2010 International Conference on Technologies and Applications of Artificial Intelligence

# Modification of UCT Algorithm with Quiescent Search in Computer GO

Feng Xiao, Zhiqing Liu
BUPT-JD Institute of Computer GO
Beijing University of Posts and Telecommunications
Beijing, China 100876
xfeng1986@gmail.com, zhiqing.liu@gmail.com

***Abstract*—The UCT algorithm has been widely used in computer GO with significant improvements since its publication in 2006. However, the UCT algorithm does not address the horizon effects in game tree search, which is an important issue in computer games and it is typically addressed through quiescent search. This paper proposes a modification of the UCT algorithm such that quiescent search can be integrated in the UCT algorithm and proposes several conditions to trigger quiescent search in computer GO. Experimental results show that our modification can increase the game tree search depth significantly and improve its winning rate against GNUGO.**

***Index Terms*—UCT algorithm; horizon effects; quiescent search; computer GO;**

## I. Introduction

The UCB algorithm [2] is a family of efficient algorithms that can balance the dilemma of "exploration-exploitation"[6] in machine learning as exemplified in the multi-armed bandit problem [7]. The UCT algorithm [1] is an application of the UCB algorithm in tree search. The main idea of the UCT algorithm is to regard each internal tree node as an independent bandit, with its child-nodes as independent arms of the node. Instead of dealing with each node once iteratively, the UCT algorithm plays a number of tree traversals within a limited amount of time, each beginning from the root and ending with an evaluation at one leaf node. The evaluation result is then propagated back to the root of the tree along the traversal path to update the statistics of all tree nodes on the traversal path.

The UCT algorithm has been successfully used in computer games especially in computer GO [8]. For UCT search on a game tree in computer GO, each node of the game tree represents a GO board situation, with child-nodes representing next situations after corresponding moves. Each GO board situation is a bandit problem, where each legal move is an arm with unknown reward but of a certain distribution. After evaluation at leaf node, only two kinds of result are possible, the winning ones and the losing ones corresponding to reward of 1 and 0 respectively. So when propagating back on the game tree, every node updates its win-rate and visit-number, which will be used to determine subsequent search paths. MoGo is the first computer GO program that incorporates the UCT algorithm in its game tree search [6].

While the UCT algorithm indeed search deeper on the game tree path that is likely to be taken by both of the players in GO, it does not specifically address "horizon effects"[9] in game tree search. As such it leads to fewer opportunities to explore for deeper tree nodes. Specifically the UCT algorithm would stop and evaluate a node that is on the horizon and does not explore any further, regardless whether the horizon node is quiet or not. Evaluation on a horizon node that is not quiet is highly unreliable, as such it may lead to incorrect search results. We shall use the term "hot nodes"to refer to any game tree nodes that are not quiet and their evaluation is unreliable, and shall use the term "quiet nodes"to refer to any game tree nodes that are quiet and their evaluation is reliable.

In order to address the horizon effects in the UCT algorithm, we present in this paper a modification of the UCT algorithm, in which hot nodes can be identified based on domain-dependent rules and the UCT algorithm is modified to continue the search beyond the hot nodes until quiet nodes are reached and as such tree node evaluations are conducted only on quiet nodes but not on hot nodes. In short, this is equivalent to integrating quiescent search into the UCT algorithm. Experimental results show that our modified UCT algorithm would increase the average search depth of the tree path returned the search algorithms, leading to stronger performance in computer GO as tested against GNUGO.

We organize the rest of the paper as follows. Section 2 briefly introduces horizon effects in computer GO and explains how they may complicate the UCT algorithm. Section 3 presents a modification of the UCT algorithm such that horizon effects can be addressed by quiescent search that bypasses evaluation on hot nodes. Section 4 specifies several conditions to identify hot nodes in computer GO. Experiment results are reported in Section 5 and Section 6 presents our conclusion remarks and discusses future work.

## II. Horizon effect

### A. Horizon Effect in Computer GO

The horizon effect, coined by the Hans J. Berliner, is an important problem in AI. It is also referred to as the horizon problem. When searching a large game tree, it is usually infeasible to search the entire tree, so the tree is normally only partially searched. This results in the horizon effect where a significant change exists just over the "horizon"(slightly beyond the depth the tree has been searched) meaning that evaluating a partial tree may give a misleading result. An

978-0-7695-4253-9/10 $26.00 © 2010 IEEE
DOI 10.1109/TAAI.2010.81



IEEE computer society

example of the horizon effect occurs when some negative event is inevitable but postponable, because only a partial game tree has been analyzed, it will appear to the system that the event can be avoided when in fact this is not the case.

