mathematics

MDPI

Article

Pruning Stochastic Game Trees Using Neural Networks for Reduced Action Space Approximation

Tasos Papagiannis *, Georgios Alexandridis and Andreas Stafylopatis

Zografou Campus, School of Electrical & Computer Engineering, National Technical University of Athens, 15780 Athens, Greece; gealexandri@islab.ntua.gr (G.A.); andreas@cs.ntua.gr (A.S.)
* Correspondence: tasos@islab.ntua.gr

Abstract: Monte Carlo Tree Search has proved to be very efficient in the broad domain of Game AI, though it suffers from high dimensionality in cases of large branching factors. Several pruning techniques have been proposed to tackle this problem, most of which require explicit domain knowledge. In this study, an approach using neural networks to determine the number of actions to be pruned, depending on the iterations run and the total number of possible actions, is proposed. Multi-armed bandit simulations with the UCB1 formula are employed to generate suitable datasets for the networks' training and a specifically designed process is followed to select the best combination of the number of iterations and actions for pruning. Two pruning Monte Carlo Tree Search variants are investigated, based on different actions' expected rewards' distributions, and they are evaluated in the collectible card game Hearthstone. The proposed technique improves the performance of the Monte Carlo Tree Search algorithm in different setups of computational limitations regarding the available number of tree search iterations and is significantly boosted when combined with supervised learning trained-state value predicting models.

Keywords: Monte Carlo Tree Search; pruning; neural networks; multi-armed bandit; Upper Confidence Bound; Hearthstone

MSC: 68T20

Citation: Papagiannis, T.; Alexandridis, G.; Stafylopatis, A. Pruning Stochastic Game Trees Using Neural Networks for Reduced Action Space Approximation. *Mathematics* **2022**, *10*, 1509. https://doi.org/10.3390/math10091509

Academic Editors: Theodore Andronikos and Georgios I. Goumas

Received: 31 March 2022
Accepted: 28 April 2022
Published: 1 May 2022

1. Introduction

Over the past years, Monte Carlo Tree Search (MCTS) has been the go-to approach concerning Game AI research, achieving exceptional performance in deterministic board games and video games, as well as stochastic games (e.g., poker) [1,2]. Several methodologies have been integrated to the original version of the algorithm in order to enhance its different phases (mainly selection and rollout), including reinforcement learning, supervised learning, pruning techniques, and statistical approaches, etc. Growing research towards this direction established the algorithm as the state of the art and eventually led to a superhuman performance in many games [3,4].

Despite its great efficiency, MCTS is budget-limited; it is only allowed to run for a predefined number of iterations or a certain amount of time. Although this is not an issue concerning its functionality (since the algorithm can stop at any time and return the current estimation on the actions' expected value), it certainly affects its performance, especially in games with large action spaces. While the number of available actions grows, the tree search is based on an insufficient number of simulations per action, often leading to incorrect evaluation. As such, the agent's decisions are highly dependent on the total number of iterations executed per move.

To address this problem, a pruning technique involving neural networks is introduced in this work. Two networks are combined to firstly define the subset of actions that could be pruned safely at each timestep of the algorithm and then determine the optimal

number of iterations to execute before pruning, along with the actions to prune. For the networks' training, a synthetic dataset has been generated through a specifically implemented environment simulating the multi-armed bandit problem [5]. The distributions of the actions' expected values were also examined and taken under consideration for the dataset's creation, resulting in two different variants of the proposed algorithm. Its effect is evaluated on both the vanilla MCTS version and an already enhanced MCTS-based agent, in the collectible card game Hearthstone [6]. The experiments highlight the benefits of the proposed technique, as the pruning networks integrated agents outperform the non-pruning ones in both cases.

The aforementioned methodology's contribution is twofold. Firstly, the suggested algorithm introduces a machine learning-based approach for reducing the action space during the MCTS selection phase, as opposed to statistical methodologies and handcrafted pruning. For this purpose, a simulation process is also developed in order to generate appropriate data for the proposed models' training. Secondly, this approach is domain independent as the pruning networks are trained on simulated data rather than data obtained from a specific game. As a result, there are no constraints concerning the environment, thus the algorithm is generally applicable to any task suitable for the vanilla MCTS without need for additional modifications.

The rest of the paper is structured as follows: In Section 2, related approaches concerning pruning methodologies in MCTS are presented. Section 3 explains the Upper Confidence Bound formula for the multi-armed bandit problem, and Section 4 briefly describes the game and the framework used for carrying out the experiments. The functionality of the MCTS algorithm is illustrated in Section 5, followed by the detailed description of the proposed technique (Section 6). Section 7 analyzes the effect of the aforementioned methodology to the agent's performance, while Section 8 concludes and discusses potential future work.

2. Related Work

Several pruning approaches have been proposed for the MCTS algorithm, both domain dependent and domain independent. In Ref. [7], the authors introduced absolute and relative pruning conditions for the Upper Confidence Bounds applied to Trees (UCT) algorithm [8]. In the former case, actions are pruned when it becomes impossible to be the most visited ones, depending on the remaining number of iterations, whereas in the latter case, an upper bound is calculated for the expected number of visits of each action and actions which can not reach the current highest number of visits are pruned. Both of these methods fall into the category of hard pruning, as the actions are no longer examined in the tree search once they are pruned.

A more flexible approach, which allows pruned moves to be reconsidered after a specific amount of time or number of iterations, called soft pruning, has also been investigated. This technique ensures that actions do not get permanently eliminated and decreases the risk of completely excluding the optimal action from consideration. Under this scope, progressive unpruning [9] and progressive widening [10] make use of domain-specific heuristics to evaluate the nodes and prune most of them after a threshold timestep is reached. Gradually the excluded actions get unpruned and become available as the MCTS iterations increase. These methods have been tested on the Go board game and improved MCTS, highlighting the impact of the branching factor in the algorithm's performance. In Ref. [11], this approach is further developed and adapted to continuous stochastic environments. In this case, different states produced by the same action-state pair can be added to the action space, depending on the number of children nodes and iterations. Although this modification improves the single progressive widening in specifically designed evaluation environments, the two methods exhibit similar performance on real world problems.

Domain knowledge has been employed for hard pruning of actions, as well. In Ref. [12], handcrafted heuristic functions have been considered to decrease the action space

in the strategic card game Lords of War, achieving superior performance to the vanilla MCTS agent. In Ref. [13], nodes of obviously detrimental actions have been hard pruned in a real-time strategy game, resulting in a significant performance increase. In Ref. [14], several existing pruning techniques have been explored on a turn-based strategy game and validated the improvement of the MCTS agent as the branching factor is reduced.

