Experiments with Monte Carlo Othello

P. Hingston, *Senior Member, IEEE*; M. Masek, *Member, IEEE*

***Abstract*— In this paper, we report on our experiments with using Monte Carlo simulation (specifically the UCT algorithm) as the basis for an Othello playing program. Monte Carlo methods have been used for other games in the past, most recently and notably in successful Go playing programs. We show that Monte Carlo-based players have potential for Othello, and that evolutionary algorithms can be used to improve their strength.**

I. Introduction

STRONG computer opponents in board games are often held up as examples of computing power trumping the analytical power of the human mind. The development of a strong computer player is however strongly dependent on the game. Whilst computer opponents for Chess have been made famous [8], the inability of computer-based Go players to compete with even moderately competent human players highlights the limitations of traditional state-space search algorithms running on current hardware.

Traditional approaches, such as mini-max search, rely on a strong evaluation function, and typically encounter problems in games where the evaluation function in the mid-game is weak, and when there is a large branching factor. In these games, a poor evaluation function leads the computer opponent to misjudge its own position and to fail to predict strong opponent moves. A large branching factor prevents the computer from searching to the end of the game, where the positions can be evaluated precisely (as the result is known).

Recently, Monte Carlo-based approaches have shown promise by generating relatively strong players for small-board (9x9 or 13x13) Go. Most recently Wang and Gelly [16] used a Monte Carlo-based tree search algorithm, the UCT algorithm, to perform initial exploitation versus exploration guidance through previously explored branches near the root of the game tree, before launching Monte Carlo simulations to estimate winning chances for board positions. These estimates can then be used in place of a designed evaluation function. Further to this, they found it beneficial to constrain the Monte Carlo simulation by guiding through moves using local template matching and other Go knowledge. This success leads naturally to the question: what is it that makes Monte Carlo methods so successful for Go, where state-based search methods have largely failed? Is there something particular about the nature of Go, or is it just that the competition in computer Go has not yet become strong enough?

We explore this question in an oblique way by applying the Monte Carlo method to another game favoured by AI researchers, Othello. Unlike the case of Go, very strong Othello players have been developed (see [6] for a survey). In this paper, initial work on a Monte Carlo Othello player is presented. We implement a UCT and Monte Carlo based approach, however instead of local template matching, we evolve a weighted piece counter to guide the Monte Carlo simulations.

II. Background

A. Othello

Othello is a game for two players, played on an 8x8 board using round pieces coloured white on one side and black on the other. The board starts in the initial configuration shown in **Figure 1**.

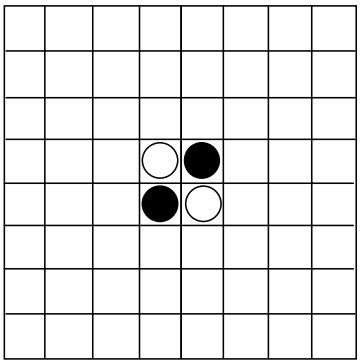

Figure 1 - The initial board state of an Othello game. Players take turns placing new pieces to capture those of their opponent.

The players are assigned a colour and take turns, with each player placing a piece with their colour facing up in order to capture opponents pieces. Only moves that capture opponent's pieces are legal and if no such move exists, the player has to pass. Vertical, horizontal, or diagonal lines of pieces are captured by surrounding them on two opposite sides, with an existing piece and a newly placed piece. Captured pieces, rather than being removed, are flipped over to show the colour of the player that has captured them and may be re-captured by the opposing player. The game ends when neither player has an available move. The winner is the player with the largest number of pieces of their colour.

