Knowledge-Based Systems 34 (2012) 21–25

Contents lists available at SciVerse ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Dynamic randomization and domain knowledge in Monte-Carlo Tree Search for Go knowledge-based systems

Keh-Hsun Chen *

Department of Computer Science, University of North Carolina at Charlotte, Charlotte, NC 28223, USA

ARTICLE INFO

Article history:
Available online 27 August 2011

Keywords:
Monte-Carlo Tree Search
UCT algorithm
Simulation game
Domain knowledge
Go
Search parameters
Move generators
Dynamic randomization

ABSTRACT

This paper is an extension of the article [13] presented at IWCG of TAAI 2010. It proposes two dynamic randomization techniques for Monte-Carlo Tree Search (MCTS) in Go. First, during the in-tree phase of a simulation game, the parameters are randomized in selected ranges before each simulation move. Second, during the play-out phase, the priority orders of the simulation move-generators are hierarchically randomized before each play-out move. Essential domain knowledge used in MCTS for Go is discussed. Both dynamic randomization techniques increase diversity while keeping the sanity of the simulation games. Experimental testing has been completely re-conducted more extensively with the latest version of GoIntellect (GI) on all three Go categories of 19×19, 13×13, and 9×9 boards. The results show that dynamic randomization increases the playing strength of GI significantly with 128K simulations per move, the improvement is about seven percentage points in the winning rate against GnuGo on 19×19 Go over the version of GI without dynamic randomization, about three percentage points on 13×13 Go, and four percentage points on 9×9 Go.

1. Introduction

Because of its intrinsic difficulty in positional evaluation and large branching factor, Go has been the most challenging board game for AI research since the dawn of computer Go 40 years ago [23,26]. Domain knowledge has been essential in the development of computer Go [2,9,22]. Go playing programs can be viewed and researched as knowledge based systems [14,21,24].

MCTS with the UCT algorithm [17,19] is the most effective approach known today in tackling Go by a computer. Some essential domain knowledge is needed for MCTS to be effective in Go [10,18]. Suitable randomization on key parameters of the UCT algorithm based MCTS and on the priorities of domain knowledge items can increase diversity while keeping the sanity of the simulation games and thus improve the performance of the program significantly. On 19×19 GI with 128K simulations per move, the increase on the winning rate against GnuGo is about seven percentage points over the version of GI without dynamic randomization. On 13×13 (resp. 9×9) GI with 128K simulations per move, the improvement is about three (resp. four) percentage points.

The MCTS with UCT algorithm is outlined in Section 2. The domain knowledge, used by GI on branching in Monte-Carlo (MC) search tree, is discussed in Section 3. The dynamic parameter randomization technique is investigated in Section 4. Section 5 is devoted to the technique of dynamic randomization of the ordering of the move-generators for the simulation play-outs. Section 6 presents the experimental results of combing both randomization techniques. Section 7 contains concluding remarks and future research.

2. Monte-Carlo Tree Search

The basic UCT algorithm with Progressive Widening for MCTS can be outlined as follows [6,15]:

```
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 [1] unless #simulation-
   games through the node exceed the threshold set by
   Progressive Widening [15], in that case a new child node is
   branched for the next highest value move determined by
   the in-tree Go knowledge.
  MC part:
  For a situation out of T, choose the move according to a
```

(*continued on next page*)

* Tel.: +1 704 687 8545; fax: +1 704 687 3516.
E-mail address: chen@uncc.edu

0950-7051/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.knosys.2011.08.007

play-out policy.
Update win/loss and #simulations statistics in all situations of T crossed by g.
Add to T a new descendant node for the first situation of g which is not yet in T.
}
Return the move simulated most often from the root of T.

We shall discuss Go knowledge used by GI in Monte-Carlo (MC) search tree branching in the next section. The UCB1 formula will be given and discussed in Section 4.1.

3. Domain knowledge for MC tree branching

Go has a very high branching factor – over 200 on average for 19×19 Go. If the MC search tree tries to cover all legal branches, then the tree depth would be rather shallow and the MCTS would not be effective. We need to use domain knowledge to order the candidate moves, and then use Progressive Widening gradually to explore more move options. Since we want to do as many simulations as feasible, speed is a key factor. The most important node in the MC search tree is the root node because the move decision is to select the move corresponding to the root node's most promising child node. More computing resource is spent on the root node to get a high quality candidate move ordering. We shall discuss domain knowledge for the root node in Section 3.1 and for non-root node in Section 3.2.

3.1. Knowledge at the root

The root node represents the current board configuration. To pursue high quality move ordering, GI uses intense computation to obtain knowledge about the current game configuration. The essential domain knowledge computed for the root node includes [9]:

a. Mortality of blocks – use heuristic search methods to decide which blocks can be captured, how to capture, and what are the escape moves if any [25].
b. Connectivity of chains – use heuristic, pattern matching, and search to find connectivity of blocks to form chains.
c. Safety of groups – use influence and chains to identify groups and then use static analysis and heuristic search to decide the life and death status of groups. Each group gets a heuristic estimate of its safety [7,8].
d. Territory and potential territory – build upon the knowledge accumulated from the first three steps to form a heuristic estimate of territory belonging of the board points.

GI contains a couple of dozen of special purpose move generators. Each use some of the above computed knowledge to generate recommended moves for a specific sub-goal, such as capturing, extend territory, connection, attack weak opponent group, etc. Each such recommended move is associated with a recommended value. The move value of a move location is a weighted linear combination of the move values from all move generators for that move point. The candidate moves are ordered according to their move values.

3.2. Knowledge at non-root nodes

Since speed is of essence for simulation games, we use a more condensed and simplified set of move generators to provide move values for successor move ordering in the tree. First we use the surrounding index vales [11] plus the residual move value from the root inversely proportioned to the node depth as a default. We then adjust values for local patterns around the last move and second to last move. Finally we add major values to moves for capturing/escaping blocks with one or two liberties using speedy pseudo ladders [10]. Again, successor moves are ordered by the final move values.

4. Dynamic parameter randomization

Rapid Action Value Estimation (RAVE) and prior knowledge have been incorporated into the UCT algorithm [6,18]) to further improve the playing strength of Go programs. There are several key parameters in the UCT algorithm incorporating RAVE and prior knowledge that needs to be tuned for the program to perform well [4,12].

These key parameters are identified in Section 4.1. The methods used by GI initially to "optimize" their values are discussed in Section 4.2. The author has recently 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 set of parameter values tried before. This technique is presented along with experimental results on 19×19, 13×13, and 9×9 Go against GnuGo 3.8 Level 10 in Section 4.3.

4.1. Key parameters of MCTS

The UCB1 formula is given as

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

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 balance exploitation vs. exploration – our first key parameter to be tuned. Within the MC search tree, the UCT algorithm advances from the parent node to the child node maximizing the score s_i. A progressive widening technique [5,15] is used to control the branching of the tree.

GI uses RAVE as described in [18]. Assume child i has a RAVE UCB1 score t_i. Then, the maximum linear combination score of t_i and s_i will be used to select the child node to advance:

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

where $b = \sqrt{\frac{k}{3n+k}}$ is a constant between 0 and 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 for child 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 $u_{\max} = \max\{u_i | i = 1, 2, \ldots, \#\text{Children}\}$. 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 MC search tree with many traditional GI knowledge routines used at the root node. So a separate parameter k_0 is used for the root and parameter k_1 for all non-root nodes. Finally, since RAVE has a 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 than h, RAVE is not used at that node when selecting the child node. Naturally, we want to tune these five key parameters c, k, k_0, k_1, and h.