Design and Parametric Considerations for Artificial Neural Network Pruning in UCT Game Playing

Clayton Burger Mathys C. du Plessis Charmain B. Cilliers

Department of Computing Sciences
Nelson Mandela Metropolitan University
Port Elizabeth, 6031, PO Box 77000
{clayton.burger2,mc.duplessis,charmain.cilliers}@nmmu.ac.za

ABSTRACT

The Upper Confidence for Trees (UCT) algorithm has been shown to perform well in complex games, but samples undesirable areas of the search space when building its game tree. This paper explores the design and parametric considerations for augmenting the UCT algorithm with an Artificial Neural Network (NN) to dynamically prune the game tree created, thus limiting the game tree created. The expansion phase of UCT is augmented with a trained NN to create a novel UCT-NN variant that includes prior knowledge and strategy. This paper considers the game of Go-Moku for investigating the design and parametric considerations of UCT-NN. The parameters considered are the exploration and exploitation balancing C parameter, the NN training and structural design parameters and the various pruning schemes which could be used in UCT-NN. Parameter tuning techniques are provided for managing the parametric concerns in the proposed algorithm. Results of parameter experiments indicate that a single value of $C = 1.41$ is suitable for the games studied. Suitable values were found for the structural and training parameters of NN, which were required to test various pruning schemes. Of the various pruning schemes considered, an exponentially decaying scheme is found to be superior in the UCT-NN algorithm where a large amount of moves are initially pruned, but fewer moves on deeper ply.

Categories and Subject Descriptors

I.2.1 [**Artificial Intelligence**]: Games; I.2.6 [**Artificial Intelligence**]: Learning; F.1.1 [**Computation by Abstract Devices**]: Models of Computation

General Terms

Algorithms

SAICSIT '13, October 8 - 9, East London, South Africa

http://dx.doi.org/10.1145/2513456.2513477.

Keywords

Neural Networks, Game Theory, Monte-Carlo Tree Search

1. INTRODUCTION

Intelligently playing games has a history of fascinating Artificial Intelligence (AI) researchers [1]. Various traditional approaches were initially developed to simulate cognitive thought processes in games, such as using features with weightings in the game of Checkers [13] and game trees representing series of moves in Chess [16].

Simple techniques, such as the Minimax algorithm, typically work well on games with a smaller search-space or with very simple rules and a limited number of possible moves. For larger games, such as Chess and Go, the traditional approaches do not scale well. Subsequent approaches to game playing have emerged based on various game representation and move selection mechanisms, most notably Monte Carlo Tree Search (MCTS). MCTS is a statistical method for constructing an approximation of a game tree by balancing exploration and exploitation factors through repeated-play penalties.

The class of MCTS variations stems from the Upper Confidence for Trees (UCT) algorithm, originally proposed by Kocsis and Szepesvari [12]. A common problem faced by UCT and by extension, the subsequent MCTS algorithms, is the lack of prior strategic knowledge as the game tree is built only through random simulation and knowledge of the game rules. Traditional game playing algorithms that construct game trees using game-specific knowledge heuristics have been shown to perform well [1], which suggests that UCT may also benefit from a similar improvement.

This study investigates the inclusion of an Artificial Neural Networks (NNs) into the UCT algorithm [6] to act as a pruning agent for removing unnecessary areas of the search space. The game tree constructed by UCT is pruned concurrently to its construction to further force UCT to focus on moves deemed to be more likely to win or to avoid a loss.

Introductory concepts of the UCT algorithm and a brief illustration of how it functions is outlined in Section 2. Various UCT improvements have been investigated by other researchers, many of them considering pruning techniques. The pruning techniques outlined by other researchers, which then form a foundation for NN-based pruning, are discussed in Section 3. The design of the novel UCT-NN algorithm is explained in Section 4 by considering the training, design and role of the NN component. Finally, various practical parametric concerns are raised in the UCT-NN algorithm,

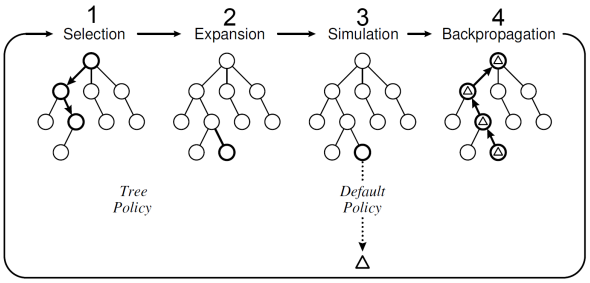

Figure 1: A single iteration of the UCT algorithm, repeated until the specified termination criteria has been met [3]

which are empirically explored using the game of Go-Moku in Section 5.

2. THE UCT ALGORITHM

Traditional game playing algorithms, such as Minimax, typically construct a data structure known as a game tree, which lists each possible move, with its subsequent possible moves as children. The leaf nodes of the tree typically represent games where a win, loss or draw has occurred and no further moves are possible. A full game tree recursively defines every possible permutation of game play, but is difficult to calculate and even more difficult to store in games with large search spaces, such as Chess [1]. The game of Chess, for example, has a game tree consisting of 10^{120} moves, which is intractable to compute and infeasible to store in memory.