Apart from the selection phase of MCTS, pruning has also been used to eliminate nodes in the playout stage of the algorithm. In Ref. [15], rapid action value estimate (RAVE) statistics [16] are exploited to prune actions with a win rate lower than a certain threshold in the playout phase. In this way, the simulations are focused on the most promising actions, leading to more accurate estimations of the nodes' expected values. The authors tested this technique for the game of Havannah and outperformed several improved versions of the algorithm.

3. Multi-Armed Bandit and Upper Confidence Bounds

The multi-armed bandit (MAB) problem (also known as K-armed bandit) is a decision-making problem based on the exploration–exploitation dilemma [5]. Namely, it describes the situation of a gambler trying to maximize their profit by iteratively choosing among several slot machines (one-armed bandits) with unknown reward distributions. Thus, the player should make decisions in a way that exploits information acquired from previous rewards, but it also investigates the more rarely selected options, in order to verify the highest rewarding bandit.

Formally, the MAB can be defined as a set of K-random variables $R = \{R_1, R_2, \ldots, R_k\}$ associated with real distributions $D = \{D_1, D_2, \ldots, D_k\}$, where each variable represents the reward of an action $x_i \in X = \{x_1, x_2, \ldots, x_k\}$ and each distribution represents the respective reward distribution. Considering a finite number of turns T and a selection policy $\pi(t)$, let x_i be the action selected at timestep t and $r_i \sim D_i$ the gained reward. The player's goal is to minimize the regret (Equation (1))

$$\rho = T \max_{x_i \in X} \mathbf{E}[R_i|x_i] - \sum_{t=1}^{T} \sum_{i=1}^{K} r_i [\![x_i = \pi(t)]\!] \tag{1}$$

that is, the difference between the total reward gained by following policy π and the total reward achieved by always selecting the optimal action (i.e., the action with the highest expected reward). In this setting, the MAB problem can be reduced to a Markov decision process (MDP) with a state transition function $P(s, s'|a) = 0 \; \forall \; s' \in \{S - s\}$, where S is the set of possible states, since taking an action does not lead to a state change. In this respect, the expected reward can be viewed as the action's value $Q(x_i)$, with $Q(x_i^*) = \max_{x_i \in X} Q(x_i)$ being the value of the optimal action.

Several strategies have been proposed to solve the multi-armed bandit problem. The Upper Confidence Bound (UCB) algorithm focuses on optimizing the selection strategy by balancing the exploration and exploitation of current information [17]. It is based on the optimistic assumption that the true value of an action is higher than its current estimation. Particularly, an upper bound is calculated for each action's value depending on its current approximation and the respective degree of uncertainty. According to the Hoeffding's inequality [18], let $X_1, X_2, \ldots, X_n$ be i.i.d. random variables in the $[0, 1]$ interval and $\overline{X} = \frac{1}{n}(X_1 + X_2 + \cdots + X_n)$, then, the probability of the difference between the mean of the random variables and its expected value being higher than a threshold k is bounded according to Equation (2):

$$P(\mathbf{E}[X] - \overline{X} \geq k) \leq e^{-2nk^2} \Rightarrow P(\mathbf{E}[X] \geq \overline{X} + k) \leq e^{-2nk^2} \tag{2}$$

In the case of the bandit problem, by replacing random variable X with action's reward R, we obtain the probability of the action's true value being higher than the upper bound (Equation (3))

$$P(\mathbf{E}[R] \geq \overline{R} + k) \leq e^{-2nk^2} \tag{3}$$

As UCB selection is made optimistically, the upper bound must be as strict as possible, i.e., it should be greater or equal to the expected value with high probability. Hence, the probability of Equation (3) should be very small. By setting this probability equal to a very small positive value a, threshold k can be determined as in Equation (4). As the number of samples grows, the confidence on the estimated value increases. Therefore, the upper bound could be decreased proportionally to the total number of current iterations. In UCB1, the most commonly used variation of the algorithm, a is set to N^{-4}.

$$e^{-2nk^2} = a \Rightarrow k = \sqrt{\frac{-\ln a}{2n}} \xrightarrow{a=N^{-4}} k_{\text{UCB1}} = \sqrt{\frac{2 \ln N}{n}} \tag{4}$$

In general, at each timestep t, the UCB algorithm selects the action with the highest upper confidence bound, as in Equation (5)

$$\text{UCB}(x_i) = Q(x_i) + c\sqrt{\frac{\log N}{n_i}} \tag{5}$$

where $Q(x_i)$ is the current approximation of the value of x_i, N is the total number of selections made until timestep t, n_i is the number of times x_i has been selected, and c is the exploration parameter.

In Equation (5) above, the first term is related to exploitation (by taking into consideration the estimated value of each action), while the second term concerns exploration and indicates the uncertainty of the current estimation. The more times an action has been evaluated, the smaller should be the the increase of its upper bound, as the confidence on that value gets higher, and vice versa. Thus, the exploration term decreases as an action gets selected and consequently the upper bound of the action's value tends closer to its estimation, as the number of timesteps grows. Finally, c controls the weight of the exploration term and, in the case of UCB1, it is set to $\sqrt{2}$.

4. Hearthstone

Hearthstone [6] is an online, two-player, collectible card game (CCG). Each player selects a hero and drafts a deck of 30 cards (with at least 15 different ones), choosing among the hero class's cards and a set of neutral cards that can be included in any deck. The cards fall into three broad categories; minions, spells, and weapons. Minions have attack and health points, meaning they are able to attack other minions as well as the opponent's hero. They can also have special abilities, such as being able to attack immediately after being summoned (which is normally not possible), attack twice a turn, etc. Spell cards can affect the board in many ways, but they are most commonly used to damage opponents or to boost friendly minions' stats. Spells may target a specific entity (minion or hero) or cause a general effect in the game. Finally, weapons grant heroes attack points, making them able to attack a specific number of times in the same way as minions. The goal of the game is to eliminate the opponent hero's health points, either by attacking them (with minions or weapons), or by casting spells on them.

The game is turn-based and each card has a cost value (mana) in order to be played. Both players draw five cards from their decks at the start of the game (the one who plays second draws one more, plus a special bonus card in order to equalize their winning chances) and one more at the start of their turn. Each player has a specific amount of available mana to use on every turn, with no limitation in the number of actions per turn (as long as they are permitted by the rules of the game). Based on that, there is no time limit per action, though there is a total time budget for every turn, independent of the number of

executed actions. However, as the moves are selected sequentially, for the purposes of this research, each action has a predefined computational budget, as described in Section 7, in order to evaluate the performance of the different algorithms equally.

