Grouping Nodes for Monte-Carlo Tree Search

Jahn-Takeshi Saito, Mark H.M. Winands,
Jos W.H.M. Uiterwijk, and H. Jaap van den Herik

MICC-IKAT
Universiteit Maastricht
P.O. Box 616, 6200 MD Maastricht
The Netherlands
{j.saito,m.winands,uiterwijk,herik}@micc.unimaas.nl

Abstract. Recently, Monte-Carlo Tree Search (MCTS) has substantially contributed to the field of computer Go. So far, in standard MCTS there is only one type of node: every node of the tree represents a single move. Instead of maintaining only this type of node, we propose a second type of node representing groups of moves. Thus, the tree may contain move nodes and group nodes. This article documents how such group nodes can be utilized for including domain knowledge to MCTS. Furthermore, we present a technique, called Alternating-Layer UCT, for managing move nodes and group nodes in a tree with alternating layers of move nodes and group nodes. A self-play experiment demonstrates that group nodes can improve the playing strength of a MCTS program.

1 Introduction

In the last fifteen years, Monte-Carlo methods have led to strong computer Go programs. The short history of Monte-Carlo based Go programs underwent two phases. In the first phase, Monte-Carlo was introduced [2, 3] as an evaluation function. A Monte-Carlo evaluation simply estimates the value of a game state S by statistically analyzing random games starting from S. In the second phase the focus of research shifted to Monte-Carlo Tree Search (MCTS, [4, 7, 8]).

Until now, all existing work on MCTS employs tree nodes to represent only a single move. The contribution of this work is to introduce nodes representing groups of moves to MCTS. This article presents a technique extending MCTS for managing group nodes. It enables the application of domain knowledge in MCTS in a natural way.

The remainder of this article is organized as follows. Section 2 explains MCTS and a specific move-selection function for MCTS called UCT. Section 3 introduces the concept of group nodes for MCTS and Alternating-Layer UCT. Section 4 describes an experiment comparing standard UCT and Alternating-Layer UCT and discusses the results. Finally, Section 5 gives a conclusion and an outlook to future research.

2 Monte-Carlo Tree Search

This section first describes Monte-Carlo Tree Search (MCTS) in general (Subsection 2.1). In Subsection 2.2 a specific move-selection function for MCTS, called UCT, is explained.

2.1 The Monte-Carlo Tree Search Framework

MCTS constitutes a further development of the Monte-Carlo evaluation. It provides a tree-search framework for employing Monte-Carlo evaluations at the leaf nodes of a particular search tree.

MCTS constitutes a family of tree-search algorithms applicable to the domain of board games [4, 7, 8]. In general, MCTS repeatedly applies a best-first search at the top level. Monte-Carlo sampling is used as an evaluation function at leaf nodes. The results of previous Monte-Carlo evaluations are used for developing the search tree. MCTS consists of four stages [5]. During each iteration, four stages are consecutively applied:

(1) *move-selection;*
(2) *expansion;*
(3) *leaf-node evaluation;*
(4) *back-propagation.*

Each node N in the tree contains at least three different tokens of information: (i) a move representing the game-state transition associated with this node, (ii) the number $t(N_i)$ of times the node has been played during all previous iterations, and (iii) a value $v(N)$ representing an estimate of the node's game value. The search tree is held in memory. Before the first iteration, the tree consists only of the root node. While applying the four stages successively in each iteration, the tree grows gradually. The four stages of the iteration work as follows.

(1) The move selection determines a path from the root to a leaf node. This path is gradually developed. At each node, starting with the root node, the best successor node is selected by applying a move-selection function to all child nodes. Then, the same procedure is applied to the selected child node. This procedure is repeated until a leaf node is reached.
(2) After the leaf node is reached, it is decided whether this node will be expanded by storing some of its children in memory. The simplest rule, proposed by Coulom [7], is to expand one node per evaluation. The node expanded corresponds to the first position encountered that was not stored yet.
(3) A Monte-Carlo evaluation (also called *playout* or *simulation*) is applied to the leaf node. Monte-Carlo evaluation is the strategic task that selects moves in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves. The main idea is to play more reasonable moves by using patterns, capture considerations, and proximity to the last move.

