A Methodology for Creating Generic Game Playing Agents for Board Games

Mateus Andrade Rezende* Luiz Chaimowicz†

Universidade Federal de Minas Gerais (UFMG), Department of Computer Science, Brazil

Abstract

General Game Playing (GGP) consists in developing agents capable of playing different games. Normally these agents go through an initial learning process to gain some knowledge about the game and be able to play it well. In board games, this normally requires learning how to evaluate a great variety of states in a game tree. This work introduces a methodology called UCT-CCNN to generate value functions for evaluating states in generic board games. The UCT-CCNN method executes a large number of matches between Monte Carlo Tree Search (MCTS) agents using a tree policy known as Upper Confidence Bounds for Tree (UCT) in an off-line process that generates a database of state-utility examples. From those examples, a value function for the game states is learned through the use of constructive neural networks known as Cascade Correlation Neural Networks (CCNN). The UCT-CCNN method was tested with two classical board games: Othello and Nine Men's Morris, and the obtained agents were capable of winning matches against agents specifically developed for these games. Moreover, the UCT-CCNN method can control the strength of the obtained agent, ensuring a flexible method capable of generating intelligent agents with different levels of difficulty. Another set of experiments shows that the UCT-CCNN method can also be easily integrated into any algorithm such as the MCTS itself, leading to higher winning rates when compared to the standard UCT with the same number of simulations.

Keywords: General Game Playing, Board Games, Monte Carlo Tree Search, Cascade Correlation Neural Networks.

1 Introduction

Normally, intelligent agents are developed for playing a specific game, exploring the unique characteristics and specific domain knowledge of each game. One problem with this approach is the need to develop a different agent for each game. Thus, a novel area of research named *General Game Playing* has emerged with the objective of creating agents capable of efficiently playing different games, maybe with an initial learning process [23]. The name GGP comes from the AAAI GGP competition, in which submitted agents are tested in many games described in the Game Description Language (GDL) [18]. The recent winners of the AAAI GGP competition are mostly based on Monte Carlo Tree Search [3], a technique that explores the game to infer state utilities using simulations. In the GGP competition, agents have a short time to run simulations in order to estimate better utilities for actions because those simulations run during the official matches of the competition. A greater challenge would be, given the rules of any game, to generate in a completely unsupervised way an intelligent agent that is competitive compared to specific agents for the game.

In this paper, we present a methodology called UCT-CCNN for creating generic game playing agents for board games. Thus our test scenarios are composed by two players, zero-sum, perfect information, deterministic, discrete and sequential games. These games are excellent domains for Artificial Intelligence (AI) experiments because they have a controllable environment defined by simple rules, but that typically have complex strategies and a large state space.

The UCT-CCNN receives as input the rules of any game, according to the constraints previously described, and generates as output a value function for the game states. Basically, a large number of matches are played between Monte Carlo Tree Search (MCTS) agents using a tree policy known as Upper Confidence Bounds for Tree (UCT) in an off-line process that generates a database of state-utility examples. An important parameter of the MCTS algorithm, known as exploration constant, is optimized for a specific game using the Cross Entropy Method (CEM). The generated examples' database goes through a filtering process to eliminate utilities that probably do not have the necessary accuracy to ensure good decisions. From those examples a value function for the game states is learned with the use of constructive neural networks known as Cascade Correlation Neural Networks, capable of iteratively building an architecture that adapts itself to the submitted problem, thus allowing the GGP characteristic of this work. A trained neural network represents the obtained value function.

UCT-CCNN is a GGP learning method since it does not use domain-specific knowledge. Unlike agents participating in the AAAI GGP competition, the UCT-CCNN requires an earlier stage of off-line processing before the agent is capable of effectively playing a new game. In this way, the generated agent will present "strong" decisions from the beginning of the matches, but will not be capable of learning during them or playing without the execution of the previous learning phase. The UCT-CCNN method was tested with two board games, *Othello* and *Nine Men's Morris*, and the obtained value functions were integrated into the Minimax search with Alpha-Beta Pruning algorithm. The resulting agents were capable of winning against specific-domain agents.

This paper is organized as follows: Section 2 discusses some related works and present some background on the techniques used in this work. Section 3 describes our methodology, detailing the steps taken by UCT-CCNN to implement the agents. The experiments are presented in Section 4 while Section 5 brings the conclusions and directions for future work.

2 Background and Related Work

Our methodology relies on the Monte Carlo Tree Search (MCTS) method in conjunction with a specific type of neural network called Cascade Correlation Neural Network (CCNN). Moreover, the Cross Entropy Method is used to determine some parameters of the MCTS. This section presents a brief overview of these methods and also discusses some related work.

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the de-

*e-mail: mandraderezende@ufmg.br

†e-mail: chaimo@dcc.ufmg.br

cision space and building a search tree according to the results [3]. Given a game state, the MCTS returns an action to be executed in that state. MCTS maintains a game state tree that is built incrementally and asymmetrically.

At the beginning of its execution, the MCTS algorithm receives a game state and creates a game tree containing only the root node representing the received state. After the process initialization, the MCTS starts an iterative process divided into four stages called: Selection, Expansion, Simulation and Backpropagation.

In the first stage, called Selection, the tree nodes are selected by the tree policy. The tree policy tries to balance between exploration (selecting nodes with few samples) and exploitation (selecting more promising nodes with higher average utility). From the tree root node, nodes are selected by the tree policy until a node with at least one child not expanded is reached and then the second stage begins from that node.

In the second stage, one child is chosen uniformly at random among the children that have not been expanded. The selected child node is expanded, which means that it is added to the tree.

The third stage is called Simulation and starts from the expanded node in the previous stage. The moves, or actions, are selected during the simulation by a default policy, which in its simplest form selects actions uniformly at random. In the end of a simulation, when a terminal node is reached (end of the match), the real utility for the end game state is returned according to the game rules.

The fourth and last stage is called Backpropagation and begins upon reaching a terminal node at the end of the simulation phase. In this last phase, the game tree is updated based on the final utility of the simulation. The number of visits is incremented and the average utility is updated for each node in the path taken by the simulation starting in the expanded child back to the root node.

The four stages of the MCTS algorithm are shown in Figure 1. In the figure, on the Selection stage the highlighted nodes were selected by the tree policy known as Upper Confidence Bounds for Tree (UCT), on the Expansion stage the highlighted node is expanded, on the Simulation stage the smaller nodes represent selected nodes by the default policy and on the Backpropagation stage the highlighted nodes have their statistical values updated.

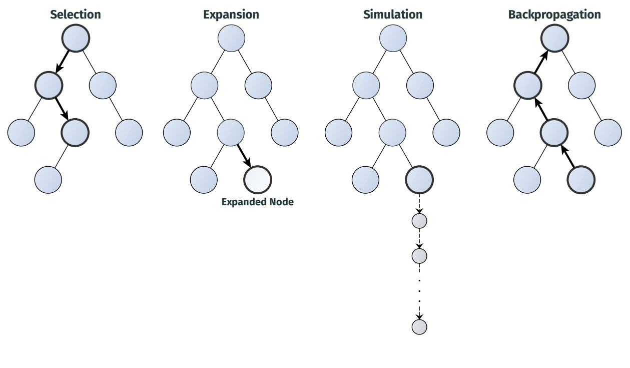

Figure 1: Stages of the MCTS algorithm.