Concerning the game strategies, there are three main approaches that determine the building of the decks.

1. **Aggro**. In this approach players attempt to fill the board with minions as early as possible and finish the game before the opponent is able to defend. Thus, the decks consist mainly of low-cost cards, which can be played in the first turns (when the available amount of mana is small). This strategy, however, may lead to a great disadvantage if the opponent survives the first turns, as there are no cards suitable for late-game in an aggro deck;
2. **Control**. A slowly progressing strategy, aiming to prevent the opponent from early development, mainly with removal spells and then gaining control of the board with powerful, high-value minions in the late-game. Control decks entail the risk of being outplayed in the first stage of the game, as they rely on high-cost cards, which are not playable in the early turns, but are very dominant when the player endures the initial pressure;
3. **Midrange**. A more flexible type of deck, aiming to control the game in the early turns and win during the mid-game. It usually contains strong low-cost minions and spells in order to trade efficiently and gain advantage from the start of the game. Most cards are of medium cost, meaning this kind of deck struggles in the late game against decks with high-value minions.

In theory, each of the above deck archetypes has an advantage over one of the other two and a disadvantage against the other. Specifically, aggro decks are considered to have more winning chances against midrange in most cases, midrange against control, and control against aggro. As this relationship introduces bias to the evaluation of the game-playing agents, in the current work, three different decks have been selected (one for each archetype), and all experiments are carried out through games where both players use the same deck.

The proposed agents were implemented and evaluated in Metastone [19], an open-source Hearthstone simulator in Java. Metastone provides a simple environment designed to serve as a developing tool for deck building and AI testing in Hearthstone by simulating the game's rules and logic. The machine learning models described in Section 6 have been developed in the Python programming language and communicate with the agent through a REST interface. More implementation details are available on the method's code repository [20].

5. Monte Carlo Tree Search

MCTS is a search algorithm that solves decision problems by representing the data structures as a tree [21]. In the particular field of Game AI, actions are mapped to the game tree's edges and the possible states to the nodes. The goal of the algorithm is to select the best action (i.e., one of the root node's children), while the game tree may be expanded up to leaf nodes, depending on the available sources. Since the game tree does not need to be expanded symmetrically, a key concept of the algorithm is the early emphasis on the most promising moves, developing the tree structure accordingly. Game tree creation and action selection may be broken down to four main stages:

1. **Selection**. One of the possible actions (children of the current node) is selected according to a specific policy. This step is repeated until a non-fully explored node (i.e., a node where not all of its children have been visited at least once) is reached;
2. **Expansion**. When a non-fully explored node is encountered, an action that has not been visited previously is selected (usually at random) and the corresponding node is added to the tree;
3. **Rollout**. A full game is simulated from the newly expanded node following a simulation policy (in the vanilla version, random moves are executed) until a terminal state;

4. **Back-propagation.** The results of the simulation are back-propagated from the expanded node to the root, updating the statistics of all the traversed nodes.

The above steps are executed repeatedly, starting from the root node, for a fixed number of iterations or a certain amount of time. There are several approaches for determining the best action after the process is completed, with the most common being selecting the most visited action. Other approaches also include selecting the move with the highest score or an optimal combination of score and visits.

UCT is the most widely used variation of MCTS, employing the UCB formula described above to guide the selection phase, based on the current statistics of the tree nodes [8]. In this case, node selection is treated as a multi-armed bandit problem and the win rate of each action, calculated by the Monte Carlo simulations, serves as the reward. As already discussed, UCB initially boosts exploration and focuses on exploitation while the number of iterations increases, resulting in the asymmetric game tree.

6. Proposed MCTS Enhancements

6.1. Pruning Networks

A key concept of this work is to prune actions that seem to be suboptimal from the game tree as early as possible and reduce the action space. Specifically, the goal is, after a certain amount of iterations, to be able to focus the search on the most promising moves, without risking excluding the optimal one from the search space. In our approach, neural networks have been applied to the MCTS algorithm in order to determine the number of actions that should be pruned, as well as the number of iterations to run before pruning.

For this purpose, two different neural networks are employed. The first one (Safety Network) is used to predict the maximum number of actions that can be safely pruned (i.e., the largest set of low estimated-value actions that does not contain the optimal one), given the number of executed iterations and the total number of actions. As expected, the allowed remaining action space decreases while the number of iterations run gets higher, since the increase in performed simulations leads to a more precise evaluation of each action. Consequently, there is a trade-off between the actions' sets to be pruned and the remaining number of iterations, as the more time that is used until pruning, the less iterations are available for search among the remaining actions. In order to overcome this issue, a second network (Probability Network) is trained to predict the probability of selecting the optimal action, depending on the action space and the remaining iterations. Hence, in the first step, the minimum remaining action space is calculated for all possible numbers of iterations using the Safety Network and then the optimal pair of iterations to run and actions to prune is determined as the one with the highest probability of finding the best action, according to the Probability Network. Each time the selected number of iterations is reached, the determined subset of actions is pruned from the search tree and the process is repeated until the total iterations are carried out (Figure 1).

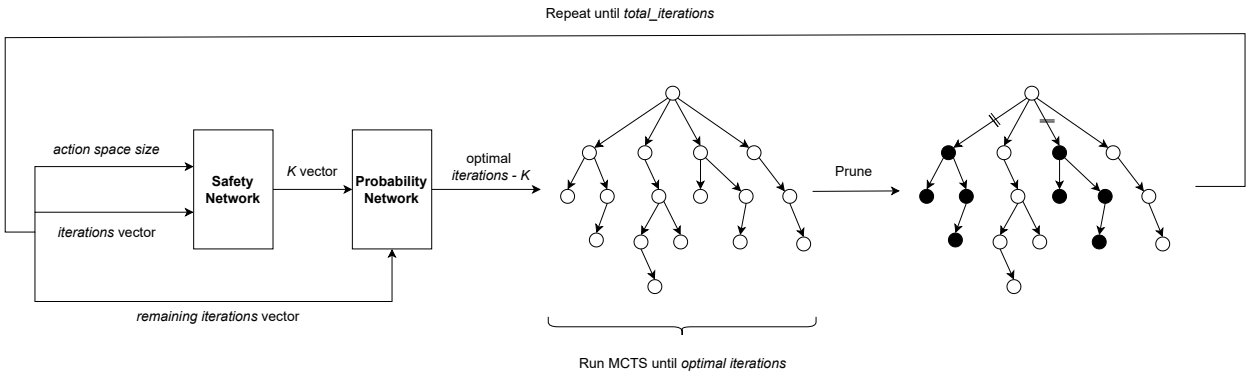

Figure 1. Proposed pruning process for MCTS (pruned nodes are marked as black).

