Journal of Computer Science 8 (10): 1594-1600, 2012
ISSN 1549-3636
© 2012 Science Publications

Building Opening Books for 9×9 Go Without Relying on Human Go Expertise

[1]Keh-Hsun Chen and [2]Peigang Zhang
[1]Department of Computer Science,
University of North Carolina at Charlotte, Charlotte, NC 28223, USA
[2]Microsoft Coorperation, Boulder, CO 80301, USA

Abstract: Problem statement: Expert level opening knowledge is beneficial to game playing programs. Unfortunately, expert level opening knowledge is only sparsely available for 9×9 Go. We set to build expert level opening books for 9×9 Go. **Approach:** We present two completely different approaches to build opening books for 9×9 Go without relying on human Go expertise. The first approach is based on game outcome statistics on opening sequences from 300,000 actual 9×9 Go games played by computer programs. The second approach uses off-line stage-wise Monte-Caro tree search. **Results:** After "solution tree" style trimming, the opening books are compact and can be used effectively. Testing results show that GoIntellect using the opening books is 4% stronger than GoIntellect without the opening books in terms of winning rates against Gnugo and other programs. In addition, using an opening book makes the program 10% faster. **Conclusion:** Classical knowledge and search approach does not work well in the game of Go. Recent development in Monte-Carlo tree search brings a breakthrough and new hope-computer programs have started challenging human experts in 9×9 Go. A well constructed opening book can further advance the state of the art in computer Go.

Key words: Computer Go, Monte-Carlo tree search, opening books

INTRODUCTION

The classical full board search paradigm has produced programs stronger than human expert players in a number of games such as Chess, Checkers and Othello. Yet this classical approach failed miserably in Go, since good Go knowledge does not translate to good evaluation function to be used by mini-max style full board game tree search (Chen, 2003). The playing strength of programs for 19×19 and 9×9 Go stuck at intermediate amateur level until the recent development of Monte-Carlo Tree Search (MCTS) (Coulom, 2007a; Gelly *et al*., 2006; Kocsis and Szepesv'ari, 2006), which bypassed the need of static evaluation functions and brought a breakthrough in computer Go. Much additional work has been done on MCTS and its enhancement in recent years (Chaslot *et al*., 2007; Chen *et al*., 2008; Chen and Zhang, 2008; Coulom, 2007b; Gelly and Silver, 2007).

Opening books are common in computer game playing (Buro, 1999; Lincke, 2000). The playing strength of 9×9 Go programs can be further enhanced by using expert level opening books. Unfortunately, there are no publicly available expert opening books for 9×9 Go. Even 9×9 Go game records by professional experts are scarce, not enough available for building opening books. We propose two approaches to build 9×9 Go opening books without relying on human Go expertise. The first approach is based on game outcome statistics on opening sequences from 300,000 actual 9×9 Go games played by computer programs. Top 9×9 Go programs can now challenge human experts. We discuss the details of this approach and described how to use such an opening book in a 9×9 Go program. We also discuss a second approach of using off-line stage-wise Monte-Caro tree search. The testing results of using the constructed opening books are, which shows GoIntellect using the opening books is 4% stronger than GoIntellect without the opening books in terms of winning rates against Gnugo. In addition, using an opening book makes the program 10% faster.

MATERIALS AND METHODS

We shall discuss the details of our approach in this study.

Corresponding Author: Keh-Hsun Chen, Department of Computer Science, University of North Carolina, Charlotte, NC 28223, USA