The UCT algorithm [11], and its later MCTS variants, aim to approximate the game theoretic distributions of games won from each node in the first ply (game tree level), thus enabling the algorithm to select the node maximising the win likelihood [12]. The algorithm constructs the information in each node by recursively gathering payoff information (number of wins, losses, draws and games sampled) from deeper ply levels. This recursive strategy is applied in four steps (Figure 1) to each search iteration until the stopping condition has been met, such as the computational requirements of the algorithm being depleted [5]. The four steps are *Selection*, *Expansion*, *Simulation* and *Backpropagation*.

1. *Selection*: A recursive tree search is applied to descend through the tree until it reaches a state that is either terminal (an end-game) or has not yet been visited.

2. *Expansion*: If the selected node is not a terminal node, one or more children are added to the current node which correspond to states available from the selected node.

3. *Simulation*: A default policy is applied to the selected node to produce an estimation of the game state by playing a random simulation of two players from the selected node.

4. *Backpropagation*: The result of the simulation from the selected node is propagated to its parents and ancestors until it reaches the root of the tree, updating the recorded statistics in each node that is traversed.

While the terminating condition of the algorithm has not yet been met, the above four steps are sequentially applied and represent a recursive descent to a node that is either a terminal leaf node or unexplored, upon which the node is expanded and a simulation follows. Thereafter the terminal node or simulation result is propagated back to the root as the recursion stack unwinds. This recursive approach is applied in multiple iterations until the computational budget of time or memory provided to the algorithm is exceeded, or after a fixed number of node visits.

On each iteration, a search is conducted to check if the node given (starting from the root) should be explored to the next level if it has children and has thus been previously visited. Alternatively, the node is a terminal node or has been not yet been visited, in which case an appropriate action is applied to allow propagation of the child back to its ancestors, and ultimately the game tree root. The Δ value calculated is the result of a random simulation from an unvisited node or the direct value of a terminal node. After the computational budget is depleted, the algorithm selects the move that is most preferable in the first ply. The preference of a move is typically calculated by weighting each child's wins with the number of visits allocated [14, 5].

The tree policy, or sample deciding equation, that distinguishes UCT apart from other MCTS variants is the UCB1 policy [2], which was adapted [12] as follows:

$$UCT(j) = \bar{\mu}_j + C\sqrt{\frac{2\ln(N)}{N_j}} \quad (1)$$

Where $(j \in (1..K) : UCT(j) \geq UCT(k) \forall k \in K)$ which opts to recursively descend to the child j that maximises UCT, with N being the number of times the parent node (the current node) has been visited, N_j being the number of times the child j has been visited and $C > 0$ is a defined exploration-exploitation tuning parameter. If multiple nodes equally maximise UCT, one is randomly selected [12]. If $N_j = 0$, then a value of ∞ is allocated to the second term, which forces the exploration of previously unexplored children when they are encountered. The $\bar{\mu}_j$ term indicates the average payoff of the child node considered.

3. PRUNING IMPROVEMENTS TO UCT

There are an abundance of variations and enhancements of the UCT algorithm in the traditional game domain of two-player, zero-sum, perfect-information games [3]. Game domains, such as imperfect information games [4] and single player games [15] have also provided interesting variations of UCT which can be applied to other domains, most notably pruning the game-state search space. Pruning the game tree created by UCT can be performed by encouraging UCT to avoid moves by using a penalty or by physically removing them from the game tree.

There are two main categories of pruning enhancements that can benefit from applying *a priori* knowledge to the UCT algorithm, namely *implicit pruning* and *explicit pruning*. Other prior knowledge mechanisms exist, such as changing the default random simulation policy of UCT to use prior games played in the same tournament or against the same player, but these typically do not consider pruning [8, 17, 18].

Implicit pruning can be used in the recursive downward navigation in the selection step of UCT to bias the searching

function to select known strong moves. Explicit pruning removes moves from the game tree to limit the size of the search space. History heuristics and prior information are not typically included in explicit pruning. The question of whether prior knowledge can improve UCT using explicit pruning is considered in this study.

In contrast to implicit pruning, explicit pruning physically removes moves from the search space to restrict the domain in which UCT searches. Instead of being applied in the selection phase, this process is applied in the expansion phase while creating children for nodes.

In the game of Go, a territory heuristic was used by Huang, Liu, Lu and Xiao [10] to remove moves that were known to be weaker depending on the board configuration. An alternative to direct heuristics, He *et al.* [9] found success with an opponent modelling technique through strong experimental results in the game of Go. By creating only nodes that were considered useful based on the opponent's behaviour, the use of pattern recognition essentially constructing a smaller game tree for the opponent [9]. These domain knowledge pruning techniques increase the strength of the UCT player at a cost of hand-crafting or training heuristics, much like hand-crafted evaluation functions in Minimax.

4. A NEURAL NETWORK AUGMENTED UCT PLAYER

NNs have been shown to perform well for pattern recognition and classification problems, as well as games [7]. A feed-forward NN consisting of a single hidden layer and output layer was trained with a back-propagation training rule to learn ranking features from UCT generated patterns. The trained NN is included in UCT to form UCT-NN by applying explicit pruning in the expansion phase of UCT. Explicit pruning is used as it decreases the width of the UCT-NN created game tree by removing weaker moves, allowing the algorithm to exploit better moves.

The structural and training design of the NN is explained in Section 4.1. The use of the trained NN is further explained in the context of being included in UCT as an explicit pruning agent to form the novel UCT-NN algorithm (Section 4.2).