B. Monte Carlo for games

Monte Carlo methods have long been popular for nondeterministic or incomplete information games, such as poker [1], Scrabble [10], backgammon [15] and bridge [11]. Abramson [1] may have been the first to propose their use in

1-4244-1340-0/07$25.00 ©2007 IEEE

complete information games. The basic idea is to define the value of a position in the game as the expected value of a random game starting from that position, and to estimate this by playing out many games using random move choices. A player then proceeds to use this evaluation function in a 1-ply search, i.e. he/she chooses the move that leads to the highest expected value of the game result. He demonstrated the method with players for 6x6 Othello, tic-tac-toe and chess. Surprisingly, there does not seem to have been any further development on Monte Carlo methods for Othello. More recently, a number of authors ([4][5][7][9][13][16]) have developed Monte Carlo-based players for Go, with very good results. These authors have explored various optimizations and enhancements, including [16] the use of UCT, a bandit-based tree search algorithm, introduced in [14] for solving large state-based Markovian Decision Problems.

C. UCT

The basic Monte Carlo method samples next board positions uniformly in order to choose the one with the best expected value of the game result. This does not make best use of the information obtained while sampling, which could be used to concentrate more effort on more promising positions for each player. This is like a soft version of mini-max. One has to be careful, however, not to neglect less promising positions, which may, on further sampling, start to look better. This is an instance of the classic exploration/exploitation dilemma in learning. In [14], Kocsis et al. introduced the UCT algorithm as a response to this dilemma.

The idea is to consider the choice of which of K next positions to explore as a K-armed bandit problem. A K-armed bandit problem is defined as follows:

Random variables $X_{i,n}$ for $1 \le i \le K$, $n \ge 1$ determine the payoffs for K gambling machines on the n^{th} trial of that machine. The aim is to choose which machine to play at each trial so as to maximize the expected payoff (more precisely, to limit the rate of increase in the difference between the actual payoff and the maximum possible payoff). An algorithm that does this for a large class of such problems is UCB1-tuned [1]:

- *Let $t_{j,n}$ be the number of times machine j is chosen up to time n, and $\bar{X}_{j,n}$ be the average payoff from machine j up to time n. Let $\bar{S}_{j,n}$ be the average squared payoff value from machine j up to time n, and let $V_{j,n} = \bar{S}_{j,n} - \bar{X}_{j,n}^2 + \sqrt{\frac{2\log n}{t_{j,n}}}$.*
- *First play each machine once*
- *Thereafter, play the machine j that maximizes*

$$\bar{X}_{j,n} + \sqrt{\frac{\log n}{t_{j,n}} \min\{1/4, V_{j,n}\}} \quad \cdots(1)$$

Figure 2 - UCB1-tuned

UCT (UCB for tree search) extends UCB1 by combining it with a mini-max tree search. The idea is to consider each node of a search tree as a multi-armed bandit, with one arm for each child node. Starting from the current board position, the method is to use UCB1-tuned to select a child node (a move for the current player), then again to select a child of this child (a move for the opponent), and so on until an unexplored node or a leaf (a completed game) is reached. A random simulated game is then played from this point. The result of the game is used to initialize the average payoff for the new node, and to update the average payoffs for all nodes on the path from the root. This process is repeated as many times as can be within the time or resources allowed for each move, and the final updated average payoffs (which may be thought of as an estimated probability of winning the game) are used to select the move.

```
1   playOneSequence(position)
2   begin
3     positions := <position>
4     while position is unexplored do
5       position := descendByUCB1(position)
6       positions := position + positions
7     od
8     game result := playOneRandomGame(position)
9     updateValue(positions, game result)
10  end of playOneSequence

11  function descendByUCB1(position)
12  begin
13    nextPositions := list of next board positions
14    if some next position is not explored yet then
15      return a random unexplored position
16    else {all next positions have been explored}
17      return a random position that maximises (1)
18    end if
19  end of descendByUCB1

20  updateValue(positions, result)
21    foreach position in positions do
22      increment corresponding t_{j,n}
23      X_{j,n} := X_{j,n} + result
24    od
25  end of updateValue
```

Figure 3 - Pseudo-code for UCT, adapted from [16].

Figure 3 gives pseudo-code for UCT based on that presented in [16]. Note that this is the "parsimonious version", which creates at most one new node in the search tree for each call to *playOneSequence()*. In the standard version of UCT, the test on line 4 becomes a test for the end of the game, and line 8 (which plays a random game to completion without creating new nodes in the search tree) is not required. This variation saves on memory requirements.

