

Learning Non-Random Moves for Playing Othello: Improving Monte Carlo Tree Search

David Robles

Philipp Rohlfshagen

Simon M. Lucas

Abstract—Monte Carlo Tree Search (MCTS) with an appropriate tree policy may be used to approximate a minimax tree for games such as GO, where a state value function cannot be formulated easily: recent MCTS algorithms successfully combine Upper Confidence Bounds for Trees with Monte Carlo (MC) simulations to incrementally refine estimates on the game-theoretic values of the game’s states. Although a game-specific value function is not required for this approach, significant improvements in performance may be achieved by derandomising the MC simulations using domain-specific knowledge. However, recent results suggest that the choice of a non-uniformly random default policy is non-trivial and may often lead to unexpected outcomes.

In this paper we employ Temporal Difference Learning (TDL) as a general approach to the integration of domain-specific knowledge in MCTS and subsequently study its impact on the algorithm’s performance. In particular, TDL is used to learn a linear function approximator that is used as an a priori bias to the move selection in the algorithm’s default policy; the function approximator is also used to bias the values of the nodes in the tree directly. The goal of this work is to determine whether such a simplistic approach can be used to improve the performance of MCTS for the well-known board game OTHELLO. The analysis of the results highlights the broader conclusions that may be drawn with respect to non-random default policies in general.

I. INTRODUCTION

Classical approaches to two-player zero-sum games of perfect information construct, for every move to be taken by the computer, a minimax game tree up to a certain depth, using a game-specific value function to evaluate leaf nodes. These values are subsequently back-propagated and the most promising move from the root may subsequently be chosen to advance the game. This technique, in conjunction with $\alpha\beta$ -pruning, has been applied successfully to a variety of games such as CHESS [8] and CHECKERS [22]. However, the performance of minimax depends significantly on the quality of the value function and in many games, such as GO, a useful value function is difficult or even impossible to construct. This in turn may prevent the construction of a (non-exhaustive) minimax game tree.

Monte Carlo Tree Search (MCTS) is a related tree search algorithm that overcomes this limitation by substituting the game-specific value function with Monte Carlo simulations: whenever a leaf node is encountered, the remainder of the game is simulated by choosing actions for both players uniformly at random until a terminal state has been reached

(default policy). A new node is subsequently added to the tree and the value of the terminal state is back-propagated in the traditional minimax fashion. The success of MCTS is largely due to the selection of previously visited nodes (tree policy): Upper Confidence Bound for Trees (UCT; [16]) encompasses the principles of Upper Confidence Bounds [2] as used in classical bandit problems to appropriately select previously visited nodes. UCT may be used to strike a proper balance between exploration and exploitation and allows the algorithm to construct an asymmetric tree that emphasises the more promising parts of the extensive game tree.

MCTS (section II-B) provides a very simple framework that is generally applicable to a wide variety of board games where it is difficult to estimate the value of a game state by inspection alone. Indeed, MCTS led to a breakthrough in the game GO where reliable value functions proved difficult to construct. Nevertheless, the performance of MCTS may often be improved significantly by increasing the reliability of the MC simulations using domain-specific knowledge to bias move selection. Indeed, all top-performing GO programs make use of (hand-crafted) default policies that make use of local patterns (partial knowledge) to bias the MC simulations [12], [25], [11]. A significant amount of effort has subsequently been devoted to the design and improvement of game-specific default policies. However, numerous studies have pointed out that, contrary to initial beliefs (e.g., [12]), the objective quality of the default policy is often insufficient to guarantee improvements in the performance of MCTS [10], [11]; improvements are thus often achieved by extensive trial and error, making these developments highly specific to particular games.

In this paper we attempt to shed further light on whether a general approach based on temporal difference learning (TDL) may be used to construct useful default policies for the well-known board game OTHELLO. Learning provides a viable alternative to hand-crafted heuristics and is not restricted to a particular game. It thus maintains the generality of MCTS and it is not surprising to note that learning has played a significant role in MCTS in the past (see section IV). In particular, TDL has proven itself particularly useful in the field of General Game Playing (GGP) [4] where the use of hand-crafted knowledge is impossible.

This paper addresses two issues: whether it is possible to learn a useful heuristic for OTHELLO in a simplistic manner, and whether this knowledge may be used effectively to improve the performance of MCTS. We extend this analysis beyond the default policy and also consider the case where the heuristic is used to assign initial values to the nodes in the