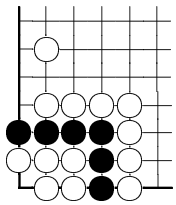

Fig. 1. Hot node on horizon.

We present an example of horizon effect in computer GO. Fig. 1 shows an example of GO, in which it's time for the black to play. The six black points surrounded by whites has only one liberty. If black do not move first in this local area, it will be captured immediately. However, if the black moves first at the lower left corner, the five white stones will be first captured by the black and the liberties of the black stones will be increased to five. But this move can't prevent the white eventually capture black points in this local area and its effect is just to delay this result. As shown in the Fig. 2, after 16 moves, the black stones will be captured by the white at last.

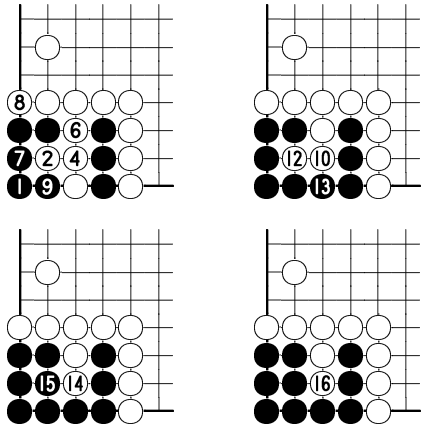


Fig. 2. The situation after horizon.

So if the searching depth is less than 16 on the game tree, the black can play at the lower left corner and keep its stone alive. Except in special circumstances such as ko, this move is always a bad move. Subsequently, we shall use the term "hot nodes"to refer to game tree nodes in which horizon effect may occur.

### *B. Horizon effect in UCT algorithm*

The UCT algorithm does not address horizon effect as shown in the example of computer GO in the last section, and this is shown in Fig. 1. So in accordance with UCT algorithm, if we evaluate the "hot nodes"of Fig. 1 immediately with the Monte Carlo method, then its evaluation result is likely to be biased to the wrong side and as such mislead subsequent search of the game tree. This is more evident in Fig. 3. After Monte Carlo evaluation, its result is 80%, meaning that it is winning with a probability of 0.80, caused by the horizon effect, because the UCT algorithm has reached the horizon in game tree search. However, this move has been proved to be bad.

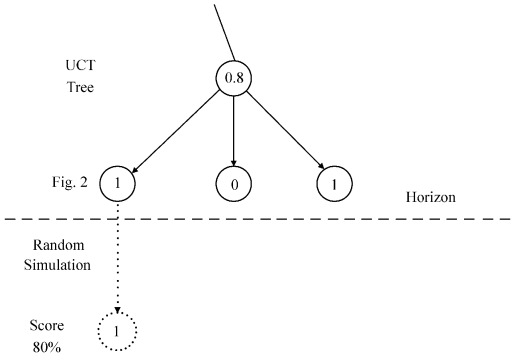


Fig. 3. Horizon effect in UCT algorithm.

This overall effect of the horizon problem on the UCT algorithm is likely to be significant, due to largely the fact that this occurs on leaf nodes of the game search tree, whose number of search visits is typically very small. A small number of visits on a game tree node is unlikely to produce a reliable evaluation of the node unless its Monte-Carlo evaluation is reliable, which is not the case as discussed above. Therefore, UCT evaluation results of all "hot nodes"on the horizon are unreliable, which may mislead subsequent and overall UCT search and evaluation results. As such, we shall propose in the next section a modification of the UCT algorithm with integrated quiescent search to overcome the horizon effect.

## III. Modification of UCT with Quiescent Search

Quiescent search is the standard solution to the horizon effect in game tree search. This section presents a modification of the UCT algorithm in which quiescent search is integrated. The essential idea of quiescent search is that, when "hot nodes"are identified at horizon, the game trees rooted at hot nodes are expand until quiescent nodes are reached and reliable evaluation on quiescent nodes can be conducted. The pseudo code of our modified UCT algorithm is shown below.

The pseudo code without line 8 to 10 depicts the original UCT algorithm. The UCT algorithm conducts a tree search based on the UCB algorithm selecting the next node in tree search in line 4. The tree search part should end by creating a new leaf node of the tree in line 7. Evaluation of the leaf node is conducted in line 11, and the evaluation value is propogated back along the tree search path in line 12.