Note that on line 13, in the case of a forced pass, the single next board position is an unchanged board with the other player to play. As pointed out in [16], UCT has a number of advantages over traditional state-based search, namely: it is an "anytime" method; it robustly handles uncertainty; and the search tree is explored in an asymmetric way that devotes more attention to promising moves.

UCT for Go was introduced by Wang et al. [16], where they also introduced the idea of improving the effectiveness of the simulation phase, using Go knowledge to bias the choice of moves towards stronger play. Suppose that we are in the process of simulating a game starting from a board position of interest, and have a handful of next moves to choose from. In a purely random simulation, each of these moves is equally likely to be chosen. In [16], moves with more meaning in Go terms are chosen ahead of others, with ties resolved randomly. This results in more meaningful "intelligent simulations". They found that this significantly increased the strength of the Monte Carlo-based player.

D. A Monte Carlo Othello Player

Since Monte Carlo methods have proved so successful in Go, we wanted to investigate the possibility of using UCT as the basis for an Othello player. We found the implementation to be quite straightforward. We decided to use a payoff of 1 for a win, 0 for a loss or draw (draws seem to occur about 2-3% of the time with opponents of approximately equal strength). One tricky point is that a player with no legal move must pass, which does not occur in Go, but this can be handled by treating a pass as a special move.

In order to control the amount of computational effort expended for each move of our player, we decided to limit the number of simulated moves allowed. This makes the computational effort independent of the processing power of the hardware on which it runs.

TABLE 1 - % WINS BY MONTE CARLO PLAYERS AGAINST MINI-MAX PLAYERS

simulated moves per move	3-ply	4-ply
500	39(4.9)	18(3.8)
1000	**47(5)**	43(5)
2000	53(5)	41(4.9)
3000	63(4.8)	41(4.9)
5000	58(4.9)	51(5)
7000	65(4.8)	**60(4.9)**
10000	70(4.6)	57(5)

We tested this player against a mini-max player using weighted piece count as its evaluation function (the weights used were those evolved in [12]). The Monte Carlo player becomes stronger when more simulated moves are allowed per move, and of course mini-max becomes stronger with more search plies. We played Monte Carlo players with 500, 1000, 2000, 3000, 5000, 7000 and 10000 simulated moves per move, against 3- and 4-ply mini-max players. The results are shown in TABLE 1. The figures in parentheses are the standard errors of these measurements. The bolded figures show the point at which the computational effort (in terms of time needed per move) for each method is approximately equal. The winning percentage in each case is not far from 50%, suggesting these players are of approximately equal strength. This is encouraging, but we should not read too much into it. For example, the mini-max player does not use any of the usual optimizations, such as alpha-beta pruning, and we have not attempted to optimise the Monte Carlo player either. Rather, we take these results as an indication that Monte Carlo methods are feasible for Othello.

III. EVOLUTIONARY OPTIMISATION OF SIMULATION BIAS

In order to experiment with optimisation, we decided to try the "intelligent simulation" idea for our Othello player, with the additional twist that we would use an evolutionary algorithm to search for a suitable way to bias the choice of moves. We wanted to bias the choice of moves to achieve a stronger Othello player, but without relying unduly on expert knowledge of what moves constitute meaningful play. We were therefore looking for a simple (and evolvable) method to compute a probability distribution over the available next moves, which we would then use to stochastically select the next move in the simulated game.

Our idea was to base this distribution on a weighted piece count evaluation of the candidate moves, and to evolve the board weights used to calculate the weighted piece count. Given a set of board weights, and a set of candidate moves $\{m_1, m_2, \cdots, m_k\}$, the distribution is calculated as follows:

First, calculate W, the weighted piece count for the current board position. Then calculate W_i, the weighted piece counts for the boards resulting from playing each m_i. Define $D_i = W_i - W$, the weighted piece count difference between the current board and the next board (this can be calculated incrementally). Now define $\min = \min(D_i)$. Use this to calculate $N_i = D_i - \min + 1$. Thus the smallest value of N_i will be 1. Finally, calculate $p_i = N_i / \sum_{j=1}^{k} N_j$. This was used as the probability of selecting m_i. Note that every move has a chance to be selected, as every $p_i > 0$.

