2010 International Conference on Technologies and Applications of Artificial Intelligence

Dynamic Randomization Enhances Monte-Carlo Go

Keh-Hsun Chen

Department of Computer Science, University of North Carolina at Charlotte
Charlotte, NC 28223, USA
chen@uncc.edu

Abstract—**This paper proposes two dynamic randomization techniques for Monte-Carlo Go that uses Monte-Carlo tree search with UCT algorithm. First, during the in-tree phase of a simulation game, the parameters are randomized in selected ranges before each simulation move. Second, during the playout phase, the order of simulation move generators are hierarchically randomized before each playout move. Both dynamic randomization techniques increase diversity while keeping the sanity of the simulation game. The first technique, dynamic randomization of the parameters, increase the winning percentage of the author's program GoIntellect(GI) against GnuGo3.8 by 8 percentage points on the average in 19x19 games. The second technique used in conjunction with the first technique further increases the winning percentage of GI against GnuGo3.8 by up to an additional 7+ percentage points in 19x19 games.**

Keywords-component: Monte-Carlo Tree Search, UCT algorithm, simulation game, Go, search parameters, move generators, dynamic randomization.

I. INTRODUCTION

Because of its intrinsic difficulty in positional evaluation and its large branching factor, Go has been the most challenging board game for AI research. Monte-Carlo tree search (MCTS) with UCT algorithm ([1], [2]) is the most effective approach known today in tackling Go by the computer.

To further improve the playing strength of Go programs, Rapid Action Value Estimation (RAVE) and prior knowledge have been incorporated into UCT algorithm [3], [4]. There are several key parameters in the UCT algorithm incorporating RAVE and prior knowledge that need to be tuned for the program to perform well. These key parameters are identified in Section II. The methods used by Go Intellect to "optimize" their values are discussed in Section III. Recently the author realized that there is no optimal set of the parameter values. If we randomize those parameter values dynamically in reasonable intervals, the program increases diversity in sampling and performs significantly better than any fixed parameter values tried before. This technique is presented in details in section IV. Experimental results on 19x19 Go against GnuGo 3.8 are shown in Section V. We carry the parameter randomization idea one step further to hierarchically randomize the orders of the move generators in the playouts in Section VI. The substantial further improvement is shown in Section VII. Section VIII provides concluding remarks.

II. KEY PARAMETERS OF MCTS

The basic UCT algorithm can be outlined as follows [4]:

```
Initialize the tree T to only one node, representing the
                current Go board situation.
 while (total number of simulations < limit) {
    Simulate one game g from the root of the tree to a
        final position, choosing moves as below:
      Bandit part: for a situation in T, choose the move
        with maximal score according to the UCB1
        formula [5].
      MC part: For a situation out of T, choose the move
        according to a playout policy.
      Update win/loss & #simulations statistics in all
        situations of T crossed by g.
      Add in T the first situation of g which is not yet in
        T.
 }
    Return the move simulated most often from the root of
        T.
```

The UCB1 formula is given by

$$s_i = r_i + \sqrt{\frac{\log(n)}{c * n_i}}$$

Where s_i is the score of child i, $r_i = w_i / n_i$ is the win rate of child i, n is the total number of simulation games going through the parent node, n_i is the number of simulation games going through child i, and c is a constant to be tuned – our first key parameter. The formula balances exploitation vs. exploration. Within the Monte-Carlo Search Tree (MCST), the UCT algorithm advances from the parent node to the child node maximizing the score s_i. Progressive widening technique ([6], [7]) is used to control the branching of the MCST.

GI uses RAVE as described in [3]. Assume child i has a RAVE UCB1 score t_i. Then the maximum linear combination score will be used to select child node to advance:

$v_i = b * t_i + (1-b) * s_i$

978-0-7695-4253-9/10 $26.00 © 2010 IEEE
DOI 10.1109/TAAI.2010.84

where $b = \sqrt{\frac{k}{3n+k}}$ is a constant between 0 & 1. The constant k is our second key parameter. Now assume we have some prior knowledge about the successor moves represented by numerical weight u_i. We shall treat the prior knowledge as if we had played k_1 (virtual) simulation games with $k_1 * u_i / u_{max}$ (virtual) wins where $u_{max} = \max \{ u_i \mid i = 1, 2,\ldots,\#Children\}$. That is, the win rate is now calculated as $r_i = \frac{w_i + k_1 * u_i / u_{max}}{n_i + k_1}$ in the UCB1 formula. GI has considerably more knowledge at the root node than at descendant nodes in the MCST, since many traditional GI knowledge routines are used at the root. So a separate parameter k_0 is used for the root and parameter k_1 for all other nodes. Finally, since RAVE has more negative effect than benefit when n is large. GI uses a 5th parameter h, which is a threshold. When the number of simulation games passing through a node is greater h, RAVE is not used at the node in selecting the child node. Naturally, the first problem was to tune these 5 key parameters c, k, k_0, k_1, & h.