If a simulation ends up with a low utility value, it does not mean that the expanded state is poor. Statistically speaking, there is a confidence interval for a state's expected utility, given how many times that state has been selected during the selection phase. Optimistic policies exploit the upper limit of this confidence interval in order to find the best action to take. The Upper Confidence Bounds 1 (UCB1) algorithm ensures a policy within a constant factor of the optimal bound on the growth of the regret value [1], which is the utility loss for not taking the best action. In addition, UCB1 is simple and efficient. In the Upper Confidence Bounds for Tree (UCT) algorithm, the UCB1 was incorporated into the tree policy, where a child node (j) is selected in the Selection phase in order to maximize the UCT value:

$$\text{UCT} = \overline{X_j} + 2\,C_p\sqrt{\frac{2\ln n}{n_j}},$$

where $\overline{X_j}$ is the observed average utility for the node j, n is the number of times that the parent node j was visited, n_j is the number of times that the child node j was visited and $C_p > 0$ is a constant. It is considered that for $n_j = 0$, the UCT value for the state j is infinite so that states never explored before have priority to be expanded. States with the same UCT value should be randomly selected. The constant C_p is called exploration constant and can be adjusted to increase or decrease the priority in exploration rather than prioritize states with the highest observed average utility. The optimal value for the exploration constant depends on the problem being addressed and the improvements implemented in the MCTS policies. Each node stores a number $N(s)$ of visits and a total accumulated utility $Q(s)$ of the simulations passing through the state s, so $Q(s)/N(s)$ is an approximation to the average utility of state s.

In this text MCTS-UCT is an abbreviation for the MCTS algorithm that uses UCT as a tree policy and whenever the abbreviation MCTS is mentioned without specifying the policy, it is not relevant to the context. It is known that if enough execution time and memory is given to the MCTS-UCT algorithm, the game tree converges to the minimax one [16, 17]. Among the main features of the MCTS are that it is independent of domain-specific knowledge and the tree growth is asymmetrical, favoring more promising regions of the state space. It is worth mentioning that other policies may be used instead of the UCT tree policy.

When using the MCTS, many iterations of the algorithm must be executed to obtain good results. Unfortunately, the decision of which action to take in games cannot take too long. In GGP, as in the AAAI GGP competition [13], an agent must be capable of playing any game with a restricted time of preparation for the match and for taking an action (startclock and playclock, respectively). If the server does not receive a response until the timeout, a random action is selected for the agent. Due to this restricted time, the most successful agents use learning mechanisms during the official matches of the tournaments, and the learned information is used to guide the MCTS search, which is parallelized in order to be able to execute as many simulations as possible in the short time available.

The approach of this work is somewhat different from the context of agents involved in the GGP competition. The objective is to generate an agent that has strong decisions early in the match, which is specially important when playing against a strong opponent. The agent can still be considered a GGP agent because it does not use domain-specific knowledge to learn, just the rules of the game. The main difference is in a prior offline learning process before the agent is able to play effectively. With this approach, the agent is strong from the beginning of the match, and its "strength" can be controlled by the learning process, if there is interest in generating agents with varying levels of strength (difficulty modes as there are in many games). The greater the number of MCTS simulations the greater the accuracy of the estimated utilities, which will lead the agent to make stronger decisions, increasing its difficulty level. The downside would be that the agent does not learn during official matches.

MCTS is a good strategy when there are no predefined strategies nor examples of actual matches, due to its ability to estimate utility values for game states without domain-specific knowledge. But, unfortunately, it is impractical to generate and store a complete game tree through MCTS. A possible solution is to use an Artificial Neural Network (ANN) to generalize a value function to other states based on the expected utility values processed by MCTS. In this paper, we use a specific type of ANN called Cascade Correlation Neural Network, which is described in the next section.

2.2 Cascade Correlation Neural Network

Creating a neural network whose architecture is capable of generalizing different problems is a challenging task. In order to achieve a satisfactory convergence of the neural network and thus correctly generalizing entries never seen before with minimal error, a specific knowledge of the problem is needed in order to make the necessary adjustments in the neural network parameters. Among these parameters, the most critical are related to the network architecture (for example, how the hidden neurons are organized and how they connect between the input and output layers) and to the parameters of the learning algorithm, such as the learning rate parameter of the Backpropagation algorithm. In the Backpropagation algorithm, an error is calculated comparing the input example and its expected output value, and this error is propagated back adjusting the weights of the connections according to its corresponding gradient. This process can lead to convergence at a local minimum, which is avoided by random initialization of the weights. In addition, multiple networks are trained and the one with the best convergence is chosen. The step-size parameter, known as learning rate, determines the size of the adjustment made in the weights and therefore influences in the network convergence. Bad parameter values can lead to overfitting, which means a low error for the training set, but a high error for the test set that represents a set of examples never seen before by the network. There is no obvious way to adjust these parameters and there are no guarantees involved.

The problems described above can be minimized by using constructive neural networks such as Cascade Correlation Neural Networks (CCNN). The CCNN has a special architecture that grows to adapt to the problem, and also has a different learning process that reduces computational costs and solve many problems of the backpropagation algorithm [8] [2]. The CCNN stars with a minimal architecture with only the input and output layers. Neurons are added to the network one at a time and each one in a new hidden layer connected to all previous layers. Figure 2 shows a CCNN with two hidden neurons added.

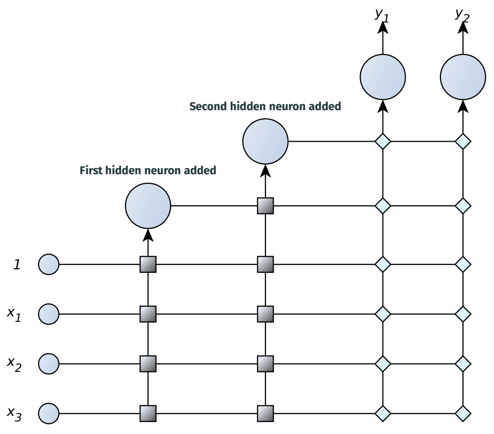

Figure 2: Basic CCNN architecture.

The CCNN training consists in an iterative process in which a neuron is trained and added to the network at each iteration. Before being added to the network, a neuron is trained and, after being added, has its weights of the input connections frozen. To train a new neuron, the input weights are calculated through gradient ascent to maximize the covariance C between the output of the candidate neuron and the output of the network built so far. All the input weights of the output layer, including the output weights of the candidate neuron to be added, are trained after the candidate is added to the network. The covariance C is defined according to the following formula:

$$C = \sum_{o \in O} \left| \sum_{s \in S} (y_s - \bar{y})(e_{o,s} - \overline{e_o}) \right|,$$

where O is the set of the network output neurons, S is the set of the training examples, y_s is the candidate output for the example s, $e_{o,s}$ is the output error of the output neuron o for the example s, $\bar{y}$ and $\overline{e_o}$ are the mean values y_s and $e_{o,s}$ over the examples set S.