(4) During the back-propagation stage, the result of the leaf node is back-propagated through the path created in the move-selection stage. For each node in the path back to the root, the node's game values are updated according to the updating function.[1] After the root node has been updated, this stage and the iteration are completed.

As a consequence of altering the values of the nodes on the path, the move selection of the next iteration is influenced. The various MCTS algorithms proposed in the literature differ in their move-selection functions and update functions. The following subsection briefly describes a specific move-selection function called UCT.

2.2 UCT

Kocsis and Szepesvári [8] introduced the move-selection function UCT. It was fruitfully applied in top-level Go programs such as Crazy Stone, Mogo, and Mango. These programs entered in the loop of various KGS tournaments successfully.

Given a node N with children N_i, the move-selection function of UCT chooses the child node N_i which maximizes[2] the following criterion.

$$\overline{X}_i \; + \; C \; \sqrt{\frac{ln \;\; t(N)}{t(N)_i}} \;\; . \tag{1}$$

This criterion takes into account the number of times $t(N)$ that node N was played in previous iterations, the number of times $t(N_i)$ the child N_i was selected in previous iterations, and the average evaluation $\overline{X}_i$ of the child node N_i.

The weighted square-root term in Equation 1 describes an upper confidence bound for the average game value. This value is assumed to be normally distributed among the evaluations selected during all iterations. Each iteration passing through N represents a random experiment influencing the estimate of the mean parameter $\overline{X}$ of this distribution.

The constant C controls the balance between exploration and exploitation [5]. It prescribes how often a move represented by child node N_i with high confidence (large $t(N_i)$) and good average game value (large $\overline{X}_i$) is preferred to a move with lower confidence or lower average game value.

The update function used together with UCT sets the value $\overline{X}$ of a node N to the average of all the values of the previously selected children including the latest selected child's value $\overline{X}_i$.

3 Grouping Nodes

We describe the concept of group nodes in this section. Subsection 3.1 outlines related work in which domain knowledge is applied to form groups of moves.

[1] Coulom [7] refers to update function as back-propagation operator.
[2] In Min nodes the roles are reversed and the criterion is minimized.

Subsection 3.2 provides a detailed account of the features of group nodes. Subsection 3.3 presents an extension of the UCT which is able to manage group nodes.

3.1 Related Work

The idea of categorizing moves in games is not new. The work of Tsuruoka *et al.* [10], to name only one example, applies domain knowledge in the game of Shogi to categorize follow-up moves for a given game state according to simple features. Depending on which categories the move falls into, the amount of resources (mainly search depth) allocated to searching this particular move is fixed. In the domain of computer Go, simple features have been applied for various tasks related to search. Features have been used to assign probabilities to playing certain moves in random games or to eliminate unpromising moves.

The concept of *Metapositions* was established by Sakuta [9] to represent sets of possible game states. Metapositions are employed as nodes in game-tree search. Moreover, Metapositions were found to be suitable to represent sets of game states consistent with observations made in the imperfect information game Kriegspiel [6].

Set Pruning, introduced by Bouzy [1], is a technique, first developed for computer Go. It brings together two items: first, the idea of grouping moves according features, and second, the aspect using sets of moves in a search tree as found in Metapositions. In Set Pruning two categories of moves are maintained: moves labelled as *Good* moves and moves labelled as *Bad* moves. Then, Monte-Carlo evaluation is applied to moves in both sets and a common statistic is maintained for all moves belonging to the same category. The technique proposed here can be viewed as a twofold extension of this technique. The first extension effects the number of move groups which we suggest to generalize to more than only two. The second extension this article proposes to go beyond Set Pruning concerns the framework groups are used in: instead of considering only Monte-Carlo evaluation, we propose to apply groups to MCTS. This second extension implies that the statistics are no longer kept in only one place (a node to be evaluated). Instead, the statistics alter many nodes in the search tree. Thus, a mechanism is required to adopt the idea of groups of moves to MCTS. This mechanism is facilitated by group nodes which are the subject of the following subsection.

3.2 Group Nodes

In plain MCTS, all nodes of the tree represent simple moves. We call them *move nodes*. To facilitate the use of domain knowledge we introduce the concept of *group nodes*. A group node represents a group of moves. Domain knowledge in the form of features is used to group the nodes.