4.1 Neural Network Design

In the UCT-NN algorithm, the NN component is treated as a black-box component where the children of a node that is being expanded are provided to the NN. The NN assigns an output to each possible child, which is used as a ranking. The ranking of children is used to prune a depth-specific number of children. The NN is trained before use in the UCT-NN algorithm, which provides the weight vector corresponding to the NN. UCT-NN takes the trained NN weight vector as an input and uses it for its ranking functions.

There are various physical characteristics of the NN, namely the topology, structure of inputs, structure of outputs and activation functions employed. For simplicity, a feed-forward topology is used with sigmoid activation functions. This topology provides sufficient computational complexity for many pattern recognition and classification problems [7]. The weight vector consisting of the weights between neurons in input value layer, hidden neuron layer and output neuron layer is used when firing the NN to obtain output rankings of moves. Assuming a sufficiently accurate weight vector is obtained, the NN is computationally capable of abstracting the strategic information of the UCT-generated training patterns. In addition to sigmoid functions, other functions were informally tested, namely: linear, step and hyperbolic tangent functions. These functions were found to be ineffective for training, given the game studied and structure of training patterns. The patterns presented to the NN during training and the input data during firing correspond to flattened game boards that are linearised.

The structural design of the NN is the first consideration of the NN component of UCT-NN. The second consideration is the pattern creation and training of the NN to obtain a suitable weight vector prior to using the NN as part of UCT-NN for game playing. A large set of training patterns was generated by employing the UCT algorithm to play against another UCT player with the game of Go-Moku. A Gradient Descent back-propagation rule [7] was used to train the NN.

4.2 Pruning in UCT with a Neural Network

The trained NN is combined into UCT during the expansion phase of UCT, whereby the NN prunes a percentage of the available moves. The percentage pruned must be empirically tuned for the game considered, such as Go-Moku in this study. The number of moves removed from a given node corresponds to the tree depth, as fewer nodes are available on deeper levels as the game board becomes more filled. Various pruning schemes were considered in this study, such as constant and decaying pruning schemes, which are discussed as parameters to the algorithm in Section 5.3.1.

The process of providing a pattern to the NN component during game play is illustrated in Figure 2. When a given node is expanded by UCT-NN during its tree construction, the NN is fired with the same board. Before the NN can be fired, the board is flattened and converted to discrete integer encodings. After the NN has been fired, a symmetric conversion occurs to convert the rankings to a board. The obtained rankings on a board are then sorted to allow the pruning scheme in UCT-NN to prune the lowest ranking moves, subject to the quantity stipulated by the pruning scheme.

The NN related steps in the UCT-NN algorithm are thus: board extraction, board flattening, integer transformation of inputs, NN firing, NN output extraction, output ranking, and child pruning.

- *Board extraction*: When children are to be expanded from an unvisited node in the UCT-NN algorithm, the board and player are extracted from the game node.
- *Board flattening*: The n by n game board is converted to a n^2 sized vector.
- *Integer transformation of outputs*: Each move of the flattened board is encoded by recording a 1 for the current player, -1 for the opponent and 0 for an available move. An NN cannot process textual data, hence the need to transform the flattened board into a discrete-valued vector. By not representing each player with a fixed encoding, but rather using a current player and opponent model, the amount of learning of the NN is reduced by half as the strategies apply to both players.
- *NN firing*: The integer-represented board is presented to the NN and the NN is fired to produce an output.

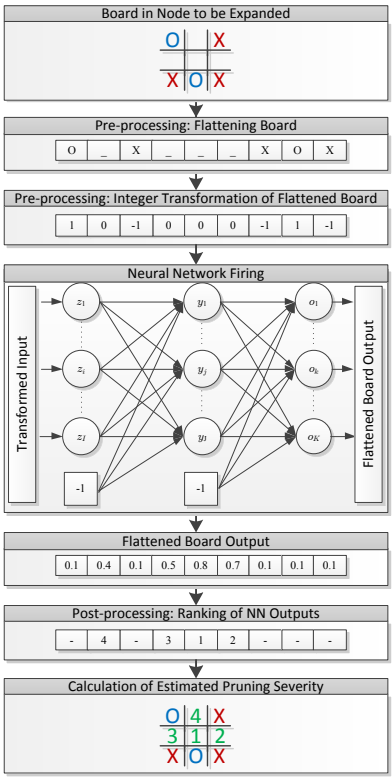

Figure 2: Summary of steps for move suggestion using a neural network with pre-processing and post-processing steps. The green moves in the output calculation represent the NN predicted rankings for pruning.

- *NN output extraction*: The output of the NN is recorded in an n^2 sized vector corresponding to predicted pruning severity, indicating how desirable a move is, as predicted by the NN.
- *Output ranking*: The output vector is converted to a vector containing ordinal rankings of moves with unavailable moves removed, which are specified by an output of 0.1.
- *Child pruning*: The ranked output vector is used in the UCT-NN algorithm. The UCT-NN algorithm provides an integer representing how many moves to prune at the current board tree-depth. The ranked output vector is used to remove the correct number of children nodes.