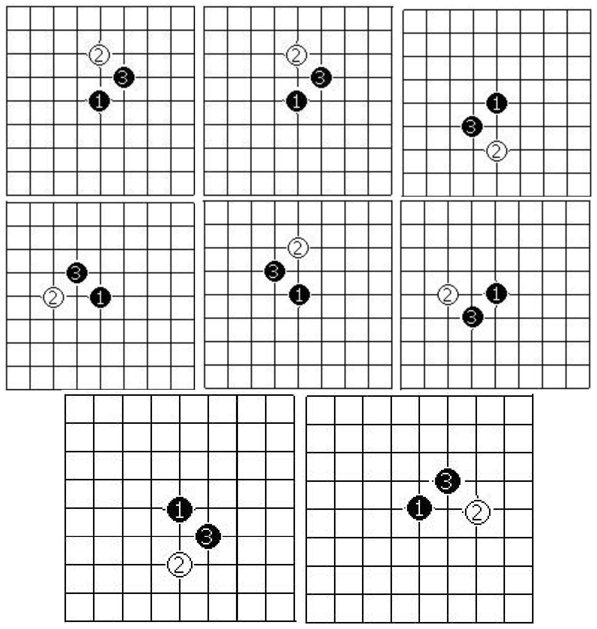

Fig. 1: Equivalent opening move sequences

Canonical orientation of a game: We call a sequence of moves of any length starting from the empty Go board an opening move sequence. An opening move sequence (including a whole game) can have equivalent move sequences in 8 different orientations. For example, the following 8 opening move sequences are all equivalent (Fig. 1)

In building an opening book, we should combine the outcome statistics of all extension games from each of the 8 equivalent initial move sequences. We don't need to consider color flip here, since Black always plays first in Go.

Let B be the set of all 81 points on the 9×9 Go board. We define 8 transformation functions from B to B:

f_0: Identity function mapping every point to itself
f_1: Rotate clockwise 90°
f_2: Rotate clockwise 180°
f_3: Rotate clockwise 270°
f_4: Reflection with respect to the vertical center line
f_5: f_1 followed by f_4
f_6: f_2 followed by f_4
f_7: f_3 followed by f_4

Applying f_0-f_7 to each move in the move sequence in the upper left diagram of Fig. 1, we get all the equivalent variations in Fig. 1.

These 8 transformations together with composite function operator form a group in modern algebra. f_0 is the identity element.

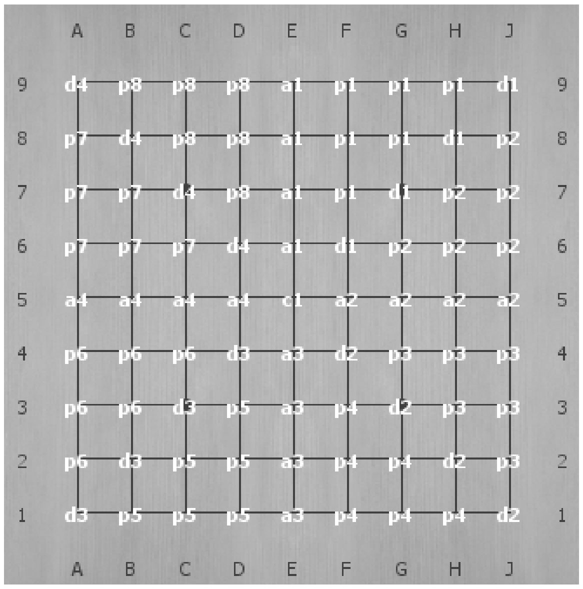
A B C D E F G H J
9 d4 p8 p8 p8 a1 p1 p1 p1 d1 9
8 p7 d4 p8 p8 a1 p1 p1 d1 p2 8
7 p7 p7 d4 p8 a1 p1 d1 p2 p2 7
6 p7 p7 p7 d4 a1 d1 p2 p2 p2 6
5 a4 a4 a4 a4 c1 a2 a2 a2 a2 5
4 p6 p6 p6 d3 a3 d2 p3 p3 p3 4
3 p6 p6 d3 p5 a3 p4 d2 p3 p3 3
2 p6 d3 p5 p5 a3 p4 p4 d2 p3 2
1 d3 p5 p5 p5 a3 p4 p4 p4 d2 1
A B C D E F G H J

Fig. 2: Position types of a 9×9 Go board

They each have an inverse transformation:

$f_1^{-1} = f_3$
$f_3^{-1} = f_1$
$f_i^{-1} = f_i$ for i = 0, 2, 4, 5, 6, 7

f_1 and f_3 are inverse to each other. The other transformations are inverse to itself. These 8 transformations can generate equivalent move sequences. Two move sequences $<m_1, m_2, \ldots, m_k>$ and $<m_1', m_2', \ldots, m_k'>$ are said to be equivalent if and only if there is an i in {0, 1, 2,…, 7} such that $f_i (m_j) = m_j'$ for j = 1, 2, …, k.