Another approach to train the candidate neurons was implemented by Fahlman, the creator of CCNN architecture, known as Cascade 2 [19]. In that approach, the candidate neuron is trained to minimize through gradient descent the difference $C2$ between the output error of the output neurons and the input of the output neurons received from the candidate being trained. The difference $C2$ is represented by the following formula:

$$C2 = \sum_{o \in O} \sum_{s \in S} (e_{o,s} - y_s \cdot w_{y,o})^2,$$

where $e_{o,s}$ is the output error of the output neuron o for the example s, y_s is the candidate neuron output for the example s and $w_{y,o}$ is the weight of the candidate neuron y to the output neuron o. Both the candidate neuron input weights and the candidate neuron output weights are updated to minimize the difference $C2$. The weighted candidate neuron activation will have a value close to the network error by minimizing the difference $C2$, so the candidate output weights must have their signal inverted to contribute to the minimization of the network error. The Cascade-Correlation approach that uses the C maximization is best for classification problems, while the Cascade 2 approach that uses the $C2$ minimization is best for regression problems [19].

Several candidate neurons can be trained independently with different activation functions and different random initialization of weights. The neuron with greater covariance C or smaller difference $C2$ is selected, discarding the others. When adding a neuron in the network the weights of its input connections are frozen, and in the case of covariance maximization C, all the weights of all the neurons connected to the output layer are trained again. The adjustment of weights in any step is done through some learning algorithm like Backprop, Quickprop [7] or Rprop [20]. The name CCNN refers to the correlation because, in the original work, in a first attempt the correlation was used for training the candidate neurons, but later it was decided that covariance would be the best option since it worked better in many situations [8].

For this work the learning algorithm chosen to train the weights of the CCNN was the iRprop [14], which is an improved variant of the original algorithm Rprop. In Rprop the update of each weight is based on the signal of the partial derivative of the error in relation to the weight, making the parameter step-size independent of the partial derivative absolute value. Roughly, being the weight w_{ij} of the connection between the neuron j and the neuron i and E a differentiable error mean with respect to the weights, if the partial derivative $\partial E / \partial w_{ij}$ has the same signal for the consecutive steps of the weight updating, the step-size is incremented, if the signal changes the step-size is decremented. Each weight has its own step-size.

The step-size adjustment constant, the initial values of the step-sizes, and the maximum and minimum limits for the step-sizes are parameters of the algorithm. This learning technique dispenses the learning rate parameter because the step-size value is dynamically adjusted, making the *Rprop* ideal to constructive architectures such as CCNN. In the *iRprop* variant, previous adjustments of the weights are reverted in case of signal change of the partial derivative only if the overall network error has increased. In addition, when the partial derivative changes the signal, the value of the signal is ignored in the next iteration and the weight update will occur without first changing its related step-size.

2.3 Cross-Entropy Method

The Cross-Entropy Method (CEM) was originally developed as a simulation method to estimate probabilities of rare events and later also came to be used as a stochastic optimization method [21]. In this work, the CEM is used to optimize the exploration constant of the UCT tree policy, and thus ensure better estimates for the states' utility in the MCTS [5].

The CEM involves an iterative procedure divided into two steps. In the first step, a random sampling of parameter value examples is performed through some parameterized probability distribution. In the second step, the parameters of the distribution used for the examples generation are updated based on the produced data, in order to generate better examples in the next iteration. An important characteristic of the CEM is its asymptotic convergence, where under mild conditions of regularity the process ends with probability 1 in a finite number of iterations [6].

As will be discussed in Section 3, we use the Cross Entropy Method to determine the best exploration constant (C_p) to be used by the MCTS in each of the games. The CEM has been successfully used to estimate MCTS parameters faster than other methods and with guaranteed convergence [5].

2.4 Related Work

In the GGP competition, the recent winners are all based on MCTS-UCT [3], as is the case of the CadiaPlayer, who won the competition in the years 2007, 2008 and 2012 [9]. One of the restrictions in the use of MCTS for GGP is the simulation time that is limited to the maximum response time allowed to the agent during the matches. Another problem is that uninteresting portions of the search tree are explored for most of the simulation time, especially in the first simulations, which leads to a need for longer simulation time or techniques to redirect the MCTS search.

There are techniques that redirect the MCTS selection and simulations based on accumulated data from the simulations, which are often simple features of the board game and actions. Among such techniques are the *First-Play Urgency* [12], *All Moves As First* (AMAF) techniques such as *Rapid Action Value Estimation* (RAVE) [11], selection improvements such as *Progressive Bias* [4], and *Pruning* techniques. These and other techniques are listed in [3], and most of them were used in winning agents from the GGP competition. Many of these techniques do not guarantee better win rates for any games, even though they improve agent performance on many of them [9]. Worth mentioning that some of these techniques require specific-domain knowledge, which is not interesting for GGP.

The MCTS algorithm, besides being the main algorithm used in GGP agents, has also proved to be a decisive tool in the creation of AlphaGo, an intelligent agent for the GO game capable of beating professional players, thus solving one of the biggest challenges of the AI [22]. AlphaGo uses general purpose techniques such as Deep Learning Neural Networks and MCTS, and is the first algorithm capable of beating experienced Go players in the standard board size (19x19). This shows the power of the MCTS algorithm if the search can be directed with the use of auxiliary techniques, such as the Neural Networks applied to the MCTS in AlphaGo. Despite the use of general purpose techniques, AlphaGo uses domain-specific knowledge in order to obtain a stronger agent: the neural networks are trained from examples of real matches between professional players and some features of the game Go are manually set to ensure their evaluation and improve learning on the network.

This work took inspiration from the CadiaPlayer [9] and its strategy for the generalization of the MCTS algorithm for GGP and non-domain-specific improvements and also from the AlphaGo [22] and its strategy of integrating MCTS with neural networks. The proposed technique improves some of the approaches found in literature. Specially, Regarding AlphaGo, the proposed technique does not depend on examples of professional moves and it is not necessary to define the neural network architecture for each different game. Regarding the MCTS improvements of the GGP agents, the proposed technique uses a full set of game state features and the generated agents are capable of performing strong actions early in the match.

3 Methodology

Our methodology is divided in two main steps. Firstly, we run a series of MCTS-UCT simulations in order to infer the utilities of a large number of game states. The exploration constant used by the simulations is determined by running the Cross Entropy method, so that it can be adapted to different games. To generalize the results obtained by the MCTS-UCT to other states, we use a Cascade Correlation Neural Network, trained with the iRprop algorithm. As discussed in the previous section, the architecture and parameters of this network do not need to be defined beforehand, which is important for the different game scenarios faced by our methodology. The details of each of these steps are presented in the next sections.

3.1 Off-line examples generation via MCTS-UCT

The first phase of the UCT-CCNN method is the generation of state-utility examples extracted from matches between MCTS-UCT agents, called MCTSPlayers. In this phase, it is crucial that the simulation time of the MCTSPlayers to be as long as possible, which will require more system resources, in order to generate state-utility examples with better accuracy.

In order to build the game tree, game description interfaces must be implemented, which indicate the game initial state, the actions that are possible from a given state, the resulting state from taking a valid action and if a state is a terminal one. The UCT exploration constant to be used in the matches is obtained by the CEM optimization method, in which the examples generated in each iteration represent a possible value for the constant, following a uniform distribution that generates values in the range $[0.2, 2.0]$. This range of possible values is suggested by [5]. By default, the initial distribution mean is the average of the lower and upper bound and the standard deviation is the half of the distance between the lower and upper bound.