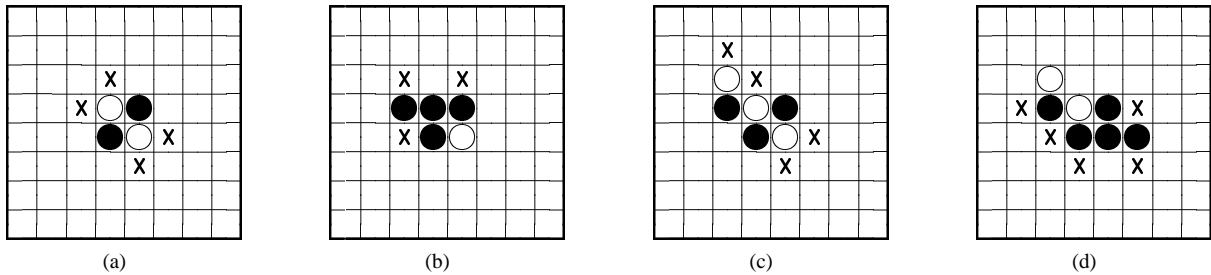


Fig. 1. Game-playing example: (a) The initial Othello board. Black plays first, and has four legal moves; (b) Black opened with move D3 and flipped D4. White is next, and has three legal moves; (c) White played C3 and flipped D4. Black is next and has four legal moves; (d) Black played E6 and flipped E5. White is next and has five legal moves.

tree (i.e., biasing the tree policy). The third scenario considers a bias in both the default policy and the tree policy, and finally we consider the case where the function approximator is adapted further while the games are played to allow for an improved opponent modelling. We subsequently assess how the combination of prior knowledge and sample-based search performs. It should be noted that the goal of this study is not an attempt to design the best possible algorithm for the game of OTHELLO. Instead, the emphasis is on the relative differences in performance of the various MCTS variants considered.

The remainder of this paper is structured as follows: first we introduce the concept of games, including a description of minimax and MCTS in section II. The role played by the tree policy and the default policy is discussed in section III. In section IV, temporal difference learning is introduced, including an overview of previous work that integrates reinforcement learning techniques into MCTS. The details of the empirical study are given in section V, followed by the presentation and analysis of results in section VI. Finally, the paper is concluded in section VII where we also consider future work.

II. PLAYING GAMES

A. Games

A game is essentially a set of well-defined rules that allows one or more players to interact with one another, often by means of manipulating a board. In this paper we consider two-player turn-taking zero-sum games of perfect information: both players have complete access to the game’s state at any moment in time, and one player’s gain equals the other player’s loss. Each game starts in state s_1 and progresses over time $t = 1, 2, \dots$ until some terminal state s_T has been reached. Each player, p_1 and p_2 , takes an action (i.e., makes a move) $a \in \mathcal{A}(s_t)$ that leads to the next state $s_{t+1} = a(s_t)$; a terminal state s_T corresponds to a state where $\mathcal{A}(s_T) = \emptyset$. Each player receives a reward $r(s_T, p_i)$ that assigns a value to the player’s performance. This reward is often equivalent to a *win*, *draw* or *loss*. In order to play the game, a player must have a strategy, known as a *policy*. A policy $\pi(s, a) = p(a | s)$ is the probability of selecting action a given state s ; the value function $Q^\pi(s, a)$ corresponds to the expected outcome of a self-play game that starts with

move a in state s , and then follows policy π . Similarly, the value of a state $V^\pi(s)$ is the state’s utility given policy π .

The game we focus on in this paper is OTHELLO, a well-known two-player game played on an 8×8 board. The initial board is shown in Figure 1: one player is Black, π_B , the other White, π_W ; Black always plays first. Each cell of the board can be in any of the three states (empty, Black, White). The game is played with pieces (stones) that are black on one side and white on the other. At each turn, t , the player whose turn it is selects one of the available legal moves. A legal move is one where the player surrounds one or more of the opponent’s stones in a continuous line, either horizontally, vertically or diagonally (Figure 1). This causes the trapped opponent pieces to be inverted, increasing the player’s piece count. If no legal moves are available, the player must pass. The game ends when neither player has a legal move (e.g., when the board is full).

The standard approach to such games is the minimax algorithm with $\alpha\beta$ -pruning: a partial game tree is constructed up to a depth d in a depth-first fashion. A value function is used at the leaf nodes to determine their game-theoretic utility. This value is back-propagated up the tree such that one player attempts to maximise it, the other player aims to minimise it. As the game is zero-sum, maximising one’s return is equivalent to minimising the opponent’s return (c.f., *negamax*). Nodes in the tree that have sub-optimal values with respect to other nodes in the tree already traversed are pruned. The minimax algorithm forms the backbone of almost all *classical* AI players, including, for instance, DEEPBLUE [8], CHINOOK [22] and LOGISTELLO [5].

Although the minimax algorithm is able to produce optimal play (Nash equilibrium), its performance may degrade significantly depending on the quality of the value function used. A significant amount of work has thus focused on constructing more reliable value functions, providing not only improved estimates of a state’s utility but also facilitating more effective pruning techniques to reduce the tree’s branching factor. However, despite the success of this approach in games such as CHESS, games with large branching factors such as GO, where a heuristic is difficult to formulate, remained impossible to solve. The breakthrough came in 2006 with MCTS (see [9], [12]), an approximation of minimax that grows an asymmetric game tree without the

need for a heuristic; this algorithm is discussed next.