We classify all 81 9×9 Go board points into 4 types: Center, axis, diagonal and pie. As shown in Fig. 2, there are:

1 center: c_1
4 axes: a_1, a_2, a_3, a_4
4 diagonals: d_1, d_2, d_3, d_4
8 pies: p_1, p_2, …, p_8

Let t(m) be the type of move location m. We call $<t(m_1), t(m_2), \ldots, t(m_k)>$ the location type sequence of move sequence $<m_1, m_2, \ldots, m_k>$. For example, let's consider a short move sequence <E5, D3, C5>, Fig. 3.

This short opening move sequence has its location type sequence $<c_1, p_5, a_4>$. We call the sequence of the subscripts, <1, 5, 4> in this example, its location type index sequence (Table 1). We shall use this index sequence to identify the canonical form of a move sequence.

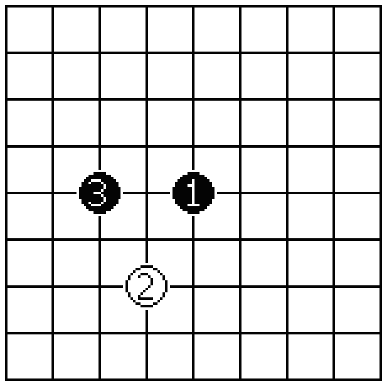
Fig. 3: An opening move sequence

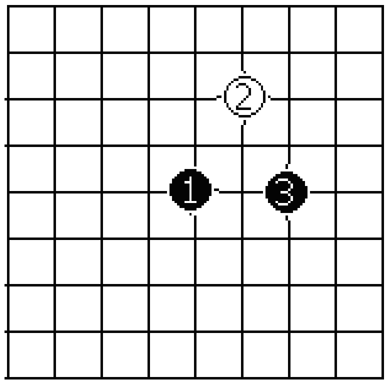
Fig. 4: Canonical form of the move sequence in Fig. 3

Table 1: The equivalent move sequences of the move sequence in Fig. 3 and their type index sequences

I	$F_i(S)$	Location type sequence	Type index sequence
0	E5, D3, C5	c_1, p_5, a_4	154
1	E5, C6, E7	c_1, p_7, a_1	171
2	E5, F7, G5	c_1, p_1, a_2	112
3	E5, G4, E3	c_1, p_3, a_3	133
4	E5, F3, G5	c_1, p_4, a_2	142
5	E5, G6, E7	c_1, p_2, a_1	121
6	E5, D7, C5	c_1, p_8, a_4	184
7	E5, C4, E3	c_1, p_6, a_3	163

A move sequence can have up to 8 equivalent move sequences under rotation and reflection (through a transform function f0-f7 on every element of the sequence). We call the one with lexically smallest location type index sequence its canonical form. Considering the earlier example opening move sequence S = <E5, D3, C5>, we have one hundred and twelve is the smallest location type index sequence, so the canonical form of S is <E5, F7, G5>, Fig. 4, which can be obtained via function f_2 (rotate clockwise 180°). We call f2 the canonical transformation for the move sequence <E5, D3, C5>. The canonical transformation converts a move sequence to the equivalent sequence in the canonical form.

The original sequence can be reconstructed via f_2^{-1} (= f_2). In this example, the canonical transformation can be determined when move 2 is played. A sufficient condition to determine a unique canonical transformation for any extension of an opening sequence is a move at a pie point.