First, i matches are executed from the initial default state between two MCTSPlayer agents. The greater the maximum simulation time t defined by the user, the better will be the utility estimation for the states. It all depends on the available time, computational power in terms of the amount of threads and memory and agent's target strength. Usually, a MCTS-UCT agent only returns an action from a given state that leads to the child state with the highest average utility. In the examples generation phase, before an agent returns the best action, the first child states of the current game state are persisted as examples in a database specific to the current match and player.

All the first children of the current state are persisted in order to generate examples of both good and bad states for better generalization of the neural network to be trained. Each persisted example contains the state in binary form, the number of visits to the state during simulations and the average utility computed for the state, represented by the formula $\frac{Q(child)}{V(child)}$. Figure 3 represents a match between two MCTS-UCT agents with states that are persisted as examples. Before an agent makes a move on his turn, the first children of the node relative to the current game state, which is the root node of the MCTS tree, are persisted in a database specific to the match and the player. This is because, afterward, the filtering is performed on the persisted examples and it is necessary to identify the examples of a match relative to the player who won the match.

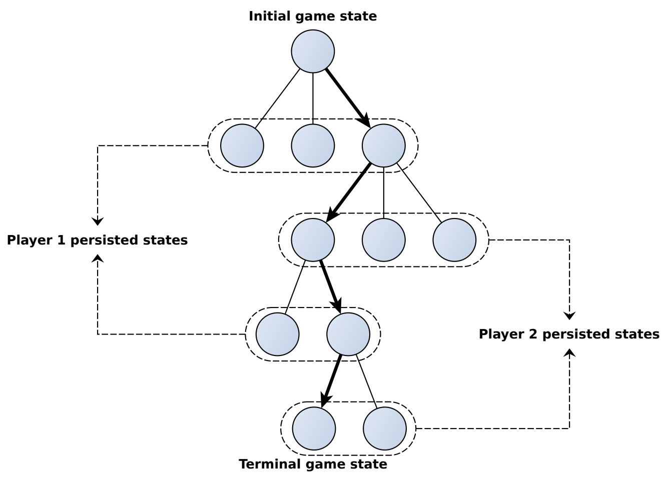

Figure 3: States that are persisted as examples in a match.

As more matches are executed, the number of generated examples increases, which will favor the neural network training. Multiple matches are performed because of the stochastic nature of the MCTS agents, ensuring a greater coverage of generated examples since, within each match, the MCTS agents explore different regions of the search space. However, since there are two “strong” agents playing against each other, a portion of the search space more consistent with actions chosen by “strong” players will be explored. In order to generate examples with not very common states, specially in situations where the opponent is not experienced, matches that begin from randomly generated valid intermediate states are also executed.

To generate the randomized intermediary states, matches are executed between random agents that choose actions uniformly at random. The visited states during the matches are persisted in a database. Initially $(e/J)+1$ matches are executed, where e is the number of random states to be generated and J is the average number of moves in a match for the related game, calculated from the matches executed previously from the default initial game state.

Of all the generated random states, those closest to the initial default state are first selected through the matches, one per match, until e unique random states are selected. If after that random states are still missing, new matches with random agents are executed until e unique random states are generated. Matches that starts in random states near the default root state will generate more examples and that is why these random states are preferred.

For each generated random state, r matches are executed between MCTSPlayer agents with the random state as initial one. From these matches, examples of state-utility are also generated and persisted in the database using the same logic.

3.2 Filtering and CCNN training

This section presents the preparation process of the test and train data for the CCNN. Each example generated in the previous phase contains the game state in its binary form, an average utility calculated by the MCTS-UCT and the number of visits from the simulations in the MCTS-UCT, plus the player and the match result.

First, the example database is filtered to deliver the best examples to the neural network, in an attempt to work around the high variance of the MCTS-UCT utility estimation. Of all the generated examples in a match, only those of the player who won the match are considered because probably the estimated utilities of those examples have led to choices closer to the optimal policy.

Among the different executed matches, repeated states may have been generated as examples. To favor a better training of the neural network the utility that will lead to a policy closer to the optimal policy must be selected. The utility calculated by a greater number of simulations presents a lower statistical variability, approaching the real utility of the state (the one that leads to the optimal policy). Therefore, in the case of repeated states, the one with the highest number of visits in selected. Another possibility would be to compute a new utility value as the mean of the utilities obtained for the repeated states.

After the examples filtering, the selected examples are separated into two data sets, according to the parameter t relative to the percentage of the total examples to be used in the test set. The default value for that parameter is 10%. The test examples are chosen uniformly at random from all examples. The remaining 90% of the examples are separated into a training set. The network is not trained with the examples from the test set.

Even though it is a constructive algorithm, it is necessary to specify the input and output configuration of the neural network since in the CCNN only hidden neurons are added. Therefore, it has been defined that the number of neurons in the input layer is equal the number of bits in the binary representation of the game state. Each input neuron receives zero or one according to the binary sequence of the state to be evaluated. The output layer consists of a single neuron whose output represents the expected utility for the state received as input.

The utility value calculated by the MCTS-UCT algorithm is in the range $[-1,1]$, because when the utility value calculated for the examples generation is equal to $\frac{Q(child)}{V(child)}$, where $Q(child)$ is the number of wins minus the number of losses of the player who made the move in the parent state and $V(child)$ is the number of visits to the child state. If in all n visits to the child state the player who made the move won in the simulation result, the calculated utility would be $(n-0)/n=1$, and if in all n visits the player losses in all simulations the utility would be $(0-n)/n=-1$.

For that reason, the hyperbolic tangent function $tanh(x) = \frac{1}{1+\exp^{-x}}$ was chosen as the activation function for the output neuron because it returns a value in range $[-1,1]$. One of the weaknesses of that approach is that the neural network only learns to evaluate a state individually, without a transition represented by a state plus an action. In order to evaluate a child state, it is necessary to calculate the transition from the parent state with the executed action to obtain the child state. Because of this, the neural network is best used in situations that the transition to a child state would be calculated anyway. For situations in which one action must be selected in a given state, it is necessary to calculate the transition to all the child states, in order to evaluate them with the trained neural network.

During the CCNN training, at each neuron addition, the mean square error in the training set and in the test set are calculated and then a copy of the network constructed so far is saved along with the test and train error obtained. The network training continues until the maximum number of neurons is reached.

When the CCNN training is finalized, the iteration of the neural network with the lowest error is chosen since it is the network that best generalized to the states never seen before. The file of the best network is then returned by the algorithm. This file can be used as a value function in conjunction with other algorithms, as a way of guiding the search in the MCTS algorithm or as an evaluation function in the Minimax algorithm, as will be shown in the following sections.

4 Results

We evaluate the method UCT-CCNN by applying it to two games: Nine Men's Morris and Othello. In the generation of examples via MCTS-UCT, two different groups of examples are generated for each game: one with 200 seconds and other with 600 seconds of simulation time. A CCNN network is trained for each group of examples, resulting in two neural networks for each game. In order to evaluate the strength of the value function associated to each of the obtained neural networks, these networks are used as evaluation functions in two scenarios for each game: (i) as Minimax agents for playing the game and (ii) as the tree policy for MCTS agents.