The pruning procedure is described analytically in Algorithm 1. Initially, the minimum number of actions needed to continue the search process (denoted as K_i) is predicted by the Safety Network for each value i of available iterations, given the current action space size (lines 4–5). As in the start of the tree search, there is no information on the actions' values (they have not been evaluated yet), the Safety Network is applied after a threshold of iterations is reached. Since the action space size in the tested environment is smaller than 50 in the vast majority of moves, the threshold value has been set to 100, so that all actions' expected rewards are estimated to an extent at the time of prediction, resulting in $i \in [current\_iterations + 100, total\_iterations]$. Subsequently, the Probability Network is used to predict for each number of iterations the probability of the optimal action being selected by MCTS, considering pruning is performed at that point (lines 6–9). The optimal iterations–action space pair is defined according to the produced probabilities (lines 10–11) and MCTS is executed for the specified iterations or until the total iterations are completed (lines 12–16). When the selected $Iter_{opt}$ is reached, the lower value estimated actions are pruned and the tree search is resumed on the top K_i actions (lines 17–18). The algorithm is continued iteratively until the computational budget is consumed.

Algorithm 1: MCTS with Pruning Networks.

```
1  Function MCTS():
2      while total_iterations > 0 do
3          probs ← []
4          for i in iterations do
5              K_i ← Safety_Network.predict(action_space, i)
6          for i in iterations do
7              remaining_iterations ← total_iterations − i
8              prob_i ← Probability_Network.predict(K_i, remaining_iterations)
9              probs.append(prob_i)
10         K_opt ← argmax_K probs
11         Iter_opt ← argmax_iter probs
12         for i in min (Iter_opt, total_iterations) do
13             Select()
14             Expand()
15             Rollout()
16             BackPropagate()
17         {actions_to_prune} ← {total_actions} − {K_opt}
18         {action_space}.remove({actions_to_prune})
19         total_iterations ← total_iterations − Iter_opt
20     return best_action
```

Concerning the pruning networks' architectures, they are both feed-forward neural networks with three hidden layers. The hidden layers' sizes are 200, 300, and 100 for the Safety Network and 300, 500, and 200 for the Probability Network, while the activation function used is the *Rectified Linear Unit*. Both networks were trained using the Adam optimizer [22] for minimizing the mean squared error loss (the task of the Safety Network was treated as a regression problem as well, with the predicted value being rounded to the closest integer).

6.2. Datasets' Creation

An issue concerning the creation of the datasets for training the networks is that there are no labels (i.e., there is no knowledge of the optimal action, even after a game is finished), as the expected value of the actions is unknown. The evaluation of the agents' performances is based on their win rates, which are not informative of the individual actions' effects on

the outcome of the game. Hence, a MAB model has been used to create a suitable dataset for the specific task.

Specifically, a simulation environment has been designed to implement the UCB formula (which is essentially the selection phase of the MCTS) on the MAB problem. Different simulations have been performed for action space sizes in the $[3, 50]$ range in order to create a complete dataset. The range bounds are set accordingly, as there is no point in pruning in case there are less than three actions available, and the total number of actions per move in Hearthstone rarely exceeds 50. For each action, the reward gained at each timestep is sampled from a *Bernoulli* distribution. In this case, each action's expected reward is represented by the Bernoulli parameter p. This way, the optimal action is known a priori and can be used as ground-truth for the networks' training.

In the first dataset, each sample should contain the action space size and the number of iterations run (inputs to Safety Network), while the respective label should indicate the optimal (minimum) subset of actions containing the action with the highest expected reward. Several values of iterations were combined with each value of possible actions to form the required samples. The numbers of iterations were sampled from the $[100, 1500]$ range, since a lower bound is needed for the algorithm to adequately evaluate all actions (as explained in Section 6.1), while the total number of iterations is usually restricted (in the current work, a maximum of 1000 iterations is considered). Each combination of action space and number of iterations was simulated 1000 times and the UCB formula was used to select the best action. Finally, the minimum number of actions (as sorted by the UCB visits) that contained the best action (i.e., the action with the highest probability p) in every simulation was calculated for each combination. Following this process, the final dataset consists of 4704 tuples (98 per number of total actions) of iterations and total actions (serving as features) and the respective sizes of action space after pruning (labels).

Regarding the Probability Network's training, a dataset consisting of tuples of the remaining iterations and action space sizes (input), along with the respective probabilities of selecting the optimal action (labels), is required. Similarly to the process described above, different pairs of available iterations and actions were formed and the probability of selecting the best action was determined for each case over a set of 1000 simulations.

Initially, the Bernoulli parameter p was drawn from the uniform distribution, meaning that any value in the $[0, 1]$ range was considered equally probable to represent an action's expected reward. In order to investigate whether the knowledge of the specific distribution can lead to more suitable pruning networks, an estimation of the actions' expected reward distributions was calculated by samples obtained from MCTS self-playing games, specifically for Hearthstone (Figure 2). Afterwards, a second pair of datasets was generated using a bimodal distribution, matching the ones shown below to select the parameter p for each action. Then, the steps described above were repeated to train a pair of pruning networks on the new, bimodal distribution-based datasets. The performance of both approaches is presented in Section 7.

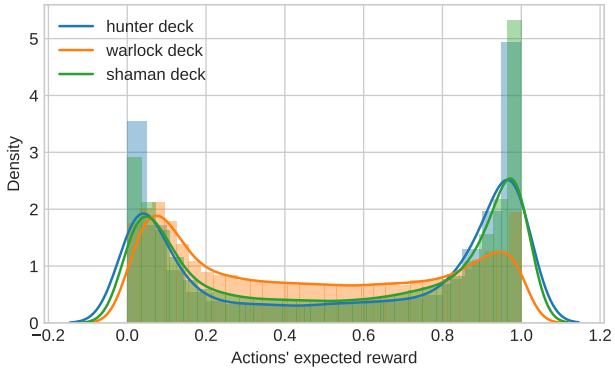

Figure 2. Actions' expected rewards estimated distributions in Hearthstone.

6.3. Supervised Learning

The concept of combining supervised learning with MCTS has been established as a promising variation of the algorithm since the dominant performance of the AlphaGo agent against human players [23]. In this approach, random rollouts are replaced by (or combined with) value networks, trained to predict the outcome of the game based on a game state. This technique tends to reduce the variance introduced by random simulations by utilizing domain knowledge, leading to more accurate estimations. In the specific area of CCG, and Hearthstone particularly, predictive models of the game states have also been employed in order to enhance the MCTS simulation phase [24].