Only about 1/8 of the actual games are in canonical form. Before we merge them into a big opening tree, we should convert them into canonical form, so we can collect all relevant statistics together for the equivalent opening move sequences.

2.2 Merge games into a tree: It would be ideal to use 9×9 games played by human Go experts to build an opening book. Unfortunately the available professional 9×9 games are rather limited. So we use 9×9 games played by computer Go programs instead. We have over 300,000 testing games of GoIntellect against GnuGo, CrazyStone, Mogo and older versions of GoIntellect plus thousands additional 9×9 games down loaded from KGS on the Internet.

All the games were in sgf format. We wrote a script to process the games one at a time. For each game, we first let GoIntellect to step through all moves in the game to reach the end configuration, then count the territory score (we use Chinese rule with 7.5 points komi) and record the win/loss result. Then find its canonical transformation by applying all transform functions to moves one at a time until the canonical orientation is determined (usually after examining no more than first 3 moves). We apply the canonical transformation to moves up to the depth limit of opening book tree; we use 16 as the limit and get an opening sequence in canonical form.

A node in the opening tree needs to record the move location plus the move/player information (we code every board location into a number and use + number for Black move,-number for White move), the number of games passing through the node and the number of winning games (say from the node move player's point of view) passing through the node. We initialize the tree, in sgf form, by using the first game. We merge a new opening sequence into the growing opening tree by tracing its move sequence (in canonical form) through the tree until it goes off the tree, then we augment the tree by attaching a branch from the node for the remaining opening move sequence in the game.

The C-like pseudo code for building an opening book from game records is shown in Fig. 5. We choose tree rather than graph as underlying data structure for the opening book for two reasons.

```
BuildOpeningBook(string OpeningBook, string [] GameLibrary)
{
   //Load existing Opening book
   if (OpeningBook exists on the disk)
        SgfTree mainTree = LoadFromSgfFile(OpeningBook);
   else
        Initialize an empty OpeningBook;
   //Add every game record to Opening book
   foreach(string sgfFile in GameLibrary)  {
     //Load one game record
     SgfTree oneGame = LoadFromSgfFile(sgfFile);
     //Transform the game to the canonical sequence
     TransformGameToCanonicalForm(oneGame);
     //Calculate game result
     bool bBlackWin = EvaluateGame(oneGame);

     //Trace down both tree along the same move path
     while(DownTree(mainTree.currentNode,oneGame.currentNode))  {
     //nuGamePassed is the # of games collected at this node
     //nuWins is the number of wins at this node
     //P_GamePassed and P_Wins are extended SGF Properties to //save those
     values
              int nuGamePassed  = GetProp(mainTree.currentNode,
                                              P_GamePassed);
              int nuWins = GetProp(mainTree.currentNode, P_Wins);
              SetProp(mainTree.currentNode, P_GamePassed,
                                              nuGamePassed++);
              SetProp(mainTree.currentNode, P_Wins, nuWins,
                                              bBlackWin);
          }

          //copy remaining move to OpeningBook tree
          while (oneGame.currentNode.childNode != NULL) {
              oneGame.currentNode = oneGame.currentNode.childNode;
              mainTree.currentNode = CreateChildNode(mainTree.
                          currentNode,oneGame.currentNode.Move);
              SetProp(mainTree.currentNode, P_GamePassed, 1);
              SetPropExt(mainTree.currentNode, P_Wins, nuWins,
                                                     bBlackWin);
          }
        }
        //Save Opening book
        SaveSgfTreetoFile(mainTree, OpeningBook);
     }
```

Fig. 5: C-like pseudo code for opening book building from game records

First, it is much more efficient to build a tree from game records than to build a graph. Second, sometimes the path leading to a node affects the set of legal moves at the node (ko status).

Due to the limitation on the memory, we can't just build a giant game tree of 300,000 games. We have to trim game trees before they get too big, then merge trimmed trees together. We developed a procedure to merge many opening trees into one big opening tree, so we can build it a reasonable size piece at a time. Several opening books can be merged into one via the pseudo code in Fig. 6.