Nine Men's Morris, also called Mill, is a two-player board game with 24 intersections, where the pieces are placed. Each player has 9 pieces and chooses a piece color. Both players start at the first phase, where the pieces must be placed on the board, one per turn, in any free position. When a player puts all his nine pieces, he moves on to the second phase. In the second phase, a player can move his pieces to adjacent free positions, one per turn. A player enters the third phase when only three of his pieces remain on the board. In the third phase, a player can move his pieces to any free position of the board. When a player's move, in any phase, results in three of his pieces consecutively aligned horizontally or vertically, the player forms a mill and can choose an opponent's piece to be removed from the board. When removing an opponent's piece, the player must give preference to a piece that is not in a mill. A player loses when only two of his pieces remain on the board and the game ends in a draw when a board configuration is repeated. Figure 4 shows a white player action that leads to a mill formation and therefore to a capture of one opponent's piece.

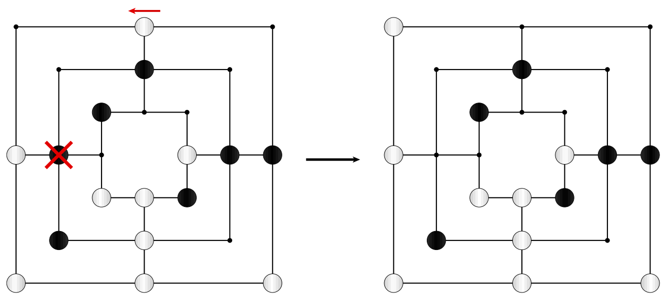
Figure 4: A white player move in the Nine Men's Morris game.

Othello, also called Reversi, is a two-player board game with 64 pieces that are black on one side and white on the other. The player that begins a match must place the pieces on the board with the black side up. The game begins with four pieces in the center with the same color on the same diagonal. Each player must place a piece in a position next to an opponent's piece that results in at least one piece captured, *i.e* at least one of the opponent's pieces will be trapped between two of the player's pieces in any direction (vertically, horizontally or diagonally). The captured pieces are turned to change their color. The game ends when neither player is able to make a valid move and the player with the most pieces on the board wins. Figure 5 shows a white player action that leads to a capture of two opponent's pieces.

The experiments were executed in a machine with Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz with 40 cores / 80 threads and 250 GB of RAM. Only 64 threads were utilized. The execution time of the experiments would be almost the same in different machines, but the faster the machine the greater the number of MCTS simulations performed.

We firstly ran the Cross Entropy Method (CEM) to determine the exploration constant for each game. The Othello's UCT exploration constant converged to the value 0.780941 and the Nine Men's Morris's constant converged to the value 0.587104, reinforcing that different games lead to different optimal exploration constant values. These exploration constant values were used in all following phases of the UCT-CCNN process.

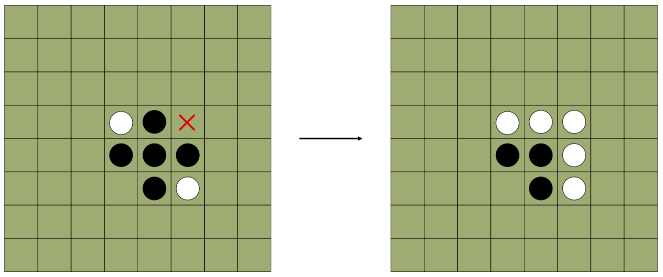
Figure 5: A white player move in the Othello game.

Consider a two-player board game match with an average of J total player moves and the following parameters for the CEM: max number of iterations (i), population size (n), matches executed per example (r), simulation time in seconds for the MCTS (t) and number of threads (h). Therefore, the cross-entropy optimization process will take approximately $\left(i \cdot \frac{n \cdot r \cdot J \cdot t}{h}\right)$ seconds to run, assuming that the number of threads does not exceed the number of matches to be executed during each iteration. The execution of parallel matches is only possible within the same iteration since the result of all of them is necessary to calculate the new distribution parameters for the next iteration. For the Othello game, which has an average of 60 moves per match, the execution time of the CEM phase was of approximately 4.4 days with 64 threads for parallel execution of matches. For the Nine Men's Morris game, which has an average of 50 moves per match, the CEM phase took approximately 3.7 days. As can be seen, the CEM phase is very costly in terms of processing time, and because of this we use small values for the parameters such as population and simulation time for the MCTS algorithm. It was decided to give more emphasis in terms of processing time to the examples generation phase.

For the examples generation phase the parameters were defined as follows: 30 matches executed from the default initial state; 300 random states to be generated; 5 matches executed from each generated random state; 64 threads for parallel execution of matches. Table 1 shows the number of examples generated for each configuration, according to the example filtering rules described in Section 3.2, recalling that in all configurations the number of test examples was defined as 10% of the filtered examples, and the remaining 90% for the training set, separated uniformly at random.

Game	Simulation Time	Training Examples	Test Examples	Exec. Time (days)
Othello	200 s	149.876	16.653	3.32
Othello	600 s	144.219	16.024	9.96
Mill	200 s	167.360	18.596	2.77
Mill	600 s	258.260	28.696	8.30

Table 1: Number of examples generated by each configuration.

The number of generated examples is variable because the number of moves in a match and the actions selected by the MCTS agents are also variable since the examples are extracted from those matches in each player move. Another possibility would be to define the number of examples to be generated instead the number of matches to be executed. For simplicity, the last option was chosen. Also, for simplicity, it was determined that the CCNN are trained until the number of neurons added is equal to the number of neurons in the input layer since in the experiments the network began to present overfitting with fewer neurons. During the training, a copy of the network is saved at each neuron addition, and in the end, the network with lowest MSE (Mean Square Error) is chosen. Whenever the network error is mentioned, it is implied that the error measure referenced is the MSE. The neural network trained for the Othello game with examples generation limited to 200 seconds of simulation is called **Othello:200** and with examples generation limited to 600 seconds the network is called **Othello:600**. The same applies to the Nine Men's Morris game and its neural networks **Mill:200** and **Mill:600**.

As expected, the training error always decreased, but the addition of new neurons in the network, and hence more hidden layers, caused an overfitting in the training set data, increasing the error in the test set from a certain number of added neurons.

While in the game Othello the selected network error decrease from the **Othello:200** to the **Othello:600**, in the Nine Men's Morris the opposite happened. This difference may have been influenced by the estimated utilities for the examples, the number of generated examples and the binary codification of each game. The **Othello:200** and **Mill:200** networks have very close test errors (MSE of 0.015167 for the Othello and 0.015397 for the Mill), as well as the number of examples used in training (149876 for the Othello and 167360 for the Mill), but the number of added neurons is higher for the Othello game (51 for the Othello and 34 for the Mill).

The **Othello:600** and **Mill:600** networks have a greater error difference (0.012592 for the Othello and 0.017419 for the Mill), although the amount of examples in the training set is higher for the game Mill (144219 for the Othello and 258260 for the Mill). The numbers of added neurons remain close to those of the previous networks (47 for the Othello and 37 for the Mill).

Figures 6 and 7 show the MSE evolution in the training set and test set for each addition, for each training configuration, limited until the addition of the hundredth neuron in the network.