5. PARAMETRIC EVALUATION

In the design of UCT-NN, three major parameters emerge which must be empirically tuned when applied to a game. These parameters are assumed to be game-specific as only the game of 5 by 5 Go-Moku was considered. The overall performance of the algorithm is not empirically evaluated in this paper, but results from parameter optimisation indicate that the algorithm performs favourably. When investigating parameter values, a scoring system is used to collapse the total wins, losses and draws of a game into a comparative measure. An uneven weighting between wins and losses prevents wins and losses cancelling scores into draws. The score for a player is calculated as:

$$score = 2\#wins + \#draws - \#losses \quad (2)$$

The first parameter, the C parameter controls exploration and exploitation balancing for how the UCT-NN algorithm constructs its game tree (Section 5.1). The second parameter considered is the NN hidden layer size, and associated learning rates when training (Section 5.2). Lastly, the various pruning schemes considered in this study are outlined (Section 5.3). It is assumed that the NN evaluation has a trivial time requirement which is achieved by not considering time as a metric. The benefit of excluding the evaluation costs is that the algorithms can be compared directly. Additionally, the fact that all experimental work was conducted in a heterogeneous cluster of computers further negates the possibility of using time as a common benchmark. All software used was written in the Java programming language as stand-alone experimental tools in conjunction with the CILIB[1] experimental framework.

5.1 Exploration and Exploitation in UCT

The main parameter present in the UCT algorithm is the exploration and exploitation balancing parameter, referred to as C. The C parameter controls the tree building of the UCT algorithm by adapting the default Monte Carlo behaviour with a penalty, or cost term, for repeatedly searching branches. Literature suggests typical values for C while maintaining the fact that the parameter is problem specific [8, 12].

A UCT player is used as the baseline against which the novel UCT-NN algorithm is benchmarked, thus presenting the need to optimise the UCT algorithm for an unbiased comparison. The UCT player is also used to create training data for the NN that is bootstrapped into UCT to create UCT-NN. The C parameter for UCT must thus be empirically tuned to facilitate the creation of realistic training data, and also be tuned for the tree-building behaviour of both the UCT player and the novel UCT-NN player.

By playing two UCT players against each other, each with different C values, to determine the C values for use in subsequent parameter tests and performance evaluations. An experimental design is discussed (Section 5.1.1) to provide quantitative results to support the selection of a suitable C value (Section 5.1.2).

5.1.1 Experimental Procedure

The value of the exploration and exploitation, or C, parameter of UCT is problem dependant and literature suggests that there is no agreed upon method of empirically selecting a suitable value [8]. One of the most common values reported in literature is $C = 1.41$, but empirical means of establishing this value are not reported. In this study, the performance effects of various C values are investigated in an

[1]Computation Intelligence LIBrary available online at www.cilib.net

effort to identify a suitable value for subsequent experiments that use the UCT algorithm.

To measure performance of a game player, an investigation of the frequency of winning, losing and drawing is required against suitable opponents. This study poses two UCT players against each other, each with their own C values in a tournament to identify which C values are linked with higher performance. Each player has an opportunity to play first as well as second, thus off-setting any benefit that playing first or second may yield. Each game between two UCT players is repeated 30 times with time-dependant seeds for their random number generators. Repeating each game 30 times reduces the impact of outlier wins, losses or draws that are not representative of the player's likelihood of winning.

This study uses the game of Go-Moku as a case study for the novel UCT-NN algorithm, which suggests that the value for C should be optimised for the UCT player in this game. It is assumed that the C value for UCT will reflect similar performance in UCT-NN. Varying board sizes of Go-Moku are used in the C selection procedure with the goal of reaching a generalisable value.

In addition to the board size variable, the time allocated to a UCT player has a significant impact on its performance [8]. An additional variable must be introduced as an arbitrary time limit cannot be selected. While a direct time limit may be imposed on UCT, a simpler means of controlling the time limit of the algorithm is by imposing an upper limit on the number of nodes that the algorithm can traverse. Four values for the node visit limit were selected, corresponding to approximately 10 seconds, 1 minute, 10 minutes and 1 hour per move, thus providing a large sample of different time scenarios that UCT would operate in.

Each tournament is repeated for each board size and node visit value identified, providing 12 larger tournaments for each C value. the score for each player is calculated for each set of games in each tournament for each variable configuration, which is then used for evaluation.

The results of each tournament are tabulated to allow for the calculation of overall score totals for each player. In addition to the score totals, the optimal C value for each tournament is identified as both the first and second player, which is then used to create a frequency of maximal performance as a secondary means of identifying suitable C values.

5.1.2 Results and Discussion of Parameter Selection

The optimality frequency measure for each value was calculated to establish how frequently each C performed best in all tournament. Results from the frequency optimality illustrate that playing first gives a distinct boost to the winning frequency. The overall optimum was found to be $C = 2$, which outperformed every other C value in this tournament environment. By comparing the performance of each C value as $P2$, different C values appear to have performed comparatively well to the optimum ($C = 2$). Combining the scores of both $P1$ and $P2$ to calculate a total, the overall optimum ($C = 2$) was evident, whilst the secondary optimum ($C = 1$) appeared to win approximately half as often, but still by a larger margin than the other considered C values.

To further investigate the performance of each candidate C value beyond simply identifying which was optimal for each parameter configuration, the scored performance is added for each C value and these totals are compared (Figure 3). By investigating the performance scores of each C value, the performance of sub-optimal values contributes to the overall score to investigate realistic differences between value options.