Trim an opening tree to an opening book: We can trim move beyond opening depth, if we did not do so before the game merging. Also we can trim away any node with fewer than a threshold number of games passing through, we use 20 as the threshold, so the remaining nodes are more reliable. At this point, nodes with very low winning rates, say less than 25%, can be pruned, since they are likely to be bad moves.

```
MergeTwoOpeningBooks(SgfNode openBookNode1, SgfNode, openBookNode2){
        //Merge current node value
        int nuGamePassed1 = GetProp(openBookNode1, P_GamePassed);
        int nuGamePassed2 = GetProp(openBookNode2, P_GamePassed);
        int nuWins1 = GetProp(openBookNode1, P_Wins);
        int nuWins2 = GetProp(openBookNode2, P_Wins);
        nuGamePassed1 += nuGamePassed2;
        nuWins1 += nuWins2;
        SetProp(openBookNode1, P_GamePassed, nuGamePassed1);
        SetProp(openBookNode1, P_Wins, nuWins1);

        //For all child nodes of openBookNode2
        SgfNode node = openBookNode2.childNode;
        while (node != NULL){
                //If tree 1 has this child, merge it to tree 1
                SgfNode child = GetChildNode(openBookNode1,
                                                node.Move);
                if (child != NULL)
                        MergeTwoOpeningBooks(child, node);
                else    //Copy this node to opening book 1
                        CreateChildNode(openBookNode1, node);
                node = node.Brother;
        }
}

MergeOpeningBooks(string MainBook, string [] openBooks) {
        //Load main open book
        SGFTree mainTree = LoadFromSgfFile(MainBook);
        //Merge all other open book
        foreach (string openbook in openBooks) {
                //Load one openbook
                SGFTree oneBookTree = LoadFromSgfFile(openbook);
                MergeTwoOpeningBooks(mainTree.root,
                                        oneBookTree.root);
        }
        //Save open book
        SaveSgfTreetoFile(mainTree, MainBook);
}
```

Fig. 6: C-like pseudo code for merging opening books

If we are to play, we will never choose it. If the opponent selects this bad move, we probably can win without using the opening book. A sorting routine was programmed to order the children of a node according to winning rates for the whole tree providing convenience in tree manipulations.

Assume we play Black, then at each node black is to play next, we just need to keep small number of best successors and trim the rest sub-trees. In that way, we can get a compact "solution tree" opening book with size shrunk by 1000 fold. Similarly we can create a "solution tree" for White. Merging Black "solution tree" and White "solution tree", we get an opening book that can be used by either Black or White. The sgf opening tree we produced after merging 300,000 games before trimming was several hundred mega bytes in size. The final working opening tree is about 60 K bytes containing about 3000 moves.

Practice: We shall show how to make opening book moves in 9x9 Go matches. And we introduce an alternate approach of building an opening book for 9x9 G0.

Use of the opening book: The opening book is a sgf game tree containing only move sequences in canonical form. The players may play moves in any orientation. To use the book, we keep 8 tree-node pointers p_0, p_1, p_2,…, p_7, where p_i points to the node of which the move sequence from the root to it is a move sequence in canonical form $<f_i(m_1), f_i(m_2),…, f_i(m_k)>$ where $<m_1, m_2,…, m_k>$ is the actual move sequence of on the board so far, if such a node exists, otherwise p_i is null. The 8 pointers are initialized to point to the root of the opening book tree, which corresponds to the empty 9×9 board. When an actual move m is played on the board by either side, for each non-null p_i, we advance the pointer p_i to point to the successor node containing the move $f_i(m)$ if such successor exists; otherwise p_i becomes null.