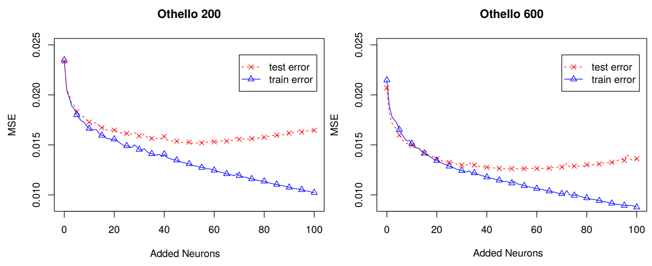

Figure 6: Error evolution in the Othello neural networks.

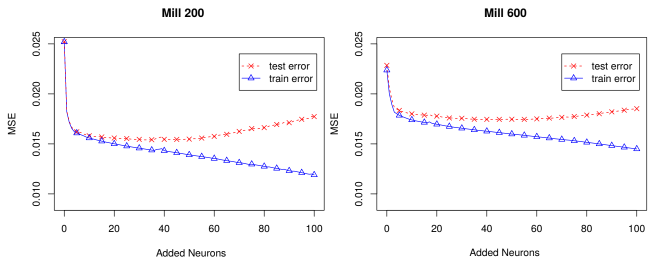

Figure 7: Error evolution in the Nine Men's Morris neural networks.

In the Othello game, no matter what the players' actions, the match will always have a maximum of 60 moves, when the entire board is filled. In the Nine Men's Morris game, the stronger the players are, the more defensive actions are taken leading to a greater number of moves until the game is finished, which may explain the greater number of examples generated from MCTSPlayer agents with greater simulation time. With a larger number of training examples, the function that the neural network tries to approach is better characterized, which for the Nine Men's Morris game has proved to be a more difficult function to learn. The Nine Men's Morris game has four different action types that include the movement of pieces, as well as different game phases that change the rules for the possible actions, which may explain the difficulty in learning the value function. Despite this, it cannot be said that learning Nine Men's Morris is harder than learning Othello, given the differences in the feature modeling and in the number of examples used in the network training.

4.1 Neural network as evaluation function in the Minimax algorithm

In order to evaluate the strength of the four generated value functions, matches were executed between Minimax with Alpha-Beta Pruning agents. One player, called NeuralMinimax, uses the value function generated by the UCT-CCNN process as an evaluation function, and the other player uses an evaluation function developed specifically for the game. In this work, the specific Minimax agents, one for each game, were developed by students in a Artificial Intelligence course. The selected agents were the winners of the competition, roughly the best agents among 12 other competitors.

For each experiment configuration, 100 matches were executed between the NeuralMinimax agent and one of the specific agents. Both specific agents have the Minimax search limited up to three levels. To evaluate different game states and test the generalization capacity of the value function, the NeuralMinimax agents have the search limited from 3 up to 7 for the Othello game and from 3 up to 6 for the Nine Men's Morris game. It is expected that the higher the height of the state in the tree, the better the neural network estimate since in the MCTS-UCT algorithm states next to terminal ones are most visited because each simulation is finalized in a shorter time.

In addition to the variation in the tree height, it was also considered whether the NeuralMinimax agent starts or not the match in the experiment configuration. This is important because in some games the starting player can have a certain advantage. This strategy will be maintained for the game Nine Men's Morris, even though it has already proven that in perfect plays the game always ends in a draw [10], since the evaluation of different states can lead to different results.

Next, in the Tables 2 and 3 are listed the experiment configurations for the game Othello, between the agents NeuralMinimax and the specific one called *Bothello*, with the experiment configurations indicating if the NeuralMinimax agent starts the match and its max search height, as well as the results for the NeuralMinimax agent of the number of wins, losses, draws and the 95% confidence interval for the win rate.

Starts the matches	Height	Wins	Draws	Losses	IC 95% win rate (%)
No	3	100	0	0	(96.4 , 100)
No	4	0	0	100	(0 , 3.6)
No	5	0	0	100	(0 , 3.6)
No	6	0	0	100	(0 , 3.6)
No	7	0	0	100	(0 , 3.6)
Yes	3	0	0	100	(0 , 3.6)
Yes	4	0	0	100	(0 , 3.6)
Yes	5	0	0	100	(0 , 3.6)
Yes	6	100	0	0	(96.4 , 100)
Yes	7	0	0	100	(0 , 3.6)

Table 2: Matches result for NeuralMinimax Othello:200 agent against *Bothello*.

The variation of the maximum search height for the NeuralMinimax Othello:200 agent did not result in improvements in the win rate as was expected. The benefit of previewing a final state in the

Starts the matches	Height	Wins	Draws	Losses	IC 95% win rate (%)
No	3	0	0	100	(0 , 3.6)
No	4	0	0	100	(0 , 3.6)
No	5	57	0	43	(46.7 , 66.9)
No	6	100	0	0	(96.4 , 100)
No	7	100	0	0	(96.4 , 100)
Yes	3	0	0	100	(0 , 3.6)
Yes	4	0	0	100	(0 , 3.6)
Yes	5	0	0	100	(0 , 3.6)
Yes	6	100	0	0	(96.4 , 100)
Yes	7	30	0	70	(21.2 , 40)

Table 3: Matches result for NeuralMinimax Othello:600 agent against *Bothello*.

game tree, during the final moments of a match, also made no difference in the win rate. The reason is that the Othello game tree has an average height of 60 levels, and the advantage of evaluating up to four levels was not enough for the NeuralMinimax agent since the actions performed earlier in the game determined who would win.

The variation in the max search height allowed different states set to be evaluated by the evaluation function, leading to a better understanding of its generalization capacity. Of the ten experiment configurations, the agent NeuralMinimax **Othello:200** won all matches in only two configurations. It can be concluded that the generated value function is not precise although the NeuralMinimax **Othello:200** has won against the opponent *Bothello* with the same max search height. In the matches with the agent NeuralMinimax **Othello:200** there was no draw and the behavior was absolutely deterministic, with players winning or losing all matches in all experiment configurations.

In the case of the NeuralMinimax **Othello:600** agent, of the ten game configurations, the agent won 100% in three configurations, 57% in one and 30% in another. With the increase in the simulation time for the examples generation, the value function accuracy has increased, leading to a better win rate. That increase in the accuracy has a certain tendency to happen in states with greater depth in the tree. Nevertheless, the NeuralMinimax **Othello:600** lost all matches in the configuration with three levels since the generated value function is not yet accurate, even with the improvement in the win rate for the experiment configurations. The results show that it is possible to improve the accuracy of the value function by increasing the simulation time to generate examples for the neural network.

Next, in the Tables 4 and 5 are listed the experiment configurations for the game Nine Men's Morris, between the agents NeuralMinimax and the specific one called *J.A.R.V.I.S*, with the experiment configurations indicating if the NeuralMinimax agent starts the match and its max search height, as well as the results for the NeuralMinimax agent of the number of wins, losses, draws and the 95% confidence interval for the win rate.

Starts the matches	Height	Wins	Draws	Losses	IC 95% win rate (%)
No	3	45	0	55	(35 , 55.3)
No	4	0	17	83	(0 , 3.6)
No	5	0	15	85	(0 , 3.6)
No	6	6	1	93	(2.2 , 12.6)
Yes	3	0	0	100	(0 , 3.6)
Yes	4	5	11	84	(1.6 , 11.3)
Yes	5	5	4	91	(1.6 , 11.3)
Yes	6	0	0	100	(0 , 3.6)

Table 4: Matches result for NeuralMinimax Mill:200 agent against *J.A.R.V.I.S*.