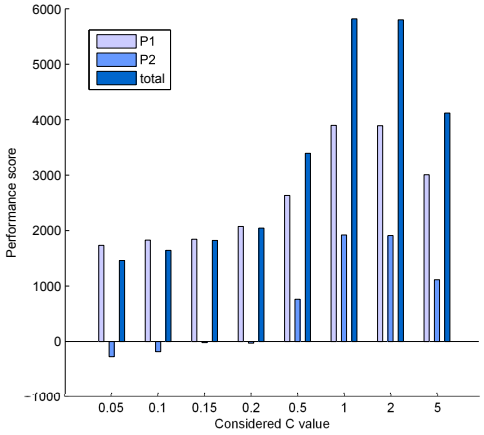

Figure 3: Performance scores of player 1, player 2 and overall total for each considered C value, calculated over each node limit and board size considered.

The computed performance score of each C value (Figure 3) shows that $P1$ consistently outperforms $P2$, which is expected, due to the benefit of playing first in Go-Moku. There is a strong visual difference between the values for $C = 1$ and $C = 2$ and every other considered C value. There was no significant difference between observed total performance scores for $C = 1$ and $C = 2$ ($\chi^2(1, N = 30) = 0.039, P = 0.8433$). This result shows that the frequency of optimality does not reflect overall continuous-valued performance, despite the fact that $C = 2$ appeared to be visually better. The relationship between $C = 1$ or $C = 2$ and each other C value was found to be extremely statistically significant, such as with $C = 5$ ($\chi^2(1, N = 30) = 1049.5, P < 0.0001$). The inconsistency between the frequency of optimality and performance score suggests the need for further consideration.

The combined frequency of optimality and performance scores (Figure 4) shows a disparity between which C value should be considered optimal for the purpose of this study. The values for $C = 1$ and $C = 2$ both show promise and indicate that a value between these two considered values may be optimal. A value of $\sqrt{2} = 1.41$ is theoretically observed to be optimal in the original UCT derivation [12]. Other researchers have also found this to be true in practice [8], which motivates the selection of the value of $C = 1.41$ for the remaining experiments in this paper.

5.2 Neural Network Structure and Training

The UCT-NN algorithm contains an NN consisting of a three-layer, feed-forward topology with sigmoid activation functions. The weights for the NN are trained using gradient-based back-propagation. Two main parameters must be considered for the NN to be effectively trained, namely the learning rate, η, and the size of the hidden layer. As decaying learning rates avoid overshooting and getting trapped in local optima [7], two learning rates are consid-

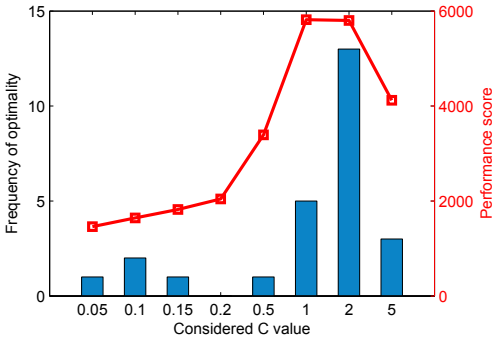

Figure 4: Combined visualisation of optimality frequencies and performance scores for each considered C value, calculated over each node limit and board size considered. The performance score is illustrated by the line chart overlay.

ered, namely: a linearly decaying learning rate (η_{lin}) and an exponentially decaying learning rate (η_{exp}).

When training an NN, the vector of weights, which maps onto the weightings between neurons of successful layers, is optimised. There are two major phases to successfully training an NN, namely:

- *Training data collection and pre-processing:* Training data is gathered through random simulation between two UCT players to capture a sufficiently large number of patterns representing a large portion of the search space for the given scenario. The sizes of these training sets are arbitrarily chosen based on the number of available patterns and are thus 5 815 training patterns and 18 260 generalisation patterns.

- *Weight vector optimisation:* through the use of gradient-descent back-propagation, the vector of weights representing the trained NN is optimised. Key parameters that are introduced in this phase are the learning rate (η), epoch limit for training and the hidden layer size of the trained NN.

The candidate values for the hidden layer size were identified as ten monotonically increasing values. The values identified range from very small (5) to reasonably large (50). The hidden layer size should ideally be as small as possible for training simplicity and speed, but should simultaneously be large enough that the problem described by the training data set can be accurately learnt by the NN [7]. Five candidate values for the learning rate were identified, ranging from 10^{-6} to 10^{-2}. Small learning rate values allow accurate exploitation, but can suffer from premature convergence and thus force the training algorithm to stagnate, thus a large range of values were considered.

The epoch limit, or iteration limit, is set at 3 000 to provide sufficient evidence for parameter optimisation without fully training each NN. With the correct parameter configuration, the epoch limit for actually training the NN is set at 20 000. Higher epoch limits are more favourable for investigating NN training performance, but are directly proportionally correlated to the time taken, thus lower values are selected for experimental feasibility. Stagnation and overfitting detection are not applied when training the NN, but rather observed *a posterior* to select the iteration which exhibits both a low learning error and a low generalisation error. As overfitting was not used as a measure of termination, instead relying on a fixed epoch limit, a training error threshold was not applied.

When optimising the hidden layer and η parameters and training the NN, the metric used is the Mean Squared Error (MSE).

$$MSE = \varepsilon = \frac{\Sigma_{p=0}^{P}\Sigma_{k=0}^{K}(t_j - o_j)^2}{PK} \quad (3)$$

where P is the number of patterns considered, K is the number of neuron outputs, t_j is the j-th target value in the pattern p considered, and o_j is the output of the j-th output of the NN for the pattern p considered.