When it is our turn to play and at least one p_i is not null, our book move selection is to consider all successor nodes of all nodes pointed by a p_i and pick the successor with highest winning rate. We also take the confidence factor into consideration-the more games through it the better. If the winning rate is below a threshold (45% in our implementation), we give up the book move and go back to MCTS. If the winning rate is high enough, get the node move m1 of the best child of the selected node pointed by p_i. $f_i^{-1}(m_1)$ will be our book move to play on the board. When all 8 p_i's become null, the game is out the opening book. We shall discuss building opening books using stage-wise off-line MCTS.

Off-line stage-wise Monte-Carlo tree search: We shall discuss building opening books using off-line stage-wise MCTS. The basic idea is to run the program's MC tree growing engine, i.e., UCT algorithm, days and nights to build a huge Monte-Carlo Search Tree (MCST) then take the top part as an opening book. But this basic idea has a drawback: as the tree gets bigger and bigger, the UCT algorithm will play the best move exponentially more often than the rest moves. It more or less converges to the "principle variation" path. A book should be able to provide moves responding to opponent's suboptimal play. To remedy this drawback, we use the following stage-wise strategy to combine many separate MC search trees into one big opening tree.

We first did 20 million simulations from the empty board position trying only moves in canonical orientation, which took about a half hour. We identified the top 6 opening moves based on winning rates. For each of the 6 candidate opening first move, we played a Black stone on the board at the position, then start a new MCTS to grow a new MCST. For each such MCST generated, we identified 3-5 top responses and grew a new set of MCSTs with first two moves already placed on the board. We then developed the next set of MC trees with first three moves specified. This process could go on many levels. We only selectively got to no more than 4 levels. We performed 20 million simulations for each MCST. Then we trimmed and merged them and then trimmed it again to form an opening book tree. Since we would like to store and reload MC search trees for later use. We used a compact text format to store essential information of a MC Search Tree (MCST). The following context-free grammar specifies the syntax of our MCST:

```
<MCST> ::= {<move> <num wins> <num games>
<MCST-list>}
<MCST-list> ::= <empty> | <MCST> <MCST-list>
<move> ::= <sign> <board point> | <sign><pass> | 0
<num wins> ::= <natural number>
<num games> ::= <natural number>
<sign> ::= + |-| <empty>
```

Where "{" and "}" are literals. A positive number represents a Black move and a negative number represent a White move. This format is simpler and more compact than sgf format and easier to write a parser for. The authors would like to thank Mr. Dawei Du for the implementation of the compact text disk read/write format for MCST.

RESULTS AND DISCUSSION

We tested the effectiveness of an opening book constructed from over 300 thousand actual games and another opening book generated from stage-wise off-line MCTS against GnuGo 6.0 level 10. The number of simulations per move for GoIntellect (GI) is set to 1, 2, 4, 8, 16, 32, 64, 128, 256 and 512 K (when it does not have an opening book move or does not use an opening book). For each of the two opening books, for each of the number of simulations per move setting, GI using the opening book played 100 games taking Black and another 100 games taking White and GI without opening book played the same number of games for comparison of the outcomes. A total of 6000 games played on various PCs. The result is summarized in Fig. 7.

The versions of GI using opening book outperformed the version without opening book by about 4% on the average in winning rates. Furthermore, when there is an acceptable opening book move available, the program consumes very little time. The time saved can be used by later moves. GI with opening book typically retrieves 2-6 opening moves from the opening book a game, saving about 10% of the time.

GI with the opening book from actual games performed slightly better than GI with the opening book from off-line MC simulations. The outcome may reverse if we use more simulations for a building block and more layers of building blocks in off-line MC simulation based opening book. Additional testing games played against other programs showed similar playing strength improvements.