B. Monte Carlo Tree Search

Monte Carlo (MC) simulations may be used to assess the value of a game state s_i by repeatedly using self-play to reach a terminal state for which the value is known. In the simplest case, self-play selects actions available to each player uniformly at random. The utility of using random play as an estimate of the expected value of a state was first demonstrated by Abramson [1] (cited in [11]). This approach may be embedded in a game tree. For each play-out from the root, one state is added to the tree and the values of the terminal states may subsequently be back-propagated to the nodes in the tree. The success of this approach was first demonstrated by the GO program CRAZY STONE [9].

Each play-out from the root may subsequently be divided into two stages: a tree policy is used to select amongst the nodes already contained within the tree and a default policy is used for self-play from any leaf nodes encountered that are non-terminal states. The phenomenal success of MCTS in GO is largely due to the Upper Confidence Bound for Tree (UCT; [16]), which employs UCB1 [2] as tree policy (MOGO [12]). The classical UCT algorithm consists of five main components: a tree policy, a default (play-out) policy, back-propagation, tree expansion and move selection. The following explains each of these components.

Starting from some initial state s_i , MCTS builds a partial game tree to determine the next move $a \in \mathcal{A}(s_i)$: for every node in the tree (initially, only s_i is in the tree), one of the following takes place. If all children of that node are already included in the tree, the *tree policy* is applied to choose one of the children. The most common choice of tree policy is UCB1 [2]:

$$\pi_{UCB1}(s_i) = \arg \max_{a \in \mathcal{A}(s_i)} \left\{ Q(s_i, a) + c \sqrt{\frac{\ln N(s_i)}{N(s_i, a)}} \right\} \quad (1)$$

where $N(s_i)$ is the number of times the parent node has been visited and $N(s_i, a)$ the number of times the child in question has been visited. The two terms in equation 1 correspond to the algorithm’s rates of exploitation (the node’s value) and exploration (taking into account the number of times the child has been visited). At each ply of the tree, the algorithm treats the choice of nodes as a multi-armed bandit problem (see [2], [16]). The exploratory term ensures that each child has a non-zero probability of selection.

If the node selected by equation 1 has children not yet part of the tree, one of those is chosen randomly and added to the tree. It is also possible to add multiple nodes at once or to require a certain number of simulations before a node is added to the tree. The default policy is then used until a terminal state has been reached. In the simplest case, this default policy is uniformly random:

$$\pi_{default}(s_i) = \text{rand}(\mathcal{A}(s_i)), \quad \forall i \quad (2)$$

The value of the terminal state s_T (e.g., win, loss, draw) is then back-propagated until the root of the tree: each node

holds two values, the number of times it has been visited (n_i) and a value v_i that corresponds to the total reward of all play-outs that passed through this state (i.e., an approximation of the node’s game-theoretic value). Every time a node is part of a play-out from the root, its values are updated. After a pre-determined number of iterations, the algorithm terminates and returns the best move found, corresponding to the child of the root with either the highest value or visit count.

III. IMPROVING THE POLICIES

The possibly biggest advantage of UCT is the freedom of domain-specific knowledge: the utility of a state is obtained exclusively from terminal states that have been reached using random play-outs. An appropriate choice of tree policy (e.g., UCB1) attempts to strike a proper balance between exploration and exploitation and allows for the tree to grow asymmetrically; given enough time and memory, UCT is guaranteed to converge to the minimax tree [16]. The lack of domain-specific knowledge makes MCTS simple to implement, highly efficient and generally applicable to a wide range of games. However, the randomness of the MC simulation also appears to be a major hurdle: although the standard algorithm may often produce reasonable performance, it is easily outperformed by variants that implement non-uniformly random simulations. In particular random simulations often suffer from a high variance and if time is limited (which it usually is), convergence may be too slow. Several authors have reasoned about the problems with uniformly random simulations and for some games, their utility has been questioned (e.g., [17]).

Numerous techniques have thus been suggested to improve the speed and/or reliability with which the nodes’ game-theoretic estimates are formed. One approach is to extract additional information from the default policy to populate the values of the nodes in the tree: usually, only the nodes in the tree that are visited by the tree policy are updated. With AMAF [13], and RAVE [11] in particular, the values of any node encountered during self-play are updated. In other words, actions are assigned rewards independent of their context, allowing the rapid accumulation of low-variance (although possibly biased) values for each action.

A second approach is to improve the reliability (i.e., decrease the variance) of the default policy itself by using, for instance, a domain-specific heuristic to guide the otherwise uniformly random move selection¹. This approach has shown to be highly effective (e.g., patterns in GO [12]), yet recent results have shown that, contrary to initial assumptions (e.g., [12]), the integration of heuristic default policies is non-trivial and that the success of this approach is often due to extensive trial and error. In particular, as pointed out by Gelly and Silver [10]:

¹It is important to note that the heuristic need not be complete: unlike algorithms like minimax which require estimates of a state’s value, UCT may utilise partial knowledge to guide the default policy; the state’s value is still determined by the terminal state reached. Nevertheless, it is possible to replace the default policy of UCT altogether if a suitable heuristic is available and still outperform minimax using the same heuristic [17].