The MSE for the NN evaluating the training set ($\varepsilon_{training}$) is used in the back-propagation algorithm to identify the error gradient to suitably adjust the weight vector in the direction that minimises $\varepsilon_{training}$. The generalisation MSE is also calculated for the NN as $\varepsilon_{generalisation}$, which is used to identify which epoch overfitting occurred on after training. The standard MSE was adapted in this study to exclude k values corresponding to moves that are already made on the board. As the training data is generated by UCT self-play, the NN is thus encouraged to train to learn features of unmade moves rather than existing moves.

Informal experimentation was conducted to investigate and compare the hidden layer size and learning rate parameters. The hidden layer size of 30 appeared to perform sufficiently well and the 0.01 η value appeared to be the best out of the identified candidate η values. Additionally, when comparing the linear and exponential decay η functions, the exponentially decaying function was found to consistently produce lower errors. By retraining a set of NNs using these parameters, the one that produces the lowest MSE after 15 000 iterations is selected and used in UCT-NN to compare the considered pruning schemes.

5.3 Pruning Schemes for Neural Network Augmented UCT

The UCT-NN algorithm proposes an optimisation to the UCT algorithm by using a trained NN to remove a portion of the game tree, and by extension, the problem search space. The natural question that arises when applying such pruning is: How much of the search space should be removed?

The trained NN in the UCT-NN algorithm can perform rankings of permissible moves in a given game board. By removing the worst performing moves, UCT can concentrate on moves that maximise the chance of winning, thus increasing the likelihood of UCT selecting moves that lie on the principal variation. The proportion of the search space to be removed is considered by identifying various candidate pruning schemes in the studied 5 by 5 Go-Moku game (Section 5.3.1). To experimentally compare the identified pruning schemes, the experimental procedure is outlined (Section 5.3.2). The results of each pruning scheme are presented and discussed to arrive at suitable pruning scheme for use by the UCT-NN algorithm (Section 5.3.3).

5.3.1 Considered Pruning Schemes

In the expansion phase of the UCT-NN algorithm, a portion of the children are pruned from the given node. The pruning scheme used in UCT-NN uses the number of possible children to determine how many children to remove. To select a suitable pruning scheme, various candidate pruning schemes must be identified and empirically compared. Two types of pruning schemes that are considered are constant pruning schemes and decaying pruning schemes.

- *Constant pruning schemes:* Remove a fixed percentage of available children when applying pruning, regardless of the current depth of the game tree.
- *Decaying pruning schemes:* Remove a calculated amount of children based on the current depth of the game tree. Decaying pruning schemes remove a large number of children in the opening game, but have a diminished effect on end-game game trees.

As UCT-NN makes use of explicit pruning, information is lost when children are removed from the game tree, which motivates the need for careful selection of a pruning scheme that does not adversely affect UCT. The phase of the game is an important consideration for how pruning is applied. UCT performs better in end-game scenarios than in opening games [8], which suggests that decaying pruning schemes are superior, as they preserve UCT algorithm's end-game searching.

The constant pruning schemes considered are fixed percentage schemes that remove 25%, 50% and 75% of available children from each parent node. These schemes remove increasingly larger sections of the game tree, which may have a negative effect on UCT-NN. The decaying pruning schemes considered are exponentially and linearly decaying schemes with an initial value of 50%. The exponential decaying scheme removes an initially large section of the game tree, but removes exponentially less children on subsequent ply. The linearly decaying scheme removes exactly one less node than on the previous ply.

The five considered pruning schemes (Figure 5) provide a sufficient spread of the possible levels of pruning severity. The most severe pruning scheme is the 75% constant pruning scheme which removes a very large portion of the game tree, especially when applied to a game with a larger search space. The least severe pruning scheme, by visual comparison, is the exponentially decaying pruning scheme which removes less than 10% of the possible children after the seventh ply. This study empirically tests the effectiveness of each scheme using an experimental procedure described by the following section.

5.3.2 Experimental Procedure

To measure the effectiveness of each considered pruning scheme, a trained NN is used in the UCT-NN algorithm, which is played against a UCT opponent. The trained NN allows the ranking of moves to allow the weakest moves to be pruned, as governed by the pruning scheme.

Tournaments where each player is given the same limit of node visits were used to gather data. This tournament style was used with 60 games with UCT as $P1$ and 60 games with UCT-NN as $P1$. Considered node visit limits are calculated in a doubling scheme from a very low value (610) to a very large value (20 000 000).

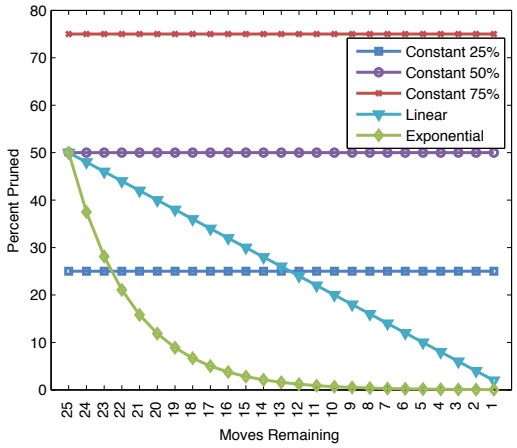

Figure 5: Percentage pruned by each candidate pruning scheme, indicating the percentage of available children removed per ply in a 5 by 5 game of Go-Moku.

