Searching Algorithms in Playing Othello

Zhifei Zhang and Yuechuan Chen
School of Mechanical, Industrial and Manufacturing Engineering
Oregon State University, Corvallis, OR 97331-6001 USA.
Email: {zhanzhif, chenyuec}@onid.orst.edu

***Abstract*—Othello is one of classical games, which can be solved by artificial intelligent methods. It belong to search techniques in artificial intelligence. Many searching algorithms have been raised to solve this problem. In this project, some basic and improved searching algorithms are implemented, including local maximization, minimax, α-β search, UCT. Finally, experiment results are given to show computation cost and winning rate of each searching algorithm.**

I. Introduction

Othello is one of the classical game, whose initial state is shown in Fig. 1. Two players take turn placing pieces—one player white and the other black—on a 8×8 grid board. Every move must capture one or more of the opponent's pieces. In order to capture, player1 should place a piece adjacent to one of player2's pieces, so that there a straight line (horizontal, vertical or diagonal) ending with player1's pieces and connected continuously by player2's pieces. Then the player2' pieces on the line will change to player1's [1].

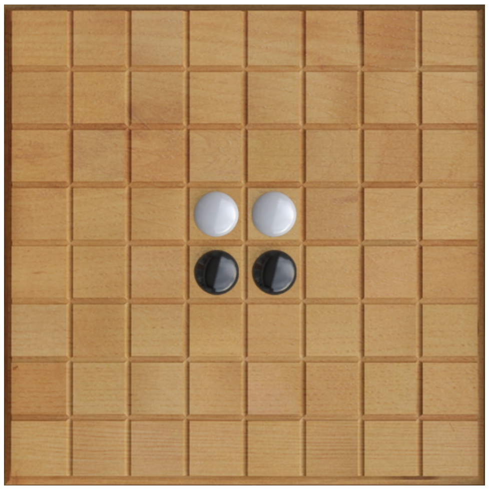

Fig. 1. Initial state of Othello

The final goal is to finish the game with as more as pieces on the board. A final state is shown in Fig.2, in which the black win obviously.

In order to record the state, a 8×8 matrix will be built, and each element in this matrix indicates state of the corresponding location on the chessboard. For example, it is 0 if the grid is empty, 1 if occupied by white piece, and -1 if occupied by black one. In programming, several steps should be involved: 1) checking moves, 2) making moves and 3) switching

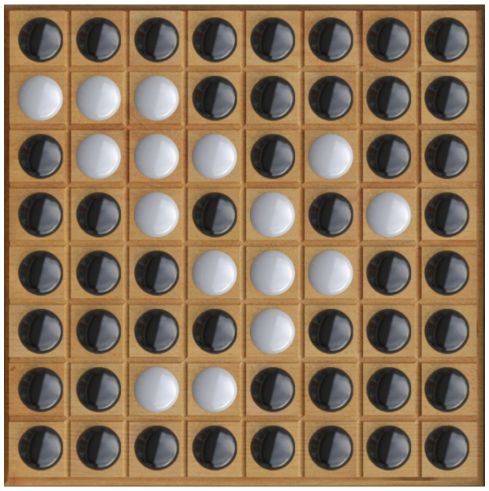

Fig. 2. Final state of Othello

players. Relative playing strategies may be random move (the easiest one), local maximization, minimax search, α-β search [2] [3] [4], NegaScout [5], MTD(f) [6], UCT [8], etc.

II. Implementation

Generally, a board need to be set up and initialized to the initial state as shown in Fig. 1. Then we need to check available moves for current player according to Othello rules. Finally, the most important is how should the play move. This section will only discuss the way to set up board and keep rules. Details about how to move (searching algorithms) will be talked about in section III.

In our project, 8×8 board is used. So, a 10×10 matrix is established the represent the initialized board as shown in Fig. 3. The "?" are boundary of the board. "." are empty squires. "@" and "O" are black and white pieces respectively. From experience, reward of each squires on the board can be set previously, whose detail is shown in Fig. 4. For the long run, we should find a move that can achieve higher reward, as well as trying to suppress opponent's reward. So, a naive searching algorithm is to search all possible moves for further steps and obtain a move with optimal reward.