It appears that the nature of the simulation policy may be as or more important than its objective performance. Each policy has its own bias, leading it to a particular distribution of episodes during Monte Carlo simulation. If the distribution is skewed towards an objectively unlikely outcome, then the predictive accuracy of the search algorithm may be impaired.

Similarly, Gelly [12] state that it is vital for the default policy to produce meaningful results but that this may not strictly depend on the strength of play. Silver and Tesauro [26] propose the concept of *simulation-balancing* to efficiently learn a simulation policy for Monte Carlo search (also see [15], [15]). The main idea of this approach is to balance the minimax error of the default policy rather than minimising the error directly.

These results have some profound implications. In particular, it appears difficult to judge a priori the benefit of some default policy with regards to the overall performance of the MCTS algorithm. As a result, trial and error plays a vital part in improving the overall performance of the algorithm. However, such use of domain-specific knowledge is problematic: not only do game-specific default policies compromise the generality of UCT, but principled approaches appear limited (apart from simulation balancing [26]) to the integration of this knowledge, largely due to the problems outlined above. A better understanding of the role played by non-random default policies is thus essential and in this paper we address the issue whether a simple function approximator may be used to improve the performance of MCTS in the case of Othello.

IV. TEMPORAL DIFFERENCE LEARNING AND MCTS

A. Temporal Difference Learning

One of the fundamental components of game-playing algorithms is the value function (or static evaluation function). The value function heuristically determines the relative value of a position, i.e., how good or bad a state is from the point of view of one of the players. Crafting a good value function typically requires expert knowledge, but simple value functions may often be computed automatically by using a learning algorithm, such as temporal difference learning. TDL has been very successful in learning or improving game-playing agents in the games of CHESS [3], CHECKERS [23], OTHELLO [6] and, more recently, in Go [25]. Similarly, TDL has been used to improve MCTS in the domain of General Game Playing [4], as well as in numerous other domains, such as in Go itself [10], [25].

TDL is a model-free method for policy evaluation that bootstraps by approximating its current estimate based on previously learned estimates [27], [24]. The simplest TDL algorithm is TD(0). It consists in updating the value function by bootstrapping from the very next time-step:

$$\delta_{t+1} = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \quad (3)$$

$$V(s_{t+1}) = V(s_t) + \alpha \delta_{t+1} \quad (4)$$

where α is the learning rate, and δ is the *temporal difference error*, which is defined as the difference between values of states visited in successive time steps [28]. In other words, rather than waiting until the episode has ended to get the return, in TD(0) the value function of the next state is used to approximate the expected return.

TDL methods are known to work well with *lookup-tables*, but these can only be applied when the number of inputs is relatively small, and an exact representation would generally require storing distinct actions for every possible state, which is computationally impossible for games with large state-space. In order to use TDL with large state-spaces, it is necessary to use *value function approximation*, in which the values of all states and actions are approximated using small number of parameters (weights).

Here we are interested in the simplest possible approach. The weights are trained by TDL from games of self-play to determine the relative contribution of each feature to the expected value. The value function can then be estimated by a function of the features and parameters θ and a very simple and common technique [27] is to use a linear combination of features and parameters to approximate the value function $V(s) = \phi(s) \cdot \theta$.

Although learning requires additional computational steps and thus time (which may be a problem in some games), it offers numerous advantages over the use of (hand-coded) domain-specific knowledge. For instance, a learning algorithm may uncover information missed by the practitioners and may adjust this information as time goes by (e.g., the utility of specific information may vary depending on the state and progression of the game). However, possibly the greatest advantage of making use of learning in MCTS is the preservation of the algorithm's generality: learning may be used in many different different circumstances and hence provides a robust and flexible enhancement.

B. Previous Work

The attributes of TDL listed above make it an excellent tool to improve the performance of MCTS and numerous studies in the past have considered the combination of MCTS and TDL.

Osaki [21] used a new reinforcement learning method, Temporal Difference Learning with Monte Carlo simulation (TDMC), to learn a value function for OTHELLO. They used winning probability as substitute for rewards in non-terminal positions, and outperformed TD(λ) in their experiments. They concluded that maximising the expected total sum of substitute rewards is a successful learning policy for similar games.

With respect to MCTS in OTHELLO, initial studies of its performance were investigated by Hingston [14]. A WPC was used to bias the choice of moves, without the use of expert knowledge. Two versions of a WPC were used, one hand-coded, and one learnt from an evolutionary algorithm which adapts the weights to improve their strength. It was shown that a Monte Carlo approach to OTHELLO is a feasible alternative to classic game-tree search techniques,

and that evolutionary algorithms can be used to improve their strength.