The results of each game are totalled to investigate how each player performs at each node visit limit with the UCT-NN playing using the considered pruning scheme. After the results of each game are totalled, the score of each player per node visit limit is calculated. The scored totals are considered for each node visit limit for each pruning scheme to identify regions in which the UCT-NN player, with the considered pruning scheme, outperforms the UCT player.

The overall performance of the UCT-NN player with each pruning scheme is aggregated and compared with a pair-wise two-tailed Fischer's exact test to evaluate which scheme performs pair-wise higher than the other considered schemes. The statistical and visual performance of each pruning scheme is considered to select a suitable scheme for further performance experiments.

5.3.3 Results and Discussion of Parameter Selection

Considering the win-rate of UCT versus UCT-NN, each pruning scheme is considered when used in UCT-NN. In the first considered pruning scheme, a constant 25% scheme, UCT-NN has a higher win-rate only at node visit limits of 2 442 and 4 883, suggesting that the constant 25% scheme performs well only in the initial node visit limit range. The apparent decline in the win-rate of UCT-NN is mirrored by the increase in the number of draws between UCT-NN and UCT. In the second considered pruning scheme, a constant 50% scheme, UCT has a higher win-rate than UCT-NN for each considered node visit limit. A noteworthy trend is the increasing number of draws which matches the decrease in wins for UCT-NN. In the third considered pruning scheme, a constant 75% scheme, the win-rate for UCT is similar to the 50% scheme while the UCT-NN experiences fewer draws. The apparent disparity between the win-rate of UCT-NN and UCT is less pronounced than in the 50% scheme, but does not display any cross-over points such as in the 25% scheme. In the fourth considered pruning scheme, an exponentially decaying scheme, UCT-NN initially wins twice as many games as UCT. The strong initial play of UCT-NN

decreases consistently becoming lower than the win-rate of UCT after a 39 083 node visit limit. The region from 610 to 39 083 shows a large disparity between UCT-NN and UCT, with UCT-NN clearly outperforming UCT. The number of draws increases as the node visit limit increases; a trend that is consistent between each pruning scheme considered. The final considered pruning scheme, a linearly decaying scheme, reflects similar results to the exponentially decaying scheme, but with a sharper increase in draws. UCT-NN initially has a higher win-rate than UCT, which changes from a node visit limit of 19 532. UCT consistently has a higher win-rate than UCT-NN in node visit limits of 39 083 and higher.

By aggregating the number of wins, losses and draws into a score, a fixed value can be allocated to evaluate the performance of a player at a given node visit limit. The scoring of each player is considered relatively to compare performance on each node visit limit and summarised in Figure 6. The first considered pruning scheme, a constant 25% scheme, performed worse against the UCT player except at 1 221 and 2 442 node visit limits. The early improvement of UCT-NN indicates that the constant 25% scheme performs initially well but when more time is made available, UCT exploits the removed information in UCT-NN's game tree. The second considered pruning scheme, a constant 50% scheme, performed worse against the UCT player at each node visit limit. A similar result is found for the constant 75% scheme. The constant 75% scheme appears to perform better in initial node visit limits than the constant 50% scheme, but there is a lack of statistical support to indicate a difference (Table 1).

The two decaying schemes exhibit better performance than the constant schemes at lower node visit limits, both consistently performing statistically better than the UCT opponent. The exponentially decaying scheme has a higher performance score than UCT up to the 78 125 node visit limit, while statistically performing better than the UCT opponent up to the 19 532 node visit limit. The linearly decaying scheme has a lower maximum and also indicates a high score for lower node visit limits, but decays quickly after 9 532. The scores of the five considered pruning schemes are compared visually (Figure 6) and statistically (Table 1).

Visually, the scheme appears to perform better on the highest number of node visit limits is the exponentially decaying scheme. The exponentially decaying scheme performs best on all node visit limits except two limits (9 766 and 1 9532), where the linearly decaying scheme performs only marginally better. A statistical comparison of scores using a pair-wise two-tailed Fischer's exact test reveals that only 11 of the considered 60 tournaments between UCT-NN and UCT are not statistically significant. The exponentially decaying pruning scheme is chosen for the evaluation of the UCT-NN algorithm due to its high scoring against UCT and its stable win-rate for initial node visit limits, which is where the NN is expected to provide a comparable performance boost to UCT-NN against UCT.

6. CONCLUSION AND FUTURE WORK

The inclusion of an NN into UCT to form UCT-NN has the potential to assist UCT in selecting suitable moves when applied to complex games. Various design considerations emerge when forming UCT-NN, namely the algorithm design and the tuning of the various parameters that emerge.

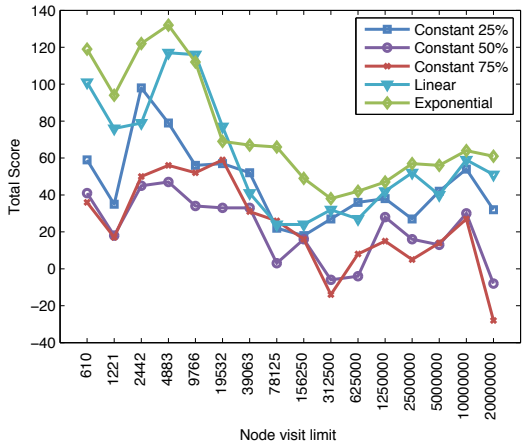

Figure 6: Combined calculated scores for each considered pruning scheme.