To test the effect of this biasing scheme, we created a Monte Carlo player, *MCP(hand),* that uses a hand-coded set of weights from [12], and another that uses an improved evolved set of weights from [12], *MCP(LR).* We played a tournament between these players, plus a Monte Carlo player with random choices, *MCP(random)*; a heuristic player using the evolved weights from [12], *Heuristic*; and a 4-ply mini-max player using weighted piece count with the evolved weights as its evaluation function, *MiniMax*. The Monte Carlo players all used 7000 simulated moves per move, which equates roughly to the same time per move as 4-ply mini-max. Each player played 50 games as black and 50 games as white against each other player. TABLE 2 shows the outcome.

TABLE 2 – PERFORMANCE OF VARIOUS PLAYERS COMPETING IN A ROUND ROBIN OTHELLO TOURNAMENT. *Table entries are number of wins for the row player versus the column player in 100 games. The final entry in each row is the total number of wins for that player. The value in brackets is the standard error.*

	Heuristic	*MCP(random)*	*MCP(hand)*	*MCP(LR)*	*MiniMax(4)*	*Total wins/400*
Heuristic		20(4)	17(3.8)	10(3)	11(3.1)	58(7.04)
MCP(random)	79(4.1)		47(5)	37(4.8)	60(4.9)	223(9.93)
MCP(hand)	81(3.9)	52(5)		34(4.7)	50(5)	217(9.95)
MCP(LR)	86(3.5)	59(4.9)	65(4.8)		45(5)	255(9.61)
MiniMax(4)	86(3.5)	35(4.8)	44(5)	54(5)		219(9.95)

From the table, we see that *MCP(random), MCP(hand)* and *MiniMax* are about equal in strength, while *MCP(LR)* is significantly stronger overall, and about equal to *MiniMax* in head-to-head contests. Curiously, *MCP(random)* seems stronger than *MiniMax* head-to-head, so that *MCP(random), MCP(LR)* and *MiniMax* make up a "rock-paper-scissors" triple. Still, the good relative performance of *MCP(LR)* suggests that stronger play can be achieved by optimizing the bias. An evolutionary algorithm seems a good choice of optimization algorithm for this task.

For this purpose, we chose to follow [12] and use an Evolution Strategy (ES), as the task is basically that of tuning a set of real-values parameters, and computational requirements restricted us to a small population, a combination that ES's usually perform well. In order to reduce the search space, we opted to take advantage of the board symmetries, and restrict the search to sets of board weights of the form:

a	*b*	*c*	*d*	*d*	*c*	*b*	*a*
b	*e*	*f*	*g*	*g*	*f*	*e*	*b*
c	*f*	*h*	*i*	*i*	*h*	*f*	*c*
d	*g*	*i*	*j*	*j*	*i*	*g*	*d*
d	*g*	*i*	*j*	*j*	*i*	*g*	*d*
c	*f*	*h*	*i*	*i*	*h*	*f*	*c*
b	*e*	*f*	*g*	*g*	*f*	*e*	*b*
a	*b*	*c*	*d*	*d*	*c*	*b*	*a*

Figure 4 - Board weights with symmetries.

Therefore, the genome is a sequence of 10 real numbers <*a,b,c,d,e,f,g,h,i,j*>. We used a simple self-adaptive mutation, and no crossover.

Figure 5 displays the ES that we used.