Gelly and Silver [10] combined offline and online value functions in the UCT algorithm for 9 x 9 GO. It was shown that a default policy learnt offline using TD(λ) can be used to complete the play-out once the tree policy reached a leaf node. The authors also demonstrated that initial learning can be boosted using RAVE, and that prior knowledge can be used to initialise the value function, Q_{UCT} , within the UCT tree. When a new state and action (s, a) is added to the UCT tree, $n(s, a)$ is initialised to an estimated *equivalent experience* number, n_{init} , and $Q_{UCT}(s, a)$ takes the value from an existing value function $Q_{prior}(s, a)$. We use this simple method to add offline knowledge in the tree policy, which is explained in Section V.

Baxter [3] learnt the parameters of a value function for the chess program KNIGHTCAP using a variant of TD(λ) called TDLeaf(λ), which is essentially TD(λ) applied to minimax search. The learning process started from a value function in which all the parameters were set to zero except the values of the pieces, and KNIGHTCAP went from a 1650-rated player to a 2150-rated player in just three days and 308 games of online play.

Schaeffer [23] showed that TDL provides an effective solution to learning the weights of a value function used in a high-performance game-playing chess program. They used a linear combination of 23 knowledge-based features for each of the four game phases, which were learnt by TDL using self-play, indicating that a good teacher is not needed for the program to learn a value function that plays at world-championship level.

Finally, Bjornsson and Finnsson [4] utilised a simulation-based approach in General Game Playing instead of traditional game-tree search and won the 2007 and 2008 AAAI GGP competitions. Since it is not possible to use game-specific knowledge in GGP, it showed that MCTS can automatically learn to play a wide variety of games, given only descriptions of the game rules. The work in GGP is one of the reasons we opted for a simple function approximator to retain the generality of the algorithm.

C. Weighted Piece Counter

In this paper we employ TD(0) to learn a simple function approximator $\Phi : S \rightarrow \mathbb{R}$ which approximates the game's value function. It may be used to guide the MC simulations: we make use of ϵ -greedy, which selects the best possible move (according to the function approximator) with probability $1 - \epsilon$ and any other move (uniformly at random) with probability ϵ . The function approximator may also be used to guide the tree policy by initialising a node's value to the estimate of the function approximator. This initial value is subsequently refined by the random play-outs from that node.

In an attempt to keep the integration of TDL as simple as possible, we use a simple weighted piece counter (WPC) as function approximator. In particular, we follow the same approach as presented in [18] and obtain a value for state s

as follows:

$$\Phi(s) = \sum_{i=1}^{8 \times 8} w_i s_i \quad (5)$$

The values used to simulate positions on the board are: +1 (White), -1 (Black) and 0 (empty). The outcome of the game is +1 (win), 0 (draw) and -1 (loss), from White's perspective. The weights of the WPC are updated after each move as follows:

$$w_i := w_i + \alpha[v(x') - v(x)](1 - v(x)^2)s_i \quad (6)$$

where

$$v(s) = \tanh(\Phi(s)) = \frac{2}{1 + \exp(-2(\Phi(s)))} - 1 \quad (7)$$

If a terminal state has been reached, the actual value of the state r is used instead:

$$w_i := w_i + \alpha[r - v(x)](1 - v(x)^2)s_i \quad (8)$$

The weights were adjusted by playing 100,000 games of self-play using an ϵ -greedy policy, with a learning rate $\alpha = 0.001$, $\epsilon = 0.1$ (the probability of making a random move), and no discount factor ($\lambda = 1.0$). Also, α is decremented every 5,000 games by a factor of 0.95.

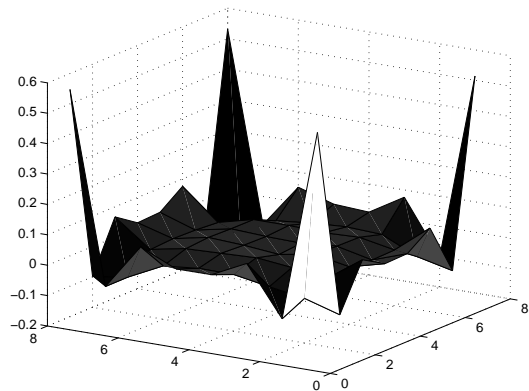


Fig. 2. Symmetric weights of the Weighted Piece Counter: the weights in the corners are positive, each surrounded by a set of three negative weights. Furthermore, the remaining weights along the boundary of the board are positive while the adjacent weights closer to the centre of the board are all negative. Finally, the weights in the centre of the board are both positive and negative and tend to be significantly lower than those found at the extremities.