The UCT-NN algorithmic design considerations include how the NN is created and used in the new algorithm. By modifying the expansion phase of UCT to include NN-advised pruning, the new algorithm is formulated. This modification includes various pre-processing required by the NN, firing the NN and converting the output into a useful format for UCT to inform its pruning.

To further investigate the design of UCT-NN, its associated parameters must be empirically evaluated using a sample game, such as Go-Moku. A 5 by 5 Go-Moku game was used in this study to elaborate a tournament style optimisation technique for the exploration and exploitation parameter of UCT-NN which controls the tree construction. Secondly, the NN necessitates that the hidden layer size and associated learning rate be investigated to find values that allow suitable learning. The learning of the NN involves providing boot-strapped UCT generated patterns which the NN can extract strategic information from. Lastly, various pruning rates are evident, namely constant and decaying pruning rates. By controlling how much of the game tree is removed in UCT-NN, three constant and two decaying schemes were investigated, revealing that exponentially decaying pruning schemes are superior. Final evaluation of the design of UCT-NN and its associated parameters is left as future work.

Future research is required to investigate the comparative performance of UCT-NN on other games to evaluate the scalability and extensibility of the algorithm. Furthermore, different NN combination techniques can be considered in UCT-NN to introduce prior knowledge at other phases for other purposes than direct pruning.

7. REFERENCES

[1] L. V. Allis. Searching for Solutions in Games and Artificial Intelligence. PhD Thesis, 1994.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47:235–256, 2002.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener,

Table 1: Comparative performance of UCT-NN against UCT for the five considered pruning schemes. Arrows indicate whether UCT-NN had a lower or higher score than UCT for the given node visit limit. Single asterisks indicate statistical significant while double asterisks indicate extreme significance.

Node visit limit	Constant 25%	Constant 50%	Constant 75%	Exponential	Linear
610	↓ 0.8335	↓ **<0.0001****	↓ **<0.0001****	↑ **<0.0001****	↑ **<0.0001****
1221	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↑ **<0.0001****	↑ **0.0023***
2442	↑ **<0.0001****	↓ **<0.0001****	↓ **0.0353***	↑ **<0.0001****	↑ **0.0002****
4883	↑ **0.0002****	↓ **0.0061***	↓ 0.4004	↑ **<0.0001****	↑ **<0.0001****
9766	↓ 0.2513	↓ **<0.0001****	↓ 0.0745	↑ **<0.0001****	↑ **<0.0001****
19532	↓ 0.403	↓ **<0.0001****	↓ 0.8335	↑ 0.1431	↑ **0.018***
39063	↓ **0.0174***	↓ **<0.0001****	↓ **<0.0001****	↑ 0.6059	↓ **<0.0001****
78125	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ 0.8392	↓ **<0.0001****
156250	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **0.001***	↓ **<0.0001****
312500	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****
625000	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****
1250000	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****
2500000	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **0.0031***
5000000	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **0.0412** *	↓ **<0.0001****
10000000	↓ **0.0392***	↓ **<0.0001****	↓ **<0.0001****	↓ 0.8377	↓ 0.0849
20000000	↓ **<0.0001****	↓ **<0.0001****	↓ **<0.0001****	↓ **0.0102***	↓ **<0.0001****

D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, Mar. 2012.

[4] T. Cazenave. A Phantom Go Program. In *Proc. Adv. Comput. Games*, pages 120–125, 2005.

[5] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *Proc. Artif. Intell. Interact. Digital Entert. Conf*, pages 216–217, 2008.

[6] M. C. du Plessis. A hybrid neural network and Minimax algorithm for zero-sum games. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '09, pages 54–59, New York, NY, USA, 2009. ACM.

[7] A. P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley Publishing, 2nd edition, 2007.

[8] S. Gelly, L. Kocsis, and M. Schoenauer. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, 2012.

[9] S. He, Y. Wang, F. Xie, J. Meng, H. Chen, S. Luo, Z. Liu, and Q. Zhu. Game Player Strategy Pattern Recognition and How UCT Algorithms Apply Pre-knowledge of Player's Strategy to Improve Opponent AI. *Proc. 2008 Int. Conf. Comput. Intell. Model. Control Automat.*, pages 1177–1181, 2008.

[10] J. Huang, Z. Liu, B. Lu, and F. Xiao. Pruning in UCT Algorithm. In *Proceedings of the 2010 International Conference on Technologies and Applications of Artificial Intelligence*, TAAI '10, pages 177–181, Washington, DC, USA, 2010. IEEE Computer Society.

[11] L. Kocsis and C. Szepesv. Bandit based Monte-Carlo Planning. In *Euro. Conf. Mach. Learn*, pages 282–293, 2006.

[12] L. Kocsis and C. Szepesvári. Universal parameter optimisation in games based on SPSA. *Mach. Learn.*, 63(3):249–286, June 2006.

[13] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[14] F. C. Schadd. *Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis*. M.sc, Maastricht University, Netherlands, 2009.

[15] M. P. D. Schadd, M. H. M. Winands, H. J. V. D. Herik, G. M. J.-b. Chaslot, and J. W. H. M. Uiterwijk. Single-Player Monte-Carlo Tree Search. 2008.

[16] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.

[17] D. Silver and G. Tesauro. Monte-Carlo simulation balancing. *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pages 1–8, 2009.

[18] M. Winands and Y. Björnsson. Evaluation function based monte-carlo LOA. *Advances in Computer Games*, 2010.