To test the fitness of a set of weights, we played 60 games against *MCP(hand),* (30 as black and 30 as white). The raw fitness value was then calculated as:

$$raw\_fit = games\_won + 0.001 \times total(margin),$$

where $total(margin)$ is the total excess of friendly pieces over opponent pieces at the ends of the games. This was included to provide more gradient information for the optimization algorithm. As this is quite a noisy fitness function, we used a fitness smoothing technique as well as fitness inheritance to convert the raw fitness into a final fitness score using the formula:

$$fit = (count \times prev\_fit + raw\_fit)/(count + 1),$$

where for a parent, $count$ is the number of times it has been evaluated, and $prev\_fit$ is its previously stored fitness, and for new offspring, $count$ is 1 and $prev\_fit$ is the fitness of its parent.

1 *Initialize:*
Set $\tau \leftarrow 1/\sqrt{2\times 10}$. Set $\tau' \leftarrow 1/\sqrt{2\sqrt{10}}$.
Set $\sigma'_{k,j} \leftarrow 1/\sqrt{10}, k = 1\cdots 20, j = 1\ldots 10$.
Set $\omega'_k \leftarrow (1/10)N(0,1), k = 1\cdots 20$.
2 *While more generations do*
3 *For k:= 1 to 20 do*
4 $\sigma_{k,j} \leftarrow \sigma'_j \exp(\tau' N(0,1) + \tau N_j(0,1)), j = 1\ldots 10$
5 $\omega_k \leftarrow \omega'_k + \sigma_k N(0,1)$
6 *Od*
7 *Evaluate fitness of members of* $_k$ *and* $_k$
8 •*select the best 10 from* +
9 *Od*

Figure 5 - Pseudo-code for the ES.

IV. EXPERIMENTAL RESULTS

We ran the ES a total of 10 times to assess the reliability of the algorithm. For each run, we used a population size of 20 and ran the algorithm for 250 generations (testing indicated that more than 250 generations did not improve the strength of the player). Both the evolved players and *MCP(hand)* used only 1000 simulated moves per move. At the end of each generation, we independently re-evaluated the fitness of the fittest individual using 100 games against *MCP(hand)*, for reporting purposes. Figure 6 shows the % of games won by the fittest player in each generation, averaged over the 10 runs, as well as the minimum and maximum over the 10 runs. On average, the fittest player after 250 generations appears to be winning just under 65% of games against *MCP(hand),* which is the winning percentage achieved by *MCP(LR)*. Note, however, that the individual identified as the fittest in each generation may not actually be the fittest, as the fitness function is noisy. So we need to look more closely at these final generations.

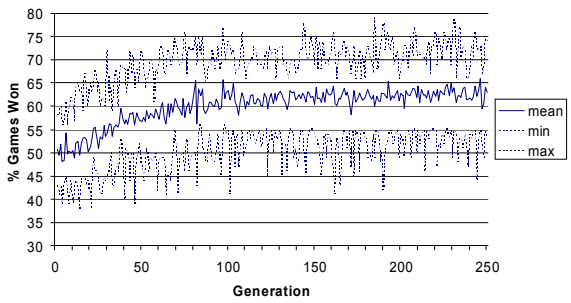

Figure 6 - Summary of best fitness values over 10 runs of the ES.

To do so, we tested the fittest three individuals from each final generation of the 10 runs of the ES, using 1000 games this time for greater precision. The fittest of the three was then designated the champion for that run. The champions for each run had a %wins between 50.6% and 75.9%. We then designated the fittest of these 10 as the overall champion, and used it to create a Monte Carlo player, which we call *MCP(OC)*.

6.21	1.88	12.4	0.37	0.37	12.4	1.88	6.21
1.88	-1.00	-5.45	-1.40	-1.40	-5.45	-1.00	1.88
12.4	-5.45	0.03	0.07	0.07	0.03	-5.45	12.4
0.37	-1.40	0.07	-1.32	-1.32	0.07	-1.40	0.37
0.37	-1.40	0.07	-1.32	-1.32	0.07	-1.40	0.37
12.4	-5.45	0.03	0.07	0.07	0.03	-5.45	12.4
1.88	-1.00	-5.45	-1.40	-1.40	-5.45	-1.00	1.88
6.21	1.88	12.4	0.37	0.37	12.4	1.88	6.21

Figure 7 - Evolved weights for MCP(OC)

Figure 7 shows the weights for *MCP(OC)*. Interestingly, this pattern of weights bears no great resemblance to those evolved in [12], except for the high values assigned to the corner positions. Attempting to use these weights in a weighted piece count heuristic player or a mini-max player based on weighted piece count results in a truly miserable standard of play. Thus the greater strength of *MCP(OC)* cannot be attributed to higher quality simulations. At this point, we don't have a viable theory to explain this.