V. EXPERIMENTS

The experiments were designed to evaluate the difference in performance between MCTS with the WPC in the tree policy, default policy and both. Whenever the WPC is used, the policy is ϵ -greedy with $\epsilon = 0.01$. In the experiments we use it in two different ways: first, it is learnt offline (a priori knowledge) and second, it is re-trained throughout the game using TDL. The latter experiment highlights the adaptation of the Q_{WPC} to each particular opponent.

First we denote the policies and combination of policies used to define the players:

- 1) Random:

$$\pi_R(s) = \text{rand}(\mathcal{A}(s))$$

where $\text{rand}()$ selects a random action $a \in \mathcal{A}(s)$.

- 2) ϵ -greedy:

$$\pi_\epsilon(s) = \begin{cases} \pi_R(s), & \text{if } u() < \epsilon; \\ \arg \max_{a \in \mathcal{A}(s)} Q_{WPC}(s, a), & \text{otherwise;} \end{cases}$$

where $\epsilon \in [0, 1]$ and $u()$ returns a random number drawn from a uniform distribution $\in [0, 1]$.

- 3) Minimax:

$$\pi_{\alpha\beta}(s) = \max_{\pi_B} \min_{\pi_W} Q_{WPC}(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$$

- 4) Monte Carlo, as in [11]:

$$\pi_{MC}(s) = \arg \max_{a \in \mathcal{A}(s)} \left\{ \frac{1}{N(s, a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s, a) z_i \right\} \quad (9)$$

where z_i is the outcome of the i th simulation; $\mathbb{I}_i(s, a)$ is an indicator function returning 1 if action a was selected in state s during the i th simulation, and 0 otherwise; and $N(s, a) = \sum_{i=1}^{N(s)} \mathbb{I}_i(s, a)$ counts the total number of simulations in which action a was selected in state s .

- 5) MCTS policy:

$$MCTS(\pi_T, \pi_D) = \begin{cases} \pi_T, & \text{if } s \in \mathcal{T}; \\ \pi_D, & \text{otherwise;} \end{cases}$$

where \mathcal{T} is the search tree that contains one node corresponding to each state s that has been seen during simulations, π_T is the *tree policy*, and π_D represents the *default policy*. For example, $MCTS(\pi_{UCB1}, \pi_R)$ is the standard UCT algorithm that uses UCB1 (Equation 1) in the tree policy and the random default policy.

- 6) UCT policy:

$$UCT(\pi_D) = MCTS(\pi_{UCB1}, \pi_D)$$

- 7) UCT policy + prior knowledge:

$$UCT(\pi_D, Q_{WPC}) = UCT(\pi_D)$$

where Q_{WPC} is the value function learnt a priori, which is used to initialise the values of new state-action pairs (s, a) added to the UCT tree as follows:

$$\begin{aligned} n(s, a) &\leftarrow n_{init} \\ Q_{UCT}(s, a) &\leftarrow Q_{WPC}(s, a) \end{aligned}$$

where $n(s, a)$ is the number of times action a has been taken in state s , and n_{init} is an estimated *equivalent*

experience number.

- 8) UCB1 policy, π_{UCB1} , as in Equation 1.

It is important to note that, whenever $\arg \max$ is used, it would be $\arg \min$ for a minimising tree level. Based on the previous policies, we defined ten different players:

- 1) Random
- 2) ϵ -greedy
- 3) Minimax(4-ply)
- 4) Monte Carlo
- 5) $UCT(\pi_R)$
- 6) $UCT(\pi_R, Q_{WPC})$
- 7) $UCT(\pi_\epsilon)$
- 8) $UCT(\pi_\epsilon, Q_{WPC})$
- 9) $UCT(\pi_\epsilon) + \text{re-training}$
- 10) $UCT(\pi_\epsilon, Q_{WPC}) + \text{re-training}$

These players were evaluated in a 50-game round-robin tournament. All the players that employ a value function (ϵ -greedy, Minimax and some MCTS variants) used the WPC learnt offline, Q_{WPC} . The sample-based players (MC and MCTS variants) were set to allow 2,000 simulations to calculate the best possible move. The Minimax player ($\alpha\beta$ -pruning with negamax) was set to search up to a depth of 4 plies. The MCTS players use the constant $c = 5$, selected following some limited preliminary testing. The ϵ -greedy policy uses $\epsilon = 0.1$, both when used directly as a player (to test the quality of the WPC), and when used to guide the default policy, such as in $UCT(\pi_\epsilon)$, and $UCT(\pi_\epsilon, Q_{WPC})$. Also, we included a simple Monte Carlo player that evaluates moves by 2,000 simulated games of self-play using the random policy.

For the MCTS players that initialise the Q_{UCT} values with the Q_{WPC} , maximum performance is achieved using an equivalent experience, $n_{init} = 50$, which indicates that Q_{WPC} is worth 50 episodes of MCTS simulations. The last 2 players (9, 10) are variants of the other MCTS players, but in these cases the Q_{WPC} provided is re-trained while MCTS is exposed to the opponents. This is to investigate if it is possible to adapt the weights against a particular opponent to increase performance.