As the current work is based on the authors' previous study [25], the model used for the states' evaluation is an XGBoost classifier [26] trained on a specifically designed dataset for this task. In particular, simple MCTS agents were employed in self-playing games, in order to create a dataset of game states (feature vectors) and winners (labels). A separate model has been trained for each deck archetype (using the respective subset of the data), as well as a general model trained on the data of all games. As the performance of the agents proved to be very similar, all the experiments presented in the remainder of this paper refer to agents enhanced with the general classifier.

Concerning the game state in Hearthstone, it is characterized by quite a large set of attributes. Furthermore, the length of the feature vector varies depending on several aspects, such as the number of cards in players' hands, the number of summoned minions on the board, etc. Therefore, a fixed subset of the features was selected in order to facilitate the model's training. Specifically, as shown in Table 1, a subset of 45 features was adopted after experimentation to capture the most important information of the state, regarding the board and the players' attributes.

Table 1. Features selected for state representation in Hearthstone.

Type	Feature	#
General Information	active player	1
	first player	1
	turn	1
Players' attributes	hero health points	2
	hero attack points	2
	# of cards in hand	2
	# of active secrets	2
	# of spells in hand for active player	1
	remaining mana for active player	1
Board	# of minions	2
	total mana cost of minions	2
	total attack points of minions	2
	total health points of minions	2
	# of minions able to attack	2
	total attack points of minions able to attack	2
	# of minions with taunt (special attribute)	2
	one hot encoded hero's class	2×9

The model's prediction is combined with a random rollout $z(s)$ executed at the simulation phase and the value being back-propagated to the traversed tree nodes results from Equation (6). Both predictions and random rollouts take values in $\{0, 1\}$, while parameter λ may be continuous in the $[0, 1]$ range, controlling each one's contribution to the final score. In the current setup, λ is set to 0.8 after experimentation, though its exact value did not seem to significantly affect the agents' behaviors.

$$combined\_score(s) = \lambda xgb_{pred}(s) + (1 - \lambda)z(s) \quad (6)$$

As the game states in the beginning of a game are quite trivial and not very informative about the possible outcome, an early simulation technique is used in the first turns of the game. Specifically, a threshold turn is defined (after experimentation), and for all turns prior to threshold, the simulation phase consists of two stages; at first, random actions are executed until threshold is reached and then the score of the new state is predicted and back-propagated, as described above. For game states beyond the threshold turn, the simulation occurs as normal from the current tree node [25].

Additionally, in order to adapt to the action space size during the rollout phase, new states are evaluated in a stochastic way. It is demonstrated that random rollouts may have a negative impact on the final prediction when the number of possible actions is relatively small. To balance this, the score of the nodes in the simulation phase is calculated, depending on the number of actions, as follows:

$$final\_score(s) = \begin{cases} xgb_{pred}(s), \text{if } u \geq \beta * action\_space \\ combined\_score(s), \text{otherwise} \end{cases} \quad (7)$$

where $U \sim U[0,1]$ is a random variable and β determines the action space size up to which the classifier's prediction may be used alone. In the tested environment, it was found that a threshold of 10 available actions leads to the desired functionality of the algorithm, and thus β was set to 0.1. This technique results in evaluations being determined in a high degree by the model's predictions in the case of small action spaces, improving the agent's performance.

7. Results

In the present work, the MCTS agent enhanced solely with pruning networks (referred to as `MCTS-PN` from now on); the one enhanced with the modifications described in Section 6.3 (`MCTS-xgboost`) and the one combining all adjustments (`MCTS-xgboostPN`) are evaluated. The latter two are tested against a simple version of the MCTS algorithm and a heuristic-driven minimax approach called Game State Value (GSV), which is the strongest algorithm provided by the Metastone framework. Additionally, these approaches are tested against each other. The `MCTS-PN` variant is tested only against the simple MCTS, as the absence of an evaluation function prevents it from being competitive against the other agents. Despite that, its performance is studied as well, since it is more efficient in terms of computational cost during play and it is completely domain-independent.

In the current implementation, the REST API used for the models' communication with the framework adds a significant delay on the running time, making it difficult to precisely determine the actual computational time and compare the different approaches in terms of time cost. In general, the model used for the states' evaluation is used more frequently than the pruning networks, making the respective variants more time consuming. However, since all models used during the execution of the algorithm are trained offline, the difference in the execution time of the simple MCTS and the suggested enhancements should not be prohibitive.

Tables 2–4 illustrate the performance of the proposed agents against MCTS, GSV, and `MCTS-xgboost`, respectively. Explicitly, each table presents the win rates of the tested approaches against a specific opponent over a set of 300 games. The experiments are carried out on three different decks corresponding to the main archetypes explained in Section 4. Particularly, the *hunter*, *warlock*, and *shaman* decks used by professional players in the 2014 Hearthstone World Championship [27] are selected for the *aggro*, *control*, and *midrange* strategies. Additionally, in order to examine the effect of the total iterations available for the tree search, each matchup has been carried out for 300, 500, and 1000 iterations. All games are played with the same deck type and number of iterations for both players in order to achieve unbiased results.

As shown in Table 2, the `MCTS-PN` variant's behavior differs depending on the datasets used to train the pruning networks. The agent using the uniform-based networks seems to gain greater advantage as the number of allowed iterations increases, in contrast with

the bimodal-based MCTS-PN, which performs better for the low iterations' budget. Overall, MCTS-PN(uniform) surpasses simple MCTS in all cases (though its performance is better in the case of 1000 iterations) whereas MCTS-PN(bimodal) achieves higher win rates for 300 and 500 iterations, but is outperformed in the case of 1000.

Concerning the MCTS-xgboost variants, the classifier integration boosts significantly the agents' performances. The number of iterations plays a significant role in the pruning networks' effects in this case as well. MCTS-xgboost without pruning achieves slightly better results in case of 500 iterations for the hunter and shaman decks and overall outperforms the bimodal-based pruning variant in cases of 300 and 500 available iterations. However, at least one of the pruning variants secures better results than MCTS-xgboost in seven out of nine individual setups, while MCTS-xgboostPN(uniform) reaches the highest win rates against MCTS over all different deck types and number of iterations examined. Even though the combination of supervised learning and pruning networks leads to higher improvement over the no-pruning MCTS-gxboost for 1000 iterations (+4.45%), the performance of all XGBoost-based agents individually decreases in relation to the total number of iterations. This is most probably ascribed to a higher improvement rate of the vanilla MCTS as the number of iterations increases, since the proposed—enhanced with supervised learning models—agents are able to achieve high performance, even for a low computational budget.