III. TUNING THE PARAMETERS

When the author first developed MCTS based version of Go Intellect about 4 years ago, dynamic hill climbing was used to tune the five key parameters as follows (selecting an opponent program of comparable strength, say of playing strength within 200 Elo points from GI to be tuned):

Initialize c, k, k_0, k_1, & h via guessing, then set tuning step sizes Δc, Δk, Δk_0, Δk_1, & Δh.

```
Parameter p = nil;
While (time is available) {
    Play a game against a selected opponent program;
    Output all parameter values & the game result;
    If (GI lost) {
        If (p) {
            Δ p = - Δ p;
            p = p + Δ p;
        }
        Randomly select a parameter p from the above five,
        let p = p + Δ p;
    }
}
```

Examine the output log to find the set of parameter vales with most wins. Last year, GI used genetic algorithm with UCB1 to tune the parameters [8].

Using both methods, the best set of parameter values were determined to be

$c = 12$, $k = 2000$, $k_0 = 24$, $k_1 = 12$, & $h = 5000$

Recently, through extensive experiments, the author found that dynamically changing parameters work much better than fixed parameters in 19x19 Go. It increases diversity and returns better statistics from Monte-Carlo simulations.

IV. DYNAMIC PARAMETER RANDONMIZATION

Each of the five key parameters are given a reasonable range through experiments. GoIntellect uses the following ranges for the key parameters:

$4 < c < 25$, $1000 < k < 4000$, $8 < k_0 < 40$, $5 < k_1 < 19$, and $2000 < h < 10000$. Before applying UCB1 formula to move down the tree one level, the 5 parameters are reset randomly in their intervals. Experimental testing shows this doesn't seem to affect 9x9 games much but improves the strength of 19x19 Go significantly. The following section reports the experimental results.

V. PERFORMANCE ENHANCEMENT

19x19 GnuGo 3.8 level 10 was used as testing opponent. Two versions of Go Intellect were tested. One version used the fixed parameter values as shown in Section 3. The other version used dynamic parameter randomization before each node advance in the MCST as described in Section IV. 10K, 20K, 40K, & 80K were used as the number of simulations per move decision for GI in the experiments.

Figure 1 shows the comparisons of the win rates of GI with the indicated number of simulations per move decision against GnuGo. Each bar is based on 400 testing games, including 200 games GI playing Black and another 200 games GI playing White. Blue bars represent the results when GI used fixed set of parameters; Red bars represent the results when GI used randomized parameters. Randomizing parameters improves win rates by an average of about 8 percentage points.

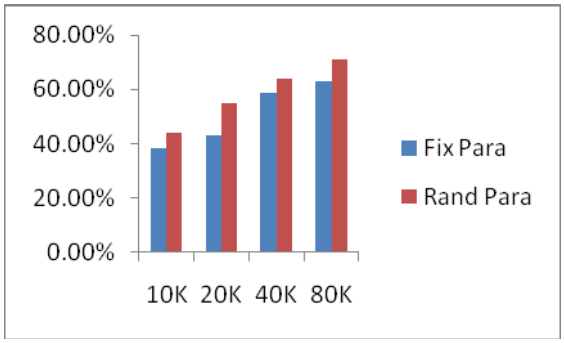

Figure 1. The X-axis is the number of simulations per move decision for GI. The Y-axis is the win rate against 19x19 GnuGo 3.8 Level 10. Each bar is based on testing results of 400 games. The blue bars represent that GI using fixed parameters. The red bars represent GI using dynamically randomized parameters.

The author also experimented with dynamic parameter randomization before each simulation game (instead of each simulation move). The result is not quite as good.

Virtual loss technique [9] should be avoided when dynamic parameter randomization is in use or the program performance would degenerate (based on a separate experiment result).

In the next section, we shall investigate randomization in playouts.

VI. HIERACHICAL MOVE GENERATOR ORDER RANDOMIZATION

The playout policy of GI is basically to perform the following 11 move generators in the given priority order. As soon as a valid move is generated, it is played in the playout.

1. Capture Last Opponent Move
2. Atari Defend
3. Nakade (play in the middle of 3 empty eye space)
4. Pattern around the opponent's last move
5. Capture
6. Last Good Reply [10]
7. Pattern around the 2nd last move
8. Play on a liberty of a 2-liberty block
9. Fill Board [4]
10. Whole board random with replacement [11]
11. Pass

For move generator randomization in the playouts, these move generators are divided into 5 priority classes: {1, 2, 3}, {4, 5, 6}, {7, 8}, {9, 10}, {11} in the given order. For each priority class, we randomize the ordering of the members. This gives us a dynamically reordered new playout policy for each simulation move in a playout.