It can also be observed that for the Nine Men's Morris game matches there is a tendency of better win rates with the increase in the simulation time, although the win rate was much worse in comparison to the Othello game. The NeuralMinimax **Mill:200** agent won 45% of the matches in its best configuration result and the NeuralMinimax **Mill:600** agent won 100% of the matches in the same configuration, whose max search height was the same as that of the opponent. In its second best configuration result, the NeuralMinimax **Mill:600** agent did not lose any match, but 35 draws occurred, leading to a 65% win rate. Again, the game Nine Men's Morris appears to be more difficult than the Othello game, as was observed in the neural network training. Another observation is the greater tendency to draws in the Nine Men's Morris game. There was no draw in the Othello game matches.

Starts the matches	Height	Wins	Draws	Losses	IC 95% win rate (%)
No	3	100	0	0	(96.4 , 100)
No	4	0	4	96	(0 , 3.6)
No	5	65	35	0	(54.8 , 74.3)
No	6	0	3	97	(0 , 3.6)
Yes	3	0	0	100	(0 , 3.6)
Yes	4	0	4	96	(0 , 3.6)
Yes	5	0	1	99	(0 , 3.6)
Yes	6	1	9	90	(0 , 5.4)

Table 5: Matches result for NeuralMinimax Mill:600 agent against *J.A.R.V.I.S*.

It is worth mentioning that the win rate is mostly 100% or 0% due to the deterministic behavior of the Minimax algorithm. Randomness only occurs when the evaluation function returns the same value for more than one state and one is chosen uniformly at random. Analyzing the Minimax execution, it is concluded that rarely the evaluation function represented by the neural network returns the same value for different states and the frequency of these repeated values increase for states near the end of the game.

With the results, it can be said that the obtained value functions are not precise, in the sense of predicting the true outcome of a perfect policy, but they are by no means bad since the NeuralMinimax agents were able to win all matches in some experiment configurations and were generated without any domain-specific knowledge. Observing in more detail the utility values attributed by the neural networks, it is noticed that many states during the matches have very close utilities. It can be concluded that the value function obtained does not indicate the best state to go, but rather it "indicates" states likely to be good. It should be mentioned that the same happens with the neural network trained from professional Go players moves in the AlphaGo algorithm [22].

The AlphaGo algorithm's paper says that agents based on neural networks trained from moves made by professional Go players only reached a win rate of 11% against specific agents that use domain-specific knowledge. One difference is that in the current work about 150 thousand examples generated by MCTS-UCT simulations were used, while in AlphaGo about 30 million examples generated by professional players were used. Another difference is that the neural network used in AlphaGo is a Deep Convolutional Neural Network, which has a great ability to detect features, while in the current work CCNN were used, which depends on a good manual feature engineering. From that point of view, the results were satisfactory, given the possible limitations. Worth mentioning that in this work, the neural network was used as an evaluation function in the Minimax with alpha-beta pruning algorithm, which may have helped in the obtained results.

It may be questioned that the specific agents chosen for the experiments are not recognized as strong agents and cannot be used to evaluate the strength of the generated agents by the UCT-CCNN method. Although this statement is correct, the idea of these experiments was to show the ability of the generated value functions against evaluation functions implemented by people with domain specific knowledge. In most of the related works, the generic agents are compared against MCTS-UCT agents. A comparison of this kind will be discussed in the next section.

4.2 Neural network as tree policy in the MCTS algorithm

Based on the AlphaGo algorithm [22], it was decided to apply the obtained value function in the tree policy of the MCTS algorithm in order to take advantage of the network characteristic that indicate the probably better child states and thus redirecting the tree growth toward these states. This strategy was implemented in an agent called NeuralMCTS. In the standard UCT algorithm, states are selected in the MCTS tree using a policy that takes into account the statistical data obtained so far. In the NeuralMCTS the UCT formula was slightly modified, as shown in the following formulas:

$$UCT = S(child) + B(child, parent, uct),$$

$$S = \begin{cases} \frac{Q(child)}{V(child)}, & \text{if } V(child) > 0 \\ 1.0, & \text{otherwise} \end{cases} \text{ and}$$

$$B = \begin{cases} \frac{(C(child)+1)}{2} \cdot uct \cdot \sqrt{\frac{2\cdot(1+\log V(parent))}{V(child)}}, & \text{if } V(child) > 0 \\ \frac{(C(child)+1)}{2}, & \text{otherwise} \end{cases},$$

where $S(child)$ represents the node *child* average utility and $B(child, parent, uct)$ represents an exploration bonus for the node *child*. In the exploration bonus, $C(child)$ is an expected outcome value of the game state associated with the node *child*, within the range $[-1, 1]$, and calculated by a neural network trained by the learning method UCT-CCNN. The purpose of the first factor in the exploration bonus formula is to reduce the exploration bonus to less promising states and represents a previous probability for the state, as with the strategy adopted in the MCTS tree policy used in the AlphaGo algorithm. Also, in the exploration bonus formula, the logarithm result is increased by one to prevent the exploration bonus from being nil for parent states visited only once. Another important difference from the default MCTS-UCT algorithm is that in the NeuralMCTS, when a leaf node is selected by the tree policy, all of its children are expanded. There was no change in the default policy, so action are performed uniformly at random during the simulation phase of the MCTS algorithm.

Experiments were executed with matches between NeuralMCTS agents and the default MCTS-UCT algorithm, with the number of simulations limited up to 5000 for all agents. For the NeuralMCTS agents, the neural networks **Othello:600** and **Mill:600** were used. A total of 200 matches were executed for each experiment configuration. In the Table 6 each experiment configuration is listed indicating the game and if the NeuralMCTS agent starts the match, as well as the results for the NeuralMCTS agent of the number of wins, losses, draws and the 95% confidence interval for the win rate.

Game	Starts the matches	Wins	Draws	Losses	IC 95% win rate (%)
Othello	No	156	6	38	78 (71.6 , 83.5)
Othello	Yes	154	3	43	77 (70.5 , 82.6)
Mill	No	60	61	79	30 (23.7 , 36.9)
Mill	Yes	31	50	119	15.5 (10.8 , 21.3)

Table 6: NeuralMCTS matches with the networks Othello:600 and Mill:600.

The idea is that the value functions represented by the **Othello:600** and **Mill:600** networks indicate a spectrum of interesting states to be investigated, and the NeuralMCTS agents use those functions to redirect the tree growth towards these states. The NeuralMCTS agent won an average of 78% matches against the default MCTS-UCT for the Othello game, which makes the NeuralMCTS a significantly better agent for the game Othello. However, for the game Nine Men's Morris the NeuralMCTS agent only won 30% of the matches when it did not start the game and 15.5% of the matches when it did start the game, which reinforces that Nine Men's Morris is a problematic game for the UCT-CCNN method.

Given the results, it can be concluded that the value function represented by the **Othello:600** network has a satisfactory accuracy to the point of increasing the win rate by redirecting the tree growth towards the states indicated by the function. The examples used for training the **Mill:600** network seem to lack the accuracy needed to obtain good results. One possible solution is to increase the simulation time for the examples generation or to adjust the binary coding for the game's state in order to improve the features engineering. This is only an initial observation since the strategy of redirecting the tree growth towards interesting states as a consequence of the tree policy modification may not work for any type of game.