Table 2. Win rate per number of iterations against MCTS (the highest win rate per deck type and number of iterations is indicated in bold).

	Iterations			Deck Type
	300	**500**	**1000**	
MCTS-PN (uniform)	52.67	52.0	55.33	**hunter**
MCTS-PN (bimodal)	54.0	50.0	48.67	
MCTS-xgboost	71.33	**68.5**	61.67	
MCTS-xgboostPN (uniform)	**76.0**	68.33	**70.33**	
MCTS-xgboostPN (bimodal)	68.67	67.67	64.0	
MCTS-PN (uniform)	49.67	51.67	54.0	**warlock**
MCTS-PN (bimodal)	47.67	56.33	48.0	
MCTS-xgboost	75.0	75.5	72.33	
MCTS-xgboostPN (uniform)	**78.33**	**80.67**	74.0	
MCTS-xgboostPN (bimodal)	75.67	78.67	**75.67**	
MCTS-PN (uniform)	51.67	55.67	54.0	**shaman**
MCTS-PN (bimodal)	54.33	49.33	50.0	
MCTS-xgboost	74.33	**75.5**	68.33	
MCTS-xgboostPN (uniform)	73.67	74.0	**71.33**	
MCTS-xgboostPN (bimodal)	**75.67**	72.0	70.0	
MCTS-PN (uniform)	51.33	53.11	54.43	**overall**
MCTS-PN (bimodal)	52.0	51.89	48.89	
MCTS-xgboost	73.55	73.17	67.44	
MCTS-xgboostPN (uniform)	**76.0**	**74.33**	**71.89**	
MCTS-xgboostPN (bimodal)	73.33	72.78	69.89	

Subsequently, the approaches with the integrated classifier are evaluated against the GSV algorithm. Table 3 shows the corresponding win rates. It is clear that each deck type has a different degree of difficulty, which could not be captured in the experiments against MCTS. Particularly, the hunter deck appears to be the easiest to handle, with all three agents being able to outperform GSV. The shaman deck, on the other hand, seems to be the most complex one, with the best variant achieving a 26.67% win rate. Additionally, the format of each deck in conjunction with the number of available iterations seems to significantly affect the different algorithms, making it more difficult to draw specific conclusions. However, similarly to the tests against MCTS, MCTS-xgboost achieves marginally higher overall win rates against GSV in the case of 500 iterations than the pruning variants, while being

outperformed in the other two categories. The bimodal-based variant is superior in the low-iterations setups and `MCTS-xgboostPN(uniform)` is the best approach when a greater amount of computational resources is available. Regarding each variant individually, there is a clear boosting in performance as more iterations are used, verifying that the decrease in win rates against simple MCTS was caused by the great improvement of the latter.

Table 3. Win rate per number of iterations against GSV (the highest win rate per deck type and number of iterations is indicated in bold).

	Iterations			Deck Type
	300	**500**	**1000**	
MCTS-xgboost	46.67	50.5	**56.0**	**hunter**
MCTS-xgboostPN (uniform)	44.33	**55.67**	55.67	
MCTS-xgboostPN (bimodal)	**51.0**	52.33	55.67	
MCTS-xgboost	32.67	**43.0**	46.0	**warlock**
MCTS-xgboostPN (uniform)	**37.67**	40.67	49.0	
MCTS-xgboostPN (bimodal)	34.67	41.33	**49.33**	
MCTS-xgboost	**17.33**	**25.0**	24.0	**shaman**
MCTS-xgboostPN (uniform)	15.33	21.67	**26.67**	
MCTS-xgboostPN (bimodal)	17.0	20.67	25.67	
MCTS-xgboost	32.22	**39.5**	42.0	**overall**
MCTS-xgboostPN (uniform)	32.44	39.34	**43.78**	
MCTS-xgboostPN (bimodal)	**34.22**	38.11	43.56	

Finally, the two pruning variants enhanced with the XGBoost model are evaluated against the no-pruning `MCTS-xgboost`, with the results presented in Table 4. Even though `MCTS-xgboost` achieved higher win rates than the uniform-based pruning variant in some cases against MCTS and GSV (mainly in the 500 iterations case), `MCTS-xgboostPN(uniform)` outperforms it in all cases (with an exception of a draw) when tested against each other. The reason for this could be that different playing algorithms have specific advantages or weakness against others, similarly to the different deck types that follow the rock-scissors-paper pattern. The bimodal-based pruning agent is less consistent, achieving both the higher (57.0% in case of warlock deck and 500 iterations) and the lowest (43.67% in case of shaman deck and 300 iterations) win rates against `MCTS-xgboost`. Overall `MCTS-xgboostPN(uniform)` is the best approach in all three iterations' setups against `MCTS-xgboost`, while `MCTS-xgboostPN(bimodal)` outperforms it only in the 500 iterations setup.

In general, the bimodal-based networks lead to more aggressive pruning, as the datasets used for the training consist of actions with more easily separable expected rewards. On the other hand, data generated based on the uniform distribution consist of more similar (in terms of expected reward) actions, leading to more preservative pruning networks. This phenomenon is confirmed by the distribution of actions' visits on each case. Specifically, the distributions followed in `MCTS-xgboost`, `MCTS-xgboostPN(uniform)`, and `MCTS-xgboostPN(bimodal)` for 300, 500, and 1000 available iterations are depicted in Figures 3–5, respectively. These graphs relate to the shaman deck for the case of 20 available actions; however, they are indicative of the majority of cases concerning the different deck types and action spaces. As expected, in the pruning agents, the worst actions (as determined by the algorithms) are less visited than in `MCTS-xgboost`, leading to a greater percentage of visits appointed to the most promising actions. Between the two pruning approaches, the distribution's peak is higher in the `MCTS-xgboostPN(bimodal)` case, meaning a larger set of actions is pruned and the search is focused on the best actions faster than in `MCTS-xgboost(uniform)`. This behavior involves the risk of pruning the best action(s) in the early stages of the process, but could lead to more specialized and effective searches when actions are evaluated correctly in the first visits. This is the main reason for the

bimodal-based variant's inconsistency, as opposed to the uniform-based one, which is more stable, achieving the highest overall win rates among the compared variants in seven out of the nine total configurations.

Table 4. Win rate per number of iterations against MCTS-xgboost (the highest win rate per deck type and number of iterations is indicated in bold).

