW14] Mandy J. W. Tak, Marc Lanctot, and Mark H. M. Winands. “Monte Carlo Tree Search variants for simultaneous move games”. In: *2014 IEEE Conference on Computational Intelligence and Games* (2014), pp. 1–8.

[LH17] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2017. DOI: 10.48550/ARXIV.1711.05101. URL: https://arxiv.org/abs/1711.05101.

[Sil+17a] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. DOI: 10.48550/ARXIV.1712.01815. URL: https://arxiv.org/abs/1712.01815.

[Sil+17b] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (2017), pp. 354–359.

[ZY19] Nicholas Zerbel and Logan Michael Yliniemi. “Multiagent Monte Carlo Tree Search”. In: *Adaptive Agents and Multi-Agent Systems*. 2019.

[Sid+20] Ahmed Siddiqui et al. “Multiagent Reinforcement Learning in a Synchronous Strategy Game”. In: (2020). URL: https://github.com/Fool-Yang/AlphaSnake-Zero/blob/master/report.pdf.

[Faw+22] Alhussein Fawzi et al. “Discovering faster matrix multiplication algorithms with reinforcement learning”. In: *Nature* 610 (2022), pp. 47–53.

[Świ+22] Maciej Świechowski et al. “Monte Carlo Tree Search: a review of recent modifications and applications”. In: *Artificial Intelligence Review* 56.3 (2022), pp. 2497–2562. DOI: 10.1007/s10462-022-10228-y.