VI. RESULTS

The results of the experiments are shown in Table I. The Random player was used as a lower benchmark and it performed as expected, losing most of the games against the strong players (Minimax, Monte Carlo and variants of MCTS) and winning 22% of the games against the ϵ -greedy policy, which is based on the Q_{WPC} . Interestingly, the Monte Carlo player performed better than Minimax(4-ply) but worse than all the MCTS players. The UCT players using the random default policy ($UCT(\pi_R)$ and $UCT(\pi_R, Q_{WPC})$) were the weakest MCTS players, while the players using the ϵ -greedy policy ($UCT(\pi_\epsilon)$ and $UCT(\pi_\epsilon, Q_{WPC})$) in the default policy were noticeably stronger.

TABLE I
 ROUND ROBIN TOURNAMENT (WINNING RATES OVER 50 GAMES); U STANDS FOR UCT (ABBREVIATED DUE TO LACK OF SPACE) AND U_2 INDICATES RE-TRAINING OF WEIGHTS THROUGHOUT GAME PLAY.

Player	Rnd	ϵ -greedy	Minimax	MC	U(R)	U(R,Q)	U(E)	U(E,Q)	U_2 (E)	U_2 (E,Q)	Avg
Rnd	-	22.0	6.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.8
ϵ-greedy	72.0	-	6.0	0.0	4.0	2.0	0.0	0.0	0.0	0.0	8.2
Minimax	94.0	92.0	-	42.0	46.0	40.0	32.0	14.0	30.0	30.0	45.6
MC	100.0	100.0	54.0	-	48.0	42.0	34.0	22.0	12.0	22.0	46.8
U(R)	100.0	98.0	54.0	50.0	-	50.0	24.0	24.0	16.0	18.0	48.6
U(R,Q)	100.0	98.0	58.0	56.0	48.0	-	24.0	22.0	12.0	14.0	48.0
U(E)	100.0	100.0	68.0	66.0	76.0	70.0	-	54.0	36.0	56.0	69.2
U(E,Q)	100.0	100.0	86.0	78.0	70.0	76.0	46.0	-	38.0	34.0	69.8
U_2(E)	100.0	100.0	68.0	86.0	80.0	86.0	62.0	62.0	-	46.0	77.2
U_2(E,Q)	100.0	100.0	70.0	76.0	82.0	82.0	42.0	64.0	50.0	-	74.4

This result shows that using the WPC in the default policy improves the performance of MCTS irrespective of other modifications. However, the impact of the WPC in the tree policy appears negligible although other choices of Q_{init} may improve the results; however, our limited testing did not reveal a better choice for Q_{init} . The re-training of weights, on the other hand, appears to allow the MCTS players to adapt to a particular opponent and performance was subsequently increased. In particular, the algorithms making use of additional learning (i.e., $UCT(\pi_\epsilon)$ + re-training and $UCT(\pi_\epsilon, Q_{WPC})$ + re-training) show the overall best performance of all algorithms considered.

We evaluated the re-trained weights to identify how their “objective” quality has been affected by the changes. In particular, we tested the ϵ -greedy player (player 1) using the original weights and the re-trained weights from players 9 and 10 to play against player 5: the original weights allowed for a winning rate of 4% whereas the new weights both produced a winning rate of 0%. As the number of games played was relatively small, it is not clear whether these differences are significant. Nevertheless, it appears that the new weights perform, by themselves, worse than the original weights, despite the fact they noticeably improved the performance of MCTS in both cases; this may illustrate a bias in the default policy (c.f., simulation balancing) and warrants additional investigation we intend to undertake in the near future.

VII. CONCLUSIONS

Although a perfect OTHELLO player has yet to be found, the leading OTHELLO programs [5], [6] are already stronger than the best human players. In this paper, we used the game of OTHELLO as a testbed to investigate how to improve the default policy of Monte Carlo Tree Search. Improving the default policy of MCTS has been studied before [10], [26], but it remains a major focus of current research in this area. In this work we investigated the impact of a non-random default policy in the game of OTHELLO, extending on previous work on OTHELLO [14]. In particular, we used Temporal Difference Learning to learn offline the weights of a Value Function Approximation, a Weighted Piece Counter, and to use it later in MCTS. We used it in three different ways: 1) to initialise the value function Q_{UCT} , 2) to guide the default policy, and 3) both.