	Iterations			Deck Type
	300	**500**	**1000**	
MCTS-xgboostPN (uniform)	53.0	**54.67**	**52.33**	**hunter**
MCTS-xgboostPN (bimodal)	**54.67**	45.33	44.0	
MCTS-xgboostPN (uniform)	**54.0**	52.67	**54.33**	**warlock**
MCTS-xgboostPN (bimodal)	51.0	**57.0**	47.0	
MCTS-xgboostPN (uniform)	**51.67**	50.0	**53.67**	**shaman**
MCTS-xgboostPN (bimodal)	43.67	**53.67**	48.33	
MCTS-xgboostPN (uniform)	**52.89**	**52.45**	**53.44**	**overall**
MCTS-xgboostPN (bimodal)	49.78	52.0	46.44	

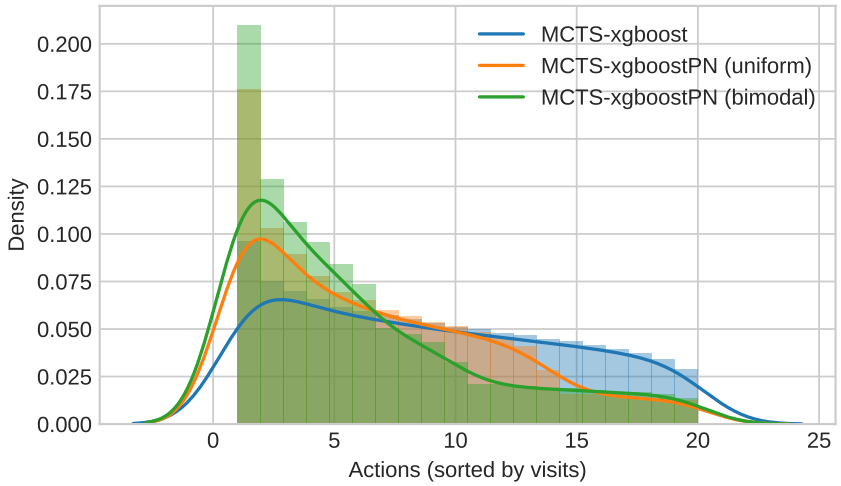

Figure 3. Actions' visits distributions with and without pruning networks (300 iterations).

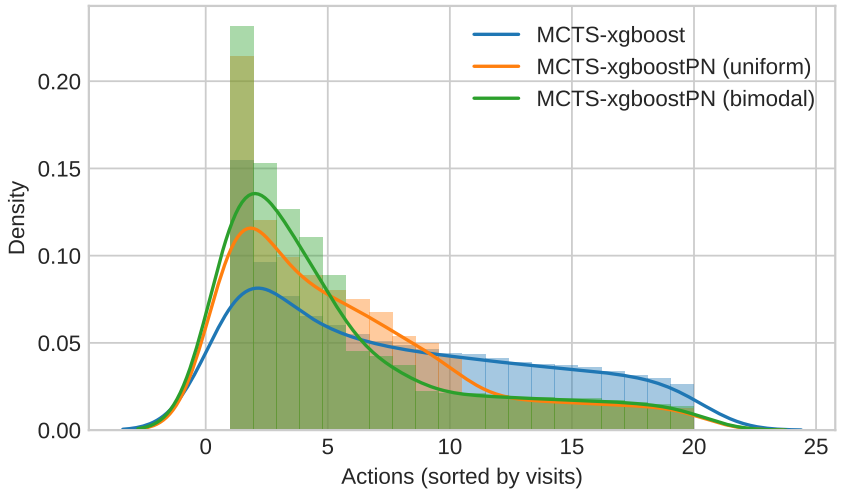

Figure 4. Actions' visits distributions with and without pruning networks (500 iterations).

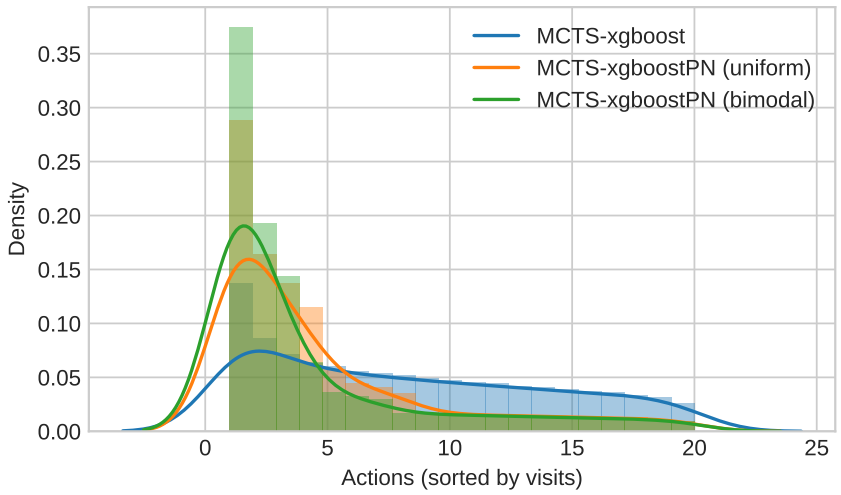

Figure 5. Actions' visits distributions with and without pruning networks (1000 iterations).

8. Conclusions

In this work, a methodology for limiting the action space during the execution of the MCTS algorithm by employing neural networks has been proposed. Concretely, two different neural networks are combined to predict the optimal number of iterations to execute and the number of actions to prune at that point, in an iterative manner. Additionally, the pruning networks' effects on the MCTS agents' performances are evaluated in conjunction with the integration of models predicting the game states' values. Different variants of the suggested algorithm were implemented and tested on the Hearthstone CCG. The conducted experiments highlight a consistent improvement over the no-pruning variants of the algorithm in different configurations, in terms of available computational resources, and demonstrate the influence of the branching factor in MCTS performance, especially in relation to the total number of tree search iterations.

Regarding the training of the pruning networks, a specially designed environment has been used to simulate the UCB algorithm on the MAB problem, in order to generate informative datasets for the pruning task. Depending on the distribution of actions' expected rewards, which was employed during the UCB simulations, two different variants of the proposed algorithm have been implemented, leading to quite different conclusions. Specifically, agents integrated with pruning networks based on uniformly distributed actions' values were very stable, exceeding baseline performance in most cases. On the other hand, the bimodal distribution-based ones exhibited more fluctuations in their performance, which is most probably attributed to the faster pruning produced by these networks. In this respect, it would be of particular interest to test the implemented algorithms in environments with different reward schemes, as well as to train pruning networks on data based on different distributions in order to verify the above-stated results and investigate the extent to which the algorithm is affected by the true actions' values' distributions.