The part of line 8 to 10 is the quiescent search applied to UCT algorithm. The quiescent search is triggered by two conditions: First, the node must be a hot node, and second, the node shall be within a search depth threshold. Such hot nodes will be expanded as a part of quiescent search until reaching a quiescent node or the predefined search depth. Fig. 4 shows of the modified UCT algorithm on the example of Section 2. With quiescent search, the simulation returns 0 in 95% probability. So we could prevent the tree's expanding to the wrong side.

**Algorithm 1** Play one sequence in UCT with quiescent search.

playOneSequence($rootNode$)

```
1: i = 0;
2: node[i] = rootNode;
3: for node[i] is not first visited do
4:    n = descendByUCB1(node[i]);
5:    node[++i] = n;
6: end for
7: addChild(node[++i]);
8: for hot(node[i]) and node[i].depth ≤ threshold do
9:    addChild(node[++i]);
10: end for
11: node[i].value = getValueByMC(node[i]);
12: updateValue(node, -node[i].value );
```

Discussions of hot node identification will be presented in the next section.

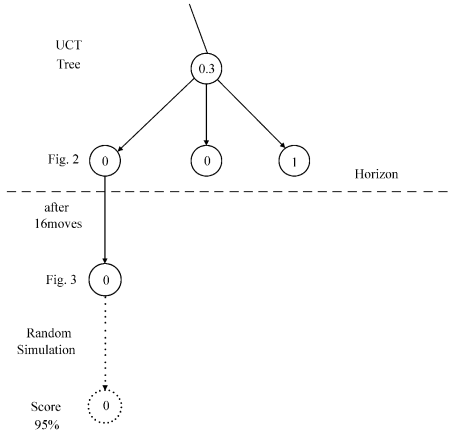


Fig. 4. Modification UCT of quiescent search.

## IV. Identify "hot nodes"with domain knowledge

Identification of hot nodes are very much domain dependent. This section presents a number of rules to identify hot nodes in game trees of computer GO. In order to help identify hot nodes, we add some quiescent information to tree nodes, and use domain knowledge to determine whether a node is hot or quiescent. This section describes two conditions for hot node identification, which complied with the following four basic rules.

1) *There is no "ko"on the board.*
2) *Never fill an eye or a tiger point of oneself.*
3) *Never fill an eye or a tiger point of the opponent.*
4) *The point played must have at least two liberties.*

We now analyze conditions in a GO board to determine whether a node is hot or quiescent. Table 5-2 will show how these conditions may improve the overall performance of a computer GO program.

### A. Capture and Escape

Capture and escape is the first class of features to determine whether a node is hot or quiescent. Players of GO usually respond to a capture move with an escape move. As such, the following two conditions are hot node triggers in this class of features.

1) *A board in which there exists points in atari caused by the last move is a hot node, a subsequent move on this board may include moves that may escape these points by either extending on their liberty or by capturing opponent's stones.*
2) *A board in which there exists points with only two liberties caused by the last move and these points can be captured by ladder if not responding is a hot node, a subsequent move on this board may include moves that may escape these points by either extending on their liberty or by capturing opponent's stones.*

Fig. 5 shows an example of the first condition of capture to escape with just one liberty. In this example, the only moves we may take include the points which could make our points escaped, which are marked "x"in Fig. 5. Fig. 6 shows an example of the second condition of capture to escape with two liberties. In this example, the only moves we may take include the points which could make our points escaped, which are marked "x"in Fig. 6.

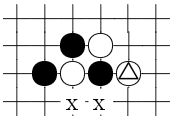


Fig. 5. Capture to escape 1.

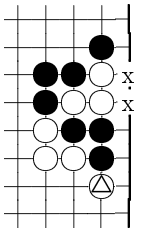


Fig. 6. Capture to escape 2.

### B. Cut and Connect

The second class of features to determine whether a node is hot or quiescent consists of 3×3 patterns used in MoGo Monte-Carlo simulation [4]. These patterns are just applied to the eight points around the last move. While many patterns were used in MoGo Monte-Carlo simulation, some are not very effective in hot node identification. Through test and error, we have settled with just a few number of the most important and urgent patterns, which improves the overall effects significantly. Furthermore the average depth in UCT game tree search can be greatly enhanced as we will discuss in the next section.

Fig. 7 and Fig. 8 show the 3×3 patterns used to identify the second class of features of hot nodes, in which the position in the middle of the 3×3 patterns is the position where the next move is supposed to be played. [5] Fig. 7 shows the cut and connect 3×3 patterns, which matches any board positions

where the 6 upper points are exactly matched and the 3 bottom points are not white. Fig. 8 shows special patterns for moves on the edge of the GO board. In this figure, a square on a black (respectively white) stone means true is returned if and only if the positions around are matched and it is black (respectively white) to play.