VII. FURTHER PERFORMANCE ENHANCEMENT

This strategy of dynamic randomization of hierarchical move generators further enhances the program performance as shown in Figure 2. We see the technique of dynamic randomization of hierarchical move generators is beneficial when the number of simulations per move is higher than some threshold (about 30K). And it seems the more simulations per move the bigger benefit it produces. At 80K playouts per move decision the further improvement is over 7 percentage points in win rate against GnuGo.

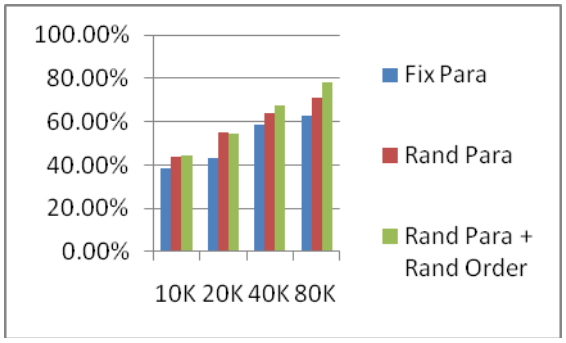

Figure 2. The X-axis is the number of simulations per move decision. The Y-axis is the win rate against 19x19 GnuGo 3.8 Level 10. Each bar is based on testing results of 400 games. The blue bars represent that GI using fixed parameters. The red bars represent GI using dynamically randomized parameters, the green bar represents GI using both randomization techniques.

If we do the move generator randomization once before each playout instead of before each playout move, we get comparable improvements except for the 80K case in which the improvement is slightly less than randomization before every playout move.

VIII. CONCLUSION AND FUTURE WORK

This paper proposed two dynamic randomization techniques for Monte-Carlo Go: dynamic parameter randomization and dynamic hierarchical move generator randomization. We demonstrated their effectiveness on 19x19 Go. These two techniques are easy to implement. They increase the diversity of simulation games while preserving the sanity. It was surprising that these techniques had only marginal effect on 9x9 Go. Finding out why 9x9 Go is immune from the dynamic randomization could give us insight into the nature of Monte-Carlo Tree Search. Can dynamic randomization work on Go MC Tactic search [12]? Or on MCTS for other games? These questions can only be answered by further research.

REFERENCES

[1] L. Kocsis and C. Szepesv´ari. Bandit Based Monte-Carlo Planning. In J. Furnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006, Lecture Notes in Artificial Intelligence 4212*, pages 282–293, 2006.

[2] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.

[3] S. Gelly and D. Silver. Combining Online and Offline Knowledge in UCT. In Zoubin Ghahramani, editor, *Proceedings of the International Conference of Machine Learning (ICML 2007)*, pages 273–280, 2007.

[4] G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud, "Adding expert knowledge and exploration in Monte-Carlo Tree Search," in Advances in Computer Games, ser. Lecture Notes in Computer Science, J. van den Herik and P. Spronck, Eds., vol. 6048. Pamplona, Spain: Springer Verlag, 2010, pp. 1–13.

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

[6] R. Coulom. Computing Elo Ratings of Move Patterns in the Game of Go. ICGA Journal, Vol. 30, No. 4, pages 198-208, 2007.

[7] G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation, 4(3):343-357.

[8] K. Chen, D. Du, and P. Zhang, "Monte-Carlo Tree Search and Computer Go", in the book "Advances in Information and Intelligent Systems", Studies in Computational Intelligence 251, edited by Z. Ras and W. Ribarsky, 2009, 201-225.

[9] G. Chaslot, M. Winands, and J. van den Herik, “Parallel Monte-Carlo tree search,” in Proceedings of the 6th International Conference on Computer and Games, ser. Lecture Notes in Computer Science, vol. 5131. Springer, 2008, pp. 60–71.
[10] P. Drake, “The Last-Good-Reply Policy for Monte-Carlo Go,” *ICGA Journal,* vol. 32, no. 4, pp. 221-227, Dec. 2009.
[11] K. Chen and P. Zhang, “Monte-Carlo Go with Knowledge-guided simulations”, ICGA Journal, Vol. 31, No. 2, 67-76, June 2008.
[12] P. Zhang and K. Chen, “Monte-Carlo Go Tactic Search”, New Mathematics and Natural Computation Journal, Volume 4, Number 3, 359-367, November 2008.