Given a move represented by a node N, we suggest to partition the set of its child nodes N_i. We call each partition a group. The partitioning is achieved by

assigning each N_i to exactly one group according to whether they meet certain pairwise exclusive features.

Group nodes can function in a MCTS framework only if they provide the basic features MCTS requires a node to have (cf. Subsection 2.1). Thus, for each group node G three features are recorded: (i) a move representing the game state transition associated with this move, (ii) the number $t(G)$ of times it was visited in previous iterations, and (iii) a game value $\overline{X}_G$. Since a group node does not represent a single move, the game state is not altered whenever a group node is met during an iteration. The game value of a group node is set to represent the average game value of all nodes belonging to the group.

3.3 Alternating-Layer UCT

Alternating-Layer UCT is a technique which manages group nodes in the UCT framework [8]. The Alternating-Layer UCT maintains two types of nodes in the MCTS tree in parallel: (1) move nodes, and (2) group nodes.

When a move node N is expanded, it is expanded in two steps. First, N is expanded into group nodes (first layer). Second, each of the newly expanded group nodes is expanded into move nodes (second layer).

In the first step, all successor nodes are grouped according to features (cf. Subsection 3.1). Each group is represented by a group node G_i. These newly created group nodes become the new children of N. In the second step, each G_i is expanded. For each member move of G_i a new move node G_{ij} is created as child node of G_i.

After completing the two steps of expanding a move node, all new leaf nodes are move nodes. Because the root is a move node, all leaf nodes are move nodes at the end of each iteration. Furthermore, the structure of the search tree is such that move nodes and group nodes form alternating layers (leading to the proposed naming).

In standard implementations of the UCT algorithm, all follow-up moves of a leaf node L are chosen randomly until L has been played a critical number x of times ($t(L) > x$). Analogously, we suggest choosing the follow-up move of a leaf node L among its succeeding groups with equal probability while $t(L) > x$. This equidistribution results in an implicit weighting of moves, because the number of members may vary between the groups. We consider, e.g., that two groups G_1 and G_2 are given with n_1 or n_2 moves, respectively. Furthermore, $n_1 \gg n_2$. If both groups are selected equally often, a move which is member of G_1 is less likely to be selected than a move which is member of G_2.

4 Experiment

In this section, we test the increase of playing strength gained by Alternating-Layer UCT. Subsection 4.1 describes the setup of a self-play experiment. Subsection 4.2 presents and comments the results obtained.

4.1 Set-up

Standard UCT and the Alternating-Layer UCT were implemented for the domain of computer Go. We refer to the resulting programs as *STD* and *AL*, respectively.

Both implementations were compared in a series of 1,000 games of direct play on 9×9 boards with 5.5 points Komi (500 games for each program playing Black, respectively). The time setting was one second per move. The Monte-Carlo sampling used in both implementations is based on small hand-tuned patterns and reaches a speed of about 22,000 sampled games per second on the given hardware (cf. below). The C parameter (cf. Equation 1) was set to 1.9, determined by trial-and-error, for *STD* and *AL*. The x parameter (cf. 3.3) for describing the threshold for expanding moves was set to the number of children of the node to be expanded.

The number of iterations available per move for *STD* is 30,000. To compensate for the grouping overhead of alternating-layer UCT, *AL* was allowed only 10,000 iterations per move.

Two features were used for grouping in *AL* resulting in three types of group nodes. The first feature is proximity to the last move. The proximity chosen is the Manhattan distance of 2. (For the empty board, the last move is set to be the center.) The second feature determines whether a move is a border move. The three resulting groups are the following.

Group 1	All legal moves in the proximity of the last move.
Group 2	All legal moves on the border of the game board which do not belong to group 1.
Group 3	All legal moves which do not belong to either group 1 or group 2.

The experiment was conducted on a Quad-Opteron server with 3.4 GHz Opteron processors and 32 GB of memory running a well-known Linux distribution. Both algorithms are implemented in C++.

4.2 Results

Playing the 1,000 games required a total playing time of ca. 30 hours. Of these, about two thirds were required by *STD* and the remainder by *AL*. Of 1,000 games *AL* won 838 and *STD* won 162.