?	?	?	?	?	?	?	?	?	?
?	.	.	.	.	.	.	.	.	?
?	.	.	.	.	.	.	.	.	?
?	.	.	.	.	.	.	.	.	?
?	.	.	.	@	O	.	.	.	?
?	.	.	.	O	@	.	.	.	?
?	.	.	.	.	.	.	.	.	?
?	.	.	.	.	.	.	.	.	?
?	.	.	.	.	.	.	.	.	?
?	?	?	?	?	?	?	?	?	?

Fig. 3. Representation of board

?	?	?	?	?	?	?	?	?	?
?	120	-20	20	5	5	20	-20	120	?
?	-20	-40	-5	-5	-5	-5	-40	-20	?
?	20	-5	15	3	3	15	-5	20	?
?	5	-5	3	3	3	3	-5	5	?
?	5	-5	3	3	3	3	-5	5	?
?	20	-5	15	3	3	15	-5	20	?
?	-20	-40	-5	-5	-5	-5	-40	-20	?
?	120	-20	20	5	5	20	-20	120	?
?	?	?	?	?	?	?	?	?	?

Fig. 4. Reward of different squires

III. Search Algorithms

A. Random

Random is the most simple algorithm, it just randomly select an available move. So, random strategy can be used as a baseline to evaluate other algorithms.

B. Local Maximization

A more sophisticated strategy evaluate every available moves according to the reward shown in Fig. 4. This consists of getting a list of available moves, applying each one to a copy of the board, and selecting the move with highest reward. Obviously, local maximization algorithm is very short-sighted. Only one further step is considered.

C. Minimax Searching

A player who can consider the implications of a move several turns in advance would have a significant advantage. The minimax algorithm does just that. Fig. 5 shown the main idea of minimax searching. Note A is player1, who will try all available moves and choose the one that obtain the best reward. Note B, C and D is the player2's turn after player1's moving. Player2 always want choose the move that make player1 get the smallest reward. So, minimax searching is to alternatively maximize and minimize the rewards of child notes, during which all available moves (notes) will be evaluated. Obviously, this is a naive and time-consuming searching algorithm.

D. α-β Pruning

α-β pruning can be seen as an improvement of minimax algorithm. It is more efficient since it seeks to decrease the number of nodes evaluated in its search tree. Specifically,

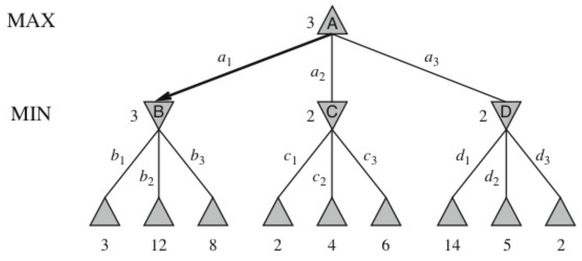

Fig. 5. α-β pruning [9]

it will stop evaluate a move when it has be proved to be worse than a previously examined move. Finally, α-β pruning algorithm returns the same moves as minimax would, but prunes the branches that cannot affect the final decision. Therefore, α-β pruning cost much less time. Fig. 6 shown the basic idea of α-β pruning.

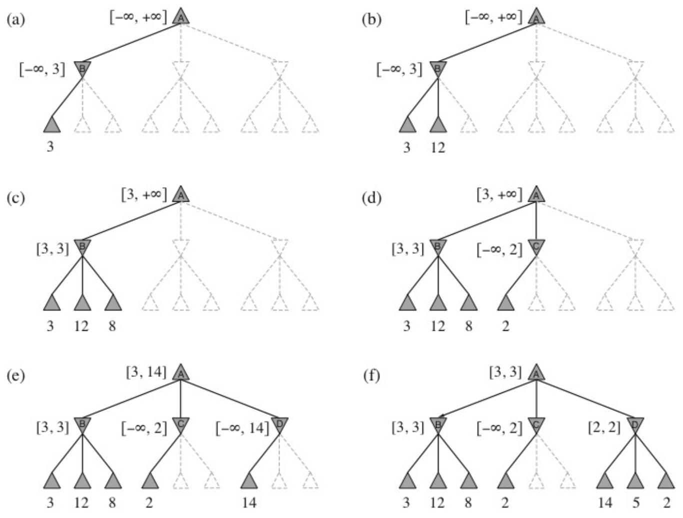

Fig. 6. α-β pruning [9]