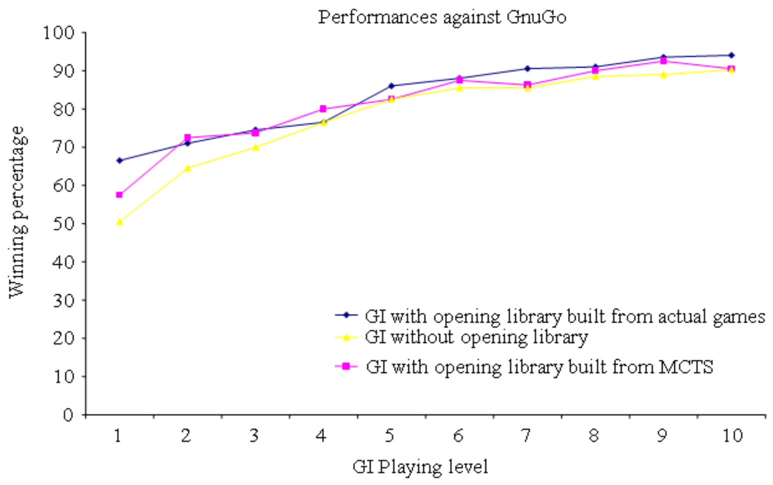

Fig. 7: Experimental results against GnuGo 6.0 level 10. GI with playing level k performs 2^{k-1} K simulations per move

CONCLUSION

Opening books can help programs play stronger and faster. When expert knowledge is not readily available, we can build opening books by combining actual games and using the outcome statistics to guide the move selection. The book can be trimmed to a compact size leaving out nonessential portions of the tree. Off-line stage-wise MCTS approach is equally effective. The full board opening book approach is effective in 9×9 Go. But when the size of the Go board increases, the outcome statistics on opening sequences become rather sparse and less reliable. For 19×19 Go, instead of building opening books for the full board, we build opening books for corners, called Joseki dictionaries. We use human expert knowledge in this case-Joseki dictionary books are abundant.

In Joseki dictionaries for corners, we also consider Black and White flip. Each of the 4 corners has 4 different variations of a Joseki from reflection w.r.t. its main diagonal and color flip. So each Joseki has 16 equivalents, 4 for each corner. The same techniques described can be used to play standard corner moves using a Joseki dictionary (a move tree) with 16 Joseki tree node pointers, 4 for each corner. The experience of Go Intellect has been that Joseki dictionaries have little benefit in 19×19 Go matches.

REFERENCES

Buro, M., 1999. Toward opening book learning. ICCA J., 22: 98-102.

Chaslot, G.M.J.B., M.H.M. Winands, H.J.V.D. Herik, J.W.H.M. Uiterwijk and B. Bouzy, 2007. Progressive strategies for monte-carlo tree search. New Math. Natural Comput., 4: 343-357.

Chen, K. and P. Zhang, 2008. Monte-Carlo go with knowledge-guided simulations. ICGA J., 31: 67-76.

Chen, K., 2003. Computer go: Increasing interest. ICGA J.

Chen, K., D. Du and P. Zhang, 2008. A fast indexing method for monte-carlo go. Comput. Games, No. 5131: 92-101. DOI: 10.1007/978-3-540-87608-3_9

Coulom, R., 2007a. Computing ELO ratings of move patterns in the game of go. University of Alberta.

Coulom, R., 2007b. Efficient selectivity and backup operators in Monte-Carlo tree search. Proceedings of the 5th International Conference on Computers and Games, (CG' 07), Springer-Verlag Berlin, Heidelberg, pp: 72-83.

Gelly, S. and D. Silver, 2007. Combining online and offline knowledge in UCT. Proceedings of the 24th International Conference on Machine Learning, (ML' 07), ACM Press, USA, pp: 273-280. DOI: 10.1145/1273496.1273531

Gelly, S., Y. Wang, R. Munos and O. Teytaud, 2006. Modifications of UCT with Patterns in Monte-Carlo Go. Institute National de Recherche en Informatique Et En Automatique.

Kocsis, L. and C. Szepesvari, 2006. Bandit based monte-carlo planning. Computer and Automation Research Institute.

Lincke, T.R., 2000. Strategies for the Automatic construction of opening books. Proceedings of the 2nd International Conference on Computers and Games, (CG' 00), Springer-Verlag, London, UK., pp: 74-86.