We now wanted to estimate the strength of *MCP(OC)*. We therefore tested this player against the same set of players as in TABLE 2. For this testing, we increased the number of simulated moves per move to 7000, so that it would be comparable to the earlier results. The results can be seen in the additional row and column of TABLE 3.

From this we see that *MCP(OC)* is very strong against other Monte Carlo players, but is one of the weakest of the Monte Carlo players against *MiniMax*. This may be related to the fact that we used performance against *MCP(hand)* to measure fitness for the ES. A tentative possible explanation is that *MCP(OC)*'s simulations use moves that are representative of the moves that another Monte Carlo player tends to make, but not of the moves that a mini-max player tends to make. In other words, it may be that the ES has evolved an implicit *opponent model*. A co-evolutionary approach, or a fitness measure combining performance against different types of players, may have produced a different result. However, we can conclude that it is feasible to use an evolutionary algorithm to tune simulation bias in a Monte Carlo player.

We also tested *MCP(OC)* by hand against several competent Othello programs on the Internet. From these informal experiments, it has to be said that *MCP(OC)* is not a strong Othello player! For example, it was only able to win an occasional game against Ajax (http://abulmo.club.fr/ajax-en.htm) playing on its "Amateur" level (there are several levels above this), even when we gave the Monte Carlo player several million simulation moves per move (which equates to several seconds of elapsed time per move). Othello programs are sophisticated, and use a lot of high level Othello knowledge, including highly developed evaluation functions, optimized tree searches, opening move libraries etc. The present study at best suggests that it may be possible in the future to develop a strong Monte Carlo-based Othello player. This contrasts with the current relative strength of Monte Carlo-based Go programs. The difference may perhaps be due to the fact that Othello is better understood than Go at this time, and that this understanding has been embedded in existing Othello programs.

TABLE 3 - PERFORMANCE OF VARIOUS PLAYERS COMPETING IN A ROUND ROBIN OTHELLO TOURNAMENT, INCLUDING PLAYERS IN TABLE 2 AS WELL AS MCP(OC). *As in Table 2, entries are number of wins for the row player versus the column player in 100 games. The final entry in each row is the total number of wins for that player. The value in brackets is the standard error.*

	Heuristic	*MCP(random)*	*MCP(hand)*	*MCP(LR)*	***MCP(OC)***	*MiniMax(4)*	*Total wins/500*
Heuristic		20(4)	17(3.8)	10(3)	**10(3)**	11(3.1)	68(7.7)
MCP(random)	79(4.1)		47(5)	37(4.8)	**21(4.1)**	60(4.9)	244(11.2)
MCP(hand)	81(3.9)	52(5)		34(4.7)	**19(3.9)**	50(5)	236(11.2)
MCP(LR)	86(3.5)	59(4.9)	65(4.8)		**29(4.5)**	40(4.9)	284(11.1)
MCP(OC)	**89(3.1)**	**78(4.1)**	**79(4.1)**	**68(4.7)**		**45(5)**	**344(10.4)**
MiniMax(4)	86(3.5)	35(4.8)	44(5)	54(5)	**53(5)**		272(11.1)