In Fig. 6, the triangle is player1 who wants to maximize his reward, and the downward-point triangle is player2 who wants to minimize player1's reward. α is the best already explored reward for player1, and β is the best reward play1 can get under player2's suppression. Now, it's player1's turn (Fig. 6(a)). From note A, player1 chooses a move to note B, then it's player2's turn. Player2 chooses a mowe and make player1 get a reward 3. Then player2 choose another move that makes player1 get a reward 12 (Fig. 6(b)), etc. Player2 always wants to make player1 get smallest reward, so the maximum reward play can get will be not more than 3 if player1 moves from note A to B (Fig. 6(c)). So the value of α can be updated to 3 since current best reward player1 can get is 3. Similarly, player1 tries next available move to note C. The first child note of C rewards 2 (Fig. 6(d)). Since player2 will choose the move that minimize player1's reward, the reward player1 can get from note C will be never more than 2. Thus, note C will never better than note B. According to α-β pruning, we will completely stop search any other child notes of note C.

Then, we go on trying other moves (Fig. 6(e)). Again, when searching to the third child note of D, we can confidently prune note D.

E. UCT

Monte Carlo Tree Search (MCTS) can be considered as assessing the estimated total reward recursively. For every possible states after the initial state, a possible state is included in the tree and the values of the terminal states are back-propagated to the nodes in the tree. Upper Confidence Bound for Tree (UCT) was formalized by Kocsis and Szepesvari in 2006 as an important improvement of UCB [8]. This algorithm is used in the vast majority of current MCTS implementations. Each iteration of UCT may be divided into four steps:

1. Selection: a tree policy is used to select the state which is already in the tree.

2. Expansion: create a possible node according to previous state. If all the children from the previous node are in the tree, create the next generation of exiting nodes.

3. Simulation: a default policy is applied to generate the following possible states until the terminal state has been reached.

4. Back-propagation: update the current estimated value based on simulation results.

MCTS generates a part of game tree from the beginning. If all subsequent states of the node are already in the tree, the algorithm choose one of the children according to UCB:

$$\pi_{UCB}(s) = \arg\max\left(Q(s,a) + c\sqrt{\ln N(s)/N(s,a))}\right) \quad (1)$$

Where N(s) is the number of times the node has been visited and N(s, a) is the number of times the branch from the node has been visited. By this way, each ply of the game tree can be treat the choice of nodes as a multi-armed bandit problem. If the prior node from UCB has children which is not in the tree, one of those children is randomly added to the tree. Then the default policy is used to select actions after leaving tree. In the general UCT case, this default policy is uniformly random.

The value of the final state is back-propagated to the initial state. Two values would be updated by this way: the number of times the nodes has been visited and the estimated value of the nodes. Finally, the algorithm returns the best solution according to the highest value or visit count.

F. ϵ-Greedy UCT

The advantage of UCT is the performance without any specific knowledge about the game. And UCT is guaranteed to converge to the minimax tree when time and memory is enough [8]. However, the drawback of uniform random default policy make general UCT less efficient than the algorithm with more intelligent default policy. So UCT is easily to be improved by modifying the default policy. The only difference between UCT and ϵ-greedy UCT is the default policy. The default policy of ϵ -greedy UCT is ϵ-greedy policy which strike a proper balance between exploration and exploitation:

$$\pi_\epsilon(s) = \begin{cases} \pi_R(s) & u < \epsilon \\ \arg\max\limits_{a \in A(s)} Q_V(s,a) & o.w. \end{cases} \quad (2)$$

where u() is a uniform random number from 0 to 1; and $\pi_R(s)$ is random policy; $Q_V(s,a)$ is the empirical reward as shown in Fig. 4

IV. Performance Analysis

A. Time Complexity

Random and local maximization searching only consider current state, so they search fast and have super low time complexity. But they are not our analyzing points.

Time complexity of minimax is $O(b^d)$, where b is board factor and d is searching depth. Bigger board will lead larger b. Usually, b can be seen as a constant. So computation complexity of minimax will increase exponentially. For α-β pruning, the time complexity will drop to $O(b^{d/2})$. Fig. 7 compares the increasing speed of minimax searching and α-β pruning.