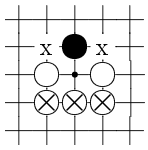

Fig. 7. Cut and connect 3×3 patterns.

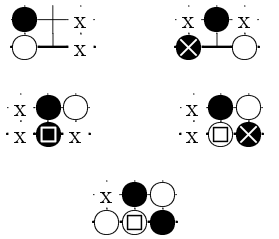

Fig. 8. Patterns on the edge.

## V. EXPERIMENTS

We have conducted experiments to compare the original UCT algorithm and our modified UCT algorithm with quiescent search. All of the experiments are conducted in computer GO with the Lingo program playing against GnuGo 3.8 with its default mode. Komi are set to be 7.5 points. Lingo is our computer GO program developed at BUPT-JD Computer-GO Research Institute, and its core is the UCT algorithm.

The experiment environment is a Linux system with 4GB primary memory and 4-core each running at 1.85GHz. Only one core is used in all of our experiments. Table 5-1 shows the performance results of Lingo using the original UCT algorithm playing against GNUGO 3.8 under various time constraints; while Table 5-2 shows the results of Lingo using our modified UCT algorithm. All results are for 9×9 board.

| $times(s)$ | $win\_rate(200)$ | $depth$ |
|---|---|---|
| 5 | $81.2 \pm 3.8$ | 8.4 |
| 8 | $87.1 \pm 2.8$ | 9.1 |
| 10 | $89.4 \pm 2.3$ | 9.8 |
| 15 | $91.0 \pm 2.9$ | 10.5 |

Table 5-1. Performance of the original UCT algorithm.

| $times(s)$ | $win\_rate(200)$ | $depth$ |
|---|---|---|
| 5 | $90.3 \pm 3.7$ | 12.9 |
| 8 | $93.7 \pm 3.8$ | 13.3 |
| 10 | $95.1 \pm 4.4$ | 13.4 |
| 15 | $95.3 \pm 2.6$ | 14.1 |

Table 5-2. Performance of the modified UCT algorithm with quiescent search.

It is clear from the experiment results that our modified UCT algorithm with quiescent search is significantly better than the original UCT algorithm, both in terms of the win-rate and in terms of the search depth. From the result of experiments we can find that the performance of UCT using horizon is better than the original. Additionally, the improvement is more significant when the time constraint of UCT search is tight. Lingo with our modified UCT algorithm with quiescent search using 5 seconds per move is as strong as Lingo with the original UCT algorithm using 15 seconds per move.

## VI. CONCLUSIONS

This paper discusses potential problems of horizon effect in use of the UCT algorithm, and proposes a modification of the UCT algorithm with an integration of quiescent search. This paper also presents two classes of features that can be used to identify hot nodes in computer GO. Our experiment results show that our modified approach is more efficient in computer GO search, especially when search time is limited.

We should also point out that the features for hot node identification and for horizon expansion are quiescence. Therefore, domain knowledge is one of the crucial elements in this approach. Because the quiescent knowledge we used is limited and subjective, the experiment results are not optimized and inevitably have certain deviations. If we could refine the static knowledge, it is possible to further improve the efficiency of the modified UCT algorithm.

## REFERENCES


[1] L. Kocsis and C. Szepesvari. Bandit based monte-carlo planning. In 15th European Conference on Machine Learning (ECML), pages $282 - 293, 2006$.

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

[3] Rmi Coulom. Efficient selectivity and backup operators in Monte-Carlo 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, To appear.

[4] P. Drake, J. Levenick, L. Veenstra, and A. Venghaus. Pattern matching in the game of Go. Submitted, 2004

[5] Sylvain Gelly, Yizao Wang, Rmi Munos, Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go, Technical Report 6062, INRIA, France, November 2006.

[6] Sylvain Gelly, Yizao Wang. Exploration exploitation in go: UCT for Monte-Carlo go. In: NIPS-2006: On-line trading of Exploration and Exploitation Workshop, Whistler Canada (2006)

[7] Shie Mannor, John N. Tsitsiklis. The Sample Complexity of Exploration in the Multi-Armed Bandit Problem. Journal of Machine Learning Research, Vol. 5 (June 2004), pp. 623-648.

[8] Xindi Cai, Donald Wunsch. Computer Go: A Grand Challenge to AI. Challenges for Computational Intelligence (2007), pp. 443-465.

[9] Russell, Stuart J. Norvig, Peter, Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 174, ISBN 0-13-790395-2