Our inexpert observation in these games was that the Monte Carlo player appeared to be doing well until late in the middle game, where it was frequently forced to pass, sometimes several times in succession, at which point it

suffered a great reversal of fortune and many friendly pieces were flipped. Based on our readings on Othello strategy, the Monte Carlo player appears to have a poor appreciation of the concept of "mobility", a well-known Othello feature that has to do with the number of move choices available to each player. (Actually, it seems that weighted piece count is known to be a poor heuristic for Othello. In this study, we used weighted piece count in order to stay close to similar previous studies featuring evolutionary methods.) One possible avenue for future research might be to attempt to use simulation bias based on more effective Othello heuristics. However, this does not really explain why the Monte Carlo-based player fails to foresee its impending doom until it is too late. The heuristic is only used to bias the choice of moves in the simulation, and the final choice of which move to play depends on the estimated "probability of winning", not on weighted piece count. The real problem is that these probability estimates mislead the player. The same problem occurs even for *MPC(random)*, when bias in the simulations is not used. We can only speculate at this point that this may have something to do with the structure of the Othello game tree: perhaps the winning line for the opponent is located within a narrow sub-tree, resulting in a misleading statistically-based probability of winning, which the Monte Carlo player bases its move selection on.

V. Conclusions

In this preliminary study, we have shown that a Monte Carlo approach to Othello is a feasible alternative to traditional state-space search methods. We demonstrated one possible optimisation of the basic UCT algorithm, but there are many other avenues to try, starting, perhaps with some of those that have been successful in Go. As UCT has a number of advantages over state-space search, this may be the right time to revisit Monte Carlo search and investigate its application to other games. Our results suggest some difference between and Go and Othello which needs to be understood. It would be interesting to see what other kinds of games are most suited to a Monte Carlo approach.

Acknowledgment

We would like to thank Simon Lucas for providing Java software for testing Othello players, and Luigi Barone for useful discussions and for executing evolutionary runs for this study.

References

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," Machine Learning, vol. 47, no. 2/3, pp. 235–256, 2002.

[2] Abramson, B. Expected-outcome: A general model of static evaluation. IEEE Trans. Pattern Analysis and Machine Intelligence 12(2):182-193, 1990.

[3] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. Artificial Intelligence, 134:201-240, 2002.

[4] Bouzy, B. The move decision process of Indigo. ICGA Journal, Vol. 26, No. 1, pp. 14–27, 2003.

[5] Bruegmann, B. (1993). Monte Carlo Go. ftp://www.joy.ne.jp/welcome/igs/Go/computer/mcgo.tex.Z.

[6] Buro, M. The evolution of strong Othello programs, in: Entertainment Computing - Technology and Applications, R. Nakatsu and J. Hoshino (ed.), Kluwer, pp. 81-88, 2003.

[7] Cazenave, T., and Helmstetter, B. Combining tactical search and Monte-Carlo in the game of Go. In Symposium on Computational Intelligence and Games, 171-175. IEEE, 2005.

[8] Campbell, M., Hoane, A.J. and Hsu, F.-h. DeepBlue, in Schaeffer, J. and van den Herik, J. (ed.) "Chips Challenging Champions: games, computer and Artificial Intelligence", pp 3-9, Elsevier, Amsterdam, 2002.

[9] R. Coulom, "Efficient selectivity and backup operators in montecarlo tree search," In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy, 2006.

[10] B. Sheppard. World-championship-caliber Scrabble. Artificial Intelligence, 134:241-275, 2002.

[11] Ginsberg, M. L. 1999. GIB: steps toward an expert-level bridge-playing program. In Sixteenth International Joint Conference on Artificial Intelligence, 584-589.

[12] Simon M. Lucas, Thomas Philip Runarsson: Temporal Difference Learning Versus Co-Evolution for Acquiring Othello Position Evaluation. CIG 2006: 52-59, 2006.

[13] Kaminski, P. Vegos home page. http://www.ideanest.com/vegos/, 2003.

[14] L. Kocsis and C. Szepesvari, "Bandit-based monte-carlo planning," ECML'06, 2006.

[15] G. Tesauro and G.R. Galperin. On-line policy improvement using Monte-Carlo search. In M.C. Mozer, M.I. Jordan, and T. Petsche, editors, NIPS 9, pages 1068-1074, 1997.

[16] Y. Wang and S. Gelly. Modification of UCT for Monte-Carlo Go with patterns. In Symposium on Computational Intelligence and Games. IEEE, 2007.