Improving the tree policy is no guarantee of success for the MCTS algorithm. The default policy plays the most important role in the MCTS algorithm because it is used to control the simulations that start from the expanded state and update the tree statistics with the sampled outcome. The tree policy only indicates the node to be expanded. After that, it is the default policy's responsibility to evaluate the state in the simulation phase, as explained in the Section 2.1. The most common implementation for the default policy is to uniformly at random select actions until a final state is reached. Improvements in the default policy can help further improve the results, and this may be why the NeuralMCTS agent did not go well in the Nine Men's Morris game.

The strategy implemented in the NeuralMCTS agent is partially inspired by the strategy adopted by the AlphaGo algorithm. In both games Go and Othello, the possible actions are only of the type of piece placement, but the placement and capture rules and board size are different. In the case of the Nine Men's Morris game, in addition to the piece placement, there are other action types like piece movement and piece removal and also there are different game phases that change the game's rules. To identify game properties that work well with specific MCTS improvements is a difficult task. In some situations, selecting tree nodes, either in the tree policy or in the default policy, using a uniform distribution may be better than applying domain-specific knowledge to redirect the selection [15].

5 CONCLUSION

This work proposed the UCT-CCNN method as a general game playing strategy to obtain intelligent agents for two-player generic board games capable of winning against specific agents that use specif-domain knowledge.

To achieve this goal the UCT-CCNN method uses the Monte Carlo Tree Search (MCTS) algorithm with the Upper Confidence Bounds for Tree (UCT) tree policy to generate a database of state-utility examples. From these examples, a Cascade Correlation Neural Network (CCNN) is trained as a value function to evaluate any game state. The generated value function can be easily integrated with any algorithm such as Minimax with Alpha-Beta Pruning and the MCTS itself.

The examples database is generated from multiple matches executed between MCTS-UCT agents. The generated examples are filtered to select only those generated from players who won the match and the repeated examples are removed while keeping the state with the higher number of visits during the MCTS simulations.

The UCT exploration constant used for the examples generation was obtained from a stochastic optimization method called Cross Entropy Method (CEM). That optimization is necessary because the optimal exploration constant for each game is different and there is no obvious way to define that constant.

The CCNN represents a category of constructive neural network architecture capable of adapting itself to the problem being applied. The use of this type of neural network favors the GGP aspect of the UCT-CCNN method, thus avoiding the need for parameters to specify the network architecture such as the number of hidden layers and the number of neurons in each layer.

As shown in the Section 4.1, agents generated by the UCT-CCNN method were able to win against specific agents in some situations in both of Othello and Nine Men's Morris games. The accuracy of the obtained value functions is tied to the accuracy of the estimated utilities for the examples used in the neural network training. The higher the MCTS simulation time the better the accuracy of the estimated utilities. Because of that, the UCT-CCNN method provides the possibility of controling the strength of the agent to be generated. In the experiments, the increase in the simulation time from 200 to 600 seconds greatly improved the generated agent strength for the Othello game. For the Nine Men's Morris game the improvement was also observed, but it was more subtle.

In addition, the experiments show that the value functions can be used in a tree policy to guide the MCTS search because they indicate a spectrum of probably better states, improving significantly the performance compared to the standard UCT strategy, with the same number of simulations, for the Othello game. In the case of Nine Men's Morris, the modified tree policy decreased the agent's performance, indicating that this strategy may not work for some games, or that the strategy needs improvements such as parameterizing the influence of the neural network in the tree policy or even modify the default policy to use the neural network. This behavior of MCTS improvements is already observed in other related works, where certain improvements work well in some games and others do not.

It is worth noting that the value function used in the generated agents benefits from a complete set of game features, which is not possible in conventional MCTS improvements such as RAVE and FAST [3]. Another UCT-CCNN feature is that the generated agent benefits from the value function as soon as the match starts, which is only possible due the off-line learning process. For GGP agents is usually necessary to accumulate data through the match in order to estimate utility values for features, often greatly simplified. What the UCT-CCNN loses in comparison to these GGP agents is the need to execute an off-line process to generate the agent's value function.

In addition, the UCT-CCNN has another important feature that is the generation of a generic value function, represented by a neural network that can be used in conjunction with any other algorithm. In this work, the value function was used with the Minimax and the MCTS algorithms.

The main feature of the UCT-CCNN method is undoubtedly its general game playing capability, defining a method that is not restricted to a particular game, through the use of powerful tools such as MCTS-UCT and constructive neural networks such as CCNN. The UCT-CCNN is an important step towards the generation of strong intelligent agents, with a completely automated process, allowing game creators to generate intelligent agents with variable difficulty levels without the need for manually implement an AI code, which is often a challenge.

Acknowledgements

The authors would like to thank CAPES, CNPq and Fapemig for their financial support.

References

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

[2] G. Balázs. Cascade-correlation neural networks: A survey. *Department of Computing Science, University of Alberta, Edmonton, Canada*, pages 1–6, 2009.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, et al. A survey of Monte-Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[4] G. Chaslot, M. Winands, J. Uiterwijk, H. Van Den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo tree search. In *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661, 2007.

[5] G. Chaslot, M. H. M. Winands, I. Szita, and H. J. van den Herik. Cross-entropy for Monte-Carlo tree search. *ICGA Journal*, 31(3):145–156, 2008.

[6] T. H. de Mello and R. Y. Rubinstein. Estimation of rare event probabilities using cross-entropy. In *Proceedings of the Winter Simulation Conference*, volume 1, pages 310–319 vol.1, Dec 2002.

[7] S. E. Fahlman. Faster-learning variations on back-propagation: An empirical study. In D. S. Touretzky, G. E. Hinton, and T. J. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 38–51. San Francisco, CA: Morgan Kaufmann, 1989.

[8] S. E. Fahlman and C. Lebiere. Advances in neural information processing systems 2. chapter The Cascade-correlation Learning Architecture, pages 524–532. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[9] H. Finnsson. *Simulation-Based General Game Playing*. PhD thesis, Reykjavik University, 2012.

[10] R. Gasser. Solving nine men's morris. *Computational Intelligence*, 12(1):24–41, 1996.

[11] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[12] S. Gelly and Y. Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, Canada, 2006.

[13] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62, 2005.

[14] C. Igel and M. Hüsken. Improving the Rprop learning algorithm. In *Proceedings of the second international ICSC symposium on neural computation (NC 2000)*, volume 2000, pages 115–121. Citeseer, 2000.

[15] S. James, G. Konidaris, and B. Rosman. An analysis of Monte-Carlo tree search. *AAAI Conference on Artificial Intelligence*, 2017.

[16] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[17] L. Kocsis, C. Szepesvári, and J. Willemson. Improved Monte-Carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1, 2006.

[18] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification, 2008.

[19] S. Nissen. Large scale reinforcement learning using Q-Sarsa(λ) and cascading neural networks. Master's thesis, Department of Computer Science, University of Copenhagen, October 2007.

[20] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The Rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[21] R. Y. Rubinstein and D. P. Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

[22] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[23] G. Yannakakis and J. Togelius. A panorama of artificial and computational intelligence in games. *Computational Intelligence and AI in Games, IEEE Transactions on*, 7(4):317–335, Dec 2015.