Experimental results have shown that it is possible to improve the default policy of MCTS in the case of OTHELLO, at least in an intermediate game-playing program that uses a simple value function. The results presented in the paper have numerous important implications. First, we have shown how a very simple linear function approximator, a weighted piece counter (WPC) obtained using TD(0), may be used to improve the performance of MCTS: the WPC may be used to either guide move selection throughout the otherwise uniformly random default policy and/or to initialise the estimates of a node’s game-theoretic value. We found a significant increase in performance using the WPC in the default policy whereas an integration in the tree policy showed no improvements. Furthermore, it was shown how additional learning, while MCTS was exposed to individual opponents, improved the algorithm’s performance even further. This combination of offline and online learning is not new (e.g., [10]) yet demonstrates how a noticeable increase in the algorithm’s winning rate may be achieved in a relatively easy manner. Interestingly, the re-trained weights appear inferior when used in conjunction with the Greedy player (see above) despite the overall improvement in the winning rate of the MCTS players. This seems to imply that learning adjusted the weights to the playing strategy of MCTS, possibly allowing for the partial correction of the inherent bias expressed by the function approximator.

For future work, it is important to see if the default policy can be also improved in a high-performance game-playing program. The value function used (WPC) in this work is very simplistic, and is important to consider a more complex one that uses more effective features, such as *stability*, *mobility*, and *parity* [7], and also more complex function approximators. For the latter, N-Tuple networks [19] are an especially good choice, especially when configured in scanning mode [20] since this enables very efficient evaluation of the value function, which is important when computing a large number of game simulations as required for high performance play with MCTS.

ACKNOWLEDGEMENTS

This work was supported by an EPSRC grant EP/H048588/1 entitled: “UCT for Games and Beyond” and a PhD Scholarship from the National Council of Science and Technology of Mexico (CONACYT).

REFERENCES

- [1] B. Abramson. Expected-Outcome: a General Model of Static Evaluation. *IEEE PAMI*, 12:182 – 193, 1990.
- [2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2):235–256, 2002.
- [3] J. Baxter, A. Tridgell, and L. Weaver. Experiments in Parameter Learning Using Temporal Differences. *International Computer Chess Association Journal*, 21(2):84–99, 1998.
- [4] Y. Björnsson and H. Hilmar. CADIAPlayer: a Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, pages 4–15, 2009.
- [5] M. Buro. LOGISTELLO: A Strong Learning Othello Program. *19th Annual Conference Gesellschaft für Klassifikation e.V.*, pages 1–3, 1995.
- [6] M. Buro. From Simple Features to Sophisticated Evaluation Functions. In *1st International Conference on Computer and Games*, pages 126–145, 1999.
- [7] Michael Buro. The Evolution of Strong Othello Programs. In *Entertainment Computing - Technology and Applications*, pages 81–88, 2003.
- [8] M. Campbell, A. J. Hoane, and F.-H. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.
- [9] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, pages 72–83, 2006.
- [10] 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, 2007.
- [11] S. Gelly and D. Silver. Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence*, pages 1–33, 2011.
- [12] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical report, INRIA, 2006.
- [13] D. P. Helmbold. All-Moves-As-First Heuristics in Monte-Carlo Go. In *Proceedings of the 2009 International Conference on Artificial Intelligence*, pages 605–610, 2009.
- [14] P. Hingston and M. Masek. Experiments with Monte Carlo Othello. In *IEEE Congress on Evolutionary Computation*, pages 4059–4064, 2007.
- [15] S. C. Huang, R. Coulom, and S. S. Lin. Monte-Carlo Simulation Balancing in Practice. *Computers and Games*, pages 81–92, 2011.
- [16] L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.
- [17] R. Lorentz. Amazons Discover Monte-Carlo. *Computers and Games*, pages 13–24, 2008.
- [18] S. M. Lucas and T. P. Runarsson. Temporal difference learning versus co-evolution for acquiring Othello position evaluation. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 52–59, 2006.
- [19] S.M. Lucas. Learning to Play Othello with N-Tuple Systems. *Australian Journal of Intelligent Information Processing*, pages 1–20, 2008.
- [20] S.M. Lucas and K.T. Cho. Fast Convolutional OCR with the Scanning N-tuple Grid. In *International Conference on Document Analysis and Recognition*, pages 799 – 803, 2005.
- [21] Y. Osaki, K. Shibahara, Y. Tajima, and Y. Kotani. An Othello Evaluation Function Based on Temporal Difference Learning using Probability of Winning. In *IEEE Symposium on Computational Intelligence and Games*, 2008.
- [22] J. Schaeffer. *One Jump Ahead: Computer Perfection at Checkers*. Springer Publishing Company, Incorporated, 2nd edition, 2008.
- [23] J. Schaeffer, M. Hlynka, and V. Jussila. Temporal Difference Learning Applied to a High-Performance Game-Playing Program. *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 529–534, 2001.
- [24] D. Silver. *Reinforcement Learning and Simulation-Based Search*. PhD thesis, University of Alberta, 2009.
- [25] D. Silver, R. Sutton, and M. Müller. Reinforcement Learning of Local Shape in the Game of Go. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2007.
- [26] D. Silver and G. Tesauro. Monte-Carlo Simulation Balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1–8, 2009.
- [27] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [28] C. Szepesvári. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010.