Furthermore, soft pruning could be considered so as to not completely exclude actions that may have been falsely evaluated after a small number of visits. As observed in the experimental process, this scenario is highly probable, especially in the case of the bimodal-based pruning networks. Since the proposed method involves iterative pruning, actions' unpruning in different stages (determined by the networks) of the tree search could be considered to reduce the risk of quickly eliminating highly rewarding actions. In this context, a promising approach would be the unpruning of moves in a probabilistic manner, as well as the use of a separate model to indicate the actions to unprune in addition to the existing pruning networks.

Finally, as Hearthstone is a game of imperfect information, several assumptions have to be made concerning hidden information necessary for the algorithm's functionality (e.g., opponent's cards). The game's internal stochasticity additionally impacts the flow

of transitions between game states to a notable degree and consequently complicates the algorithms' evaluation. Under this scope, the proposed pruning technique could be also tested in perfect-information, deterministic games, such as chess or Go to confirm its influence on the agent's performance, as it has no specific requirements and can be applied to any type of game. Nevertheless, the results achieved despite the introduced randomness of Hearthstone are quite encouraging regarding the algorithm's capability of generalization.

Author Contributions: Conceptualization, T.P. and G.A.; methodology, T.P.; software, T.P.; validation, T.P. and G.A.; formal analysis, T.P.; investigation, T.P.; resources, T.P. and G.A.; data curation, T.P. and G.A.; writing—original draft preparation, T.P.; writing—review and editing, G.A. and A.S.; visualization, T.P.; supervision, A.S.; project administration, A.S.; funding acquisition, A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CCG	Collectible Card Game
GSV	Game State Value
MAB	Multi-Armed Bandit
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
RAVE	Rapid Action Value Estimate
UCB	Upper Confidence Bound
UCT	Upper Confidence bounds applied to Trees

References

1. Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. Mastering the game of go without human knowledge. *Nature* **2017**, *550*, 354–359. [CrossRef] [PubMed]
2. Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; et al. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* **2020**, *588*, 604–609. [CrossRef] [PubMed]
3. Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv* **2017**, arXiv:1712.01815. [CrossRef]
4. Thakoor, S.; Nair, S.; Jhunjhunwala, M. Learning to Play Othello without Human Knowledge. 2016. Available online: https://www.scribd.com/document/388438020/Learning-to-Play-Othello-Without-Human-Knowledge (accessed on 28 January 2022).
5. Katehakis, M.N.; Veinott, A.F. The Multi-Armed Bandit Problem: Decomposition and Computation. *Math. Oper. Res.* **1987**, *12*, 262–268. [CrossRef]
6. *Hearthstone*; Blizzard Entertainment: Irvine, CA, USA, 2014.
7. Huang, J.; Liu, Z.; Lu, B.; Xiao, F. Pruning in UCT algorithm. In Proceedings of the 2010 International Conference on Technologies and Applications of Artificial Intelligence, Hsinchu, Taiwan, 18–20 November 2010; pp. 177–181.
8. Kocsis, L.; Szepesvári, C. Bandit based monte-carlo planning. In *European Conference on Machine Learning*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 282–293.
9. Chaslot, G.M.J.; Winands, M.H.; Herik, H.J.V.D.; Uiterwijk, J.W.; Bouzy, B. Progressive strategies for Monte-Carlo tree search. *New Math. Nat. Comput.* **2008**, *4*, 343–357. [CrossRef]
10. Coulom, R. Computing "elo ratings" of move patterns in the game of go. *ICGA J.* **2007**, *30*, 198–208. [CrossRef]
11. Couëtoux, A.; Hoock, J.B.; Sokolovska, N.; Teytaud, O.; Bonnard, N. Continuous upper confidence trees. In *International Conference on Learning and Intelligent Optimization*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 433–445.
12. Sephton, N.; Cowling, P.I.; Powley, E.; Slaven, N.H. Heuristic move pruning in Monte Carlo Tree Search for the strategic card game Lords of War. In Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games, Dortmund, Germany, 26–29 August 2014; pp. 1–7.

13. Ouessai, A.; Salem, M.; Mora, A.M. Improving the Performance of MCTS-Based μRTS Agents Through Move Pruning. In Proceedings of the 2020 IEEE Conference on Games (CoG), Osaka, Japan, 24–27 August 2020; pp. 708–715.
14. Hsu, Y.J.; Liebana, D.P. MCTS Pruning in Turn-Based Strategy Games. In Proceedings of the AIIDE 2020 Workshops Co-Located with 16th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2020), Worcester, MA, USA, 19–23 October 2020.
15. Duguépéroux, J.; Mazyad, A.; Teytaud, F.; Dehos, J. Pruning playouts in Monte-Carlo Tree Search for the game of Havannah. In *International Conference on Computers and Games*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 47–57.
16. Gelly, S.; Silver, D. Combining online and offline knowledge in UCT. In Proceedings of the 24th International Conference on Machine Learning, Corvalis, OR, USA, 20–24 June 2007; pp. 273–280.
17. Auer, P. Using Confidence Bounds for Exploitation-Exploration Trade-Offs. *J. Mach. Learn. Res.* **2003**, *3*, 397–422.
18. Hoeffding, W. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*; Springer: Berlin/Heidelberg, Germany, 1994; pp. 409–426.
19. MetaStone Simulator. Available online: https://github.com/demilich1/metastone (accessed on 28 January 2022).
20. Hearthstone AI Code Repository. Available online: https://github.com/ails-lab/hearthstone_ai/ (accessed on 28 January 2022).
21. Coulom, R. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International Conference on Computers and Games*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 72–83.
22. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015; Conference Track Proceedings; Bengio, Y., LeCun, Y., Eds.; 2015.
23. Silver, D.; Huang, A.; Maddison, C.J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. Mastering the game of Go with deep neural networks and tree search. *Nature* **2016**, *529*, 484. [CrossRef] [PubMed]
24. Świechowski, M.; Tajmajer, T.; Janusz, A. Improving hearthstone ai by combining mcts and supervised learning algorithms. In Proceedings of the 2018 IEEE Conference on Computational Intelligence and Games (CIG), Maastricht, The Netherlands, 14–17 August 2018; pp. 1–8.
25. Papagiannis, T.; Alexandridis, G.; Stafylopatis, A. Applying Gradient Boosting Trees and Stochastic Leaf Evaluation to MCTS on Hearthstone. In Proceedings of the 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 14–17 December 2020; pp. 157–162. [CrossRef]
26. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16), San Francisco, CA, USA, 13–17 August 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 785–794. [CrossRef]
27. 2014 Hearthstone World Championship. Available online: https://hearthstone.fandom.com/wiki/2014_Hearthstone_World_Championship (accessed on 28 January 2022).