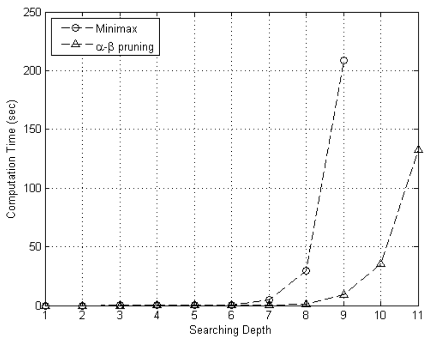

Fig. 7. Computation time with different searching depth

UCT and ϵ-greedy UCT are different from minimax-based algorithms. Since they are based on Monte Carlo tree search, iteration time is the key parameter. More iterations may lead better move, and the time complexity will increase linearly, which is shown in Fig. 8.

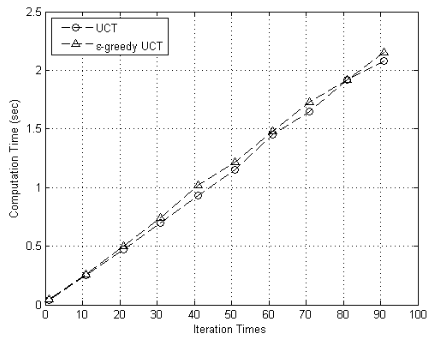

Fig. 8. Computation time with different iterations

B. Success Rate

Success rate is obtained by making two players play Othello using two different searching algorithms. For fair case, two players should use similar ”thinking time”. For example, two player should make a move within 10 seconds. Thus, the player using α-β pruning will searching deeper than the one using minimax searching. Totally, 100 rounds are played, and the results are shown in Table I.

TABLE I
SUCCESS RATE

	Random	Local	Minimax	α-β	UCT	ϵ-UCT
Random	NA	0.19	0.12	0.10	0.20	0.30
Local	0.81	NA	0.00	0.00	0.64	0.70
Minimax	0.88	1.00	NA	0.00	0.75	0.80
α-β	0.90	1.00	1.00	NA	0.90	0.60
UCT	0.80	0.36	0.25	0.10	NA	0.50
ϵ-UCT	0.70	0.30	0.20	0.40	0.50	NA

From Table I, The Random players can be regarded as a benchmark. It cannot beat more intelligent players in most of the games. The alpha-beta can get the highest wining rate among other five players whoever the opponent is. UCT and e-greedy UCT were running neck and neck. However when the competition is alpha beta. UCT get lower winning rate than e-greedy UCT. α-β pruning always win compared to minimax searching because α-β can search deeper within the same time. If they search the same depth, α-β will always lose, which is proved through our experiment. The limited testing did not reveal the obvious improvement of ϵ-greedty UCT. Thus, ϵ-greedty UCT had no advantage when the other player is Random, Local Max and Minimax.

V. CONCLUSION

As we know, the most intelligent Othello algorithm can win the best human Othello players. That is big success in AI study. In this report, several different algorithms were used to be applied in Othello. The default policy of UCT has been improved by us. there are little significant results which can prove that ϵ-greedty UCT is good enought, UCT and ϵ-greedty UCT have more potential advantage. For example, the time complexity for UCTs is much less than minimax and α-β when the depth and iteration is huge. Experimental results have shown that it is possible to get better winning rates using improved default policy.

For future work, it is interesting to know whether the default policy can be improved more to get more significant performance.For example, it may be a good choice to use Othello knowledge to handcraft a more intelligent default policy than random and ϵ-greedy. On the other hand, the value function in Fig. 4 also can be modified by more complex function approximators.

REFERENCES

[1] Daniel Connelly, *dhconnelly.com/paip-python/docs/paip/othello.html*.

[2] Peter Norvig, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.

[3] Jack Chen, *Application of Artificial Intelligence and Machine Learning in Othello*. TJHSST Computer System Lab, 2009-2010.

[4] Michael J. Korman, *Playing Othello with Artificial Intelligence*. Dec. 11, 2003.

[5] A. Reinefeld, *An Improvement of the Scout Tree Searching Algorithm*. ICCA Journal, 6(4): 4-14, 1983.

[6] A. Plaat, J. Schaeffer, W. Pijls and A. de Bruin *A New Paradigm for Minimax Search*. Research Note EUR-95-03, Erasmus University Rotterdam, 1995.

[7] Gunnar Andersson, *www.radagast.se/othello/howto.html*. Apr. 2, 2007

[8] Kocsis, Levente, and Csaba Szepesvri. *Bandit based monte-carlo planning*. Machine Learning: ECML, Springer Berlin Heidelberg, pp: 282-293, 2006.

[9] http://www.cs.tufts.edu/comp/131/classpages