A qualitative analysis of several dozen sample games shows that *AL* plays more consistently than *STD*. The program seems to beat *STD* because of its tactical superiority. This suggests that *AL* takes advantage of focusing samples on local positions more often than *STD*. In contrast, *AL* seems to shift the focus and play non-local moves with a better timing than *STD*. *AL* never plays border moves and does not seem to invest much effort on testing such moves during the first 50 moves, whereas *STD* occasionally plays border moves.

The result of the experiment clearly shows that Alternating-Layer UCT outperforms palin UCT. We may conclude that group nodes can serve to integrate

domain knowledge in the MCTS framework successfully. Moreover, it may be inferred that the UCT framework is sufficiently flexible to choose the right group nodes, and that providing group nodes significantly improves the ability of the program to focus on promising branches more quickly.

The computational overhead for grouping nodes is outweighed by the benefits of narrowing down the search space in the experiment. While this is true for the three computationally cheap feature groups tested in the experiment, it remains to be seen how well this approach scales to a larger number of groups.

5 Conclusion and Future Research

In this article we introduced the concept of *group nodes* for MCTS. Alternating-Layer UCT was proposed as a technique for adopting group nodes for MCTS. A self-play experiment showed that Alternating-Layer UCT outperformed standard UCT. Based on the outcome of the experiment we may tentatively conclude that the proposed approach can incorporate domain-specific knowledge in MCTS successfully.

Future work will address the following six items. First, in order to examine whether the results found in this work generalize to deeper search, the programs will be tested with a more generous time setting. Second, the number of groups will be increased using more refined features, e.g., by using a move predictor. Third, the weighting of the probabilities assigned to group nodes will be examined more closely. Fourth, other algorithms for including group nodes in the MCTS framework could be devised. Whereas the Alternating-Level UCT straightforwardly adds group nodes after every node expansion, it might prove more useful to expand group nodes only for certain move nodes. This might reduce the computational cost required for grouping nodes. Similarly, group nodes could be allowed to have group nodes as their child nodes. Fifth, the grouping techniques will be compared to other means of incorporating domain knowledge in UCT. Sixth, the new technique will be implemented in a tournament program, e.g., MANGO.

Acknowledgements

We are indepted to Ulaş Türkmen and Guillaume Chaslot for their valuable feedback. This work is financed by the Dutch Organization for Scientific Research, NWO, as part of the project GO FOR GO, grant number 612.066.409.

References

1. Bouzy, B.: Move Pruning Techniques for Monte-Carlo Go. In van den Herik, H.J., Hsu, S.C., sheng Hsu, T., Donkers, J.H., eds.: Advances in Computer Games (ACG 11). Volume 4250 of LNCS., Springer-Verlag, Berlin (2006) 104–119

2. Bouzy, B., Helmstetter, B.: Monte Carlo Developments. In van den Herik, H.J., Iida, H., Heinz, E.A., eds.: Proceedings of the Advances in Computer Games Conference (ACG 10), Kluwer Academic (2003) 159–174
3. Brügmann, B.: Monte Carlo Go. White paper (1993) http://www.ideanest.com/vegos/MonteCarloGo.pdf.
4. Chaslot, G., Saito, J.T., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Monte-Carlo Strategies for Computer Go. In Schobbens, P.Y., Vanhoof, W., Schwanen, G., eds.: Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium. (2006) 83–91
5. Chaslot, G., Winands, M.H.M., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Progressive Strategies for Monte-Carlo Tree Search. (2007) Submitted to JCIS 2007.
6. Ciancarini, P., Favini, G.P.: A Program to Play Kriegspiel. ICGA Journal **30**(1) (2007) 3–24
7. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In Ciancarini, P., van den Herik, H.J., Donkers, J.H., eds.: Proceedings of the Fifth Computers and Games Conference. LNCS, Springer-Verlag, Berlin (2007) 12 pages, in print.
8. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In Fürnkranz, J., Scheffer, T., Spiliopoulou, M., eds.: Proceedings of the EMCL 2006. Volume 4212 of LNLCS., Springer-Verlag, Berlin (2006) 282–293
9. Sakuta, M., Iida, H.: Solving Kriegspiel-like Problems: Exploiting a Transposition Table. ICGA Journal **23**(4) (2000) 218–229
10. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-Tree Search Algorithm Based on Realization Probability. ICGA Journal **25**(3) (2002) 145–152