Parallelized Game Tree Generation for Go

Cindy Moline, Zachary Robertson, Jacob Osterhout
Department of Electrical Engineering and Computer Science
University of Central Florida, Orlando, Floirda 32816–8005

***Abstract*—Go is an exceedingly difficult game for an artificial intelligence to conquer; this is because generating a game tree is difficult, the branching factor is too large for most systems to handle. By splitting the work of generating a game tree, that spans all potential moves available at a particular moment in the game, among all the processors available to the system, we reduce the amount of time waiting for a move to occur. We believe that our implementation of the randomized best first minimax tree will outperform the randomized Monte Carlo tree in terms of search times, since its use of heuristics should assist is pruning more nodes from the search tree. The two algorithms explored in our project indicate...**

I. Introduction

The oriental game of Go is a two player, non-chance, combinatorial game with perfect information. On regulation sized 19 x 19 game boards there are 361 positions for game pieces to be placed. Each player has the option of placing a game piece, called a stone, or passing on their turn. The game ends when there are no available moves to be played or if each player passes on their turn in succession of one another. After a game is complete we enter the scoring phase, the winner will be chosen based on the number of opponent stones captured added to the total area on that board that is surrounded by your stones.

Placed stones can be captured if all of the adjacent intersections are occupied by opponent stones. These four adjacent intersections decide the fate of a stone. Placement of a stone adjacent to one of your others creates a group, this group has a greater number of adjacent intersections and is thus harder to capture. Since stones can be removed from the board, placement of a new stone is forbidden if it recreates a previous board state, this rule prevents an infinite loop of capturing stones.

A. Complexity

Each location on the board has three potential states, empty, black or white. On a game board with N locations there are a total of 3^N possible board states. Tromp and Farneback showed that on a standard board only about 1.2% of board positions are legal, using this fact we can estimate the number of legal board states for a 19 x 19 board as

$$3^{361} * 0.012 = 2.08\ldots * 10^{170}$$

This number demonstrates the vast amount of variation inherent in Go, also that our game tree will obviously not be able to explore every possible variation of board combinations. Therefore we must optimize our tree with pruning and only expand those nodes which we feel will lead to a positive outcome. Standard alpha-beta techniques identify moves that are so good for the current player that best-play by the opponent should not have allowed it. Another type of pruning very relevant to Go, due to the option to pass on your turn, is null-move pruning in which the computer imagines that it has given up its turn and imitates a shallow Alpha Beta search. If this search would have produced a cutoff, then we do not need to expand this entire tree because we know it is now very likely that actually making a move also produces a cutoff.

Victor Allis notes that typical games between expert players last about 150 turns each with an average move complexity of 250 choices, we aim for our game tree to have a similar branching factor. Thus we expect that each node in our game tree will have around 250 child nodes, with more initial children at the start of the game and very few children near the end. Games with a high branching factor make classic search algorithms like minimax extremely costly.

B. Game Trees

Monte Carlo Tree Search (MCTS) can be used in any combinatorial game, it combines the generality of random simulation with the precision of a tree search. It does not require any strategic knowledge of the game, just its legal moves and ending conditions. Games with a large branching factor are suitable for a Monte Carlo search because they have less space complexity than a breadth or depth first approach. Due to the random nature of the algorithm there are situations where finding a reasonable move is difficult, as the vital nodes may not have been visited enough times to give desirable results. This kind of search can be quite time consuming as many nodes must be expanded in order to produce strategic moves, techniques such as parallelism and conjunction with domain specific knowledge can significantly improve the speed.

Talk about hash maps for shared board memory.

II. Implementation

Relate the ideas presented in the complexity section. Talk about why we choose to implement these two algorithms.

A. RBFM

Our implementation of a parallel minimax search tree comes in the form of a Randomized Best First Minimax Search

(RBFM). Shoham and Toledo introduce this efficient selective search algorithm which like MCTS attempts to focus on relevant nodes. RBFM associates a minimax value and a probability distribution to each node, when the algorithm expands a leaf node in the tree it updates the relative parent nodes minimax value. Each leaf is chosen based on a random walk from the root of the tree, given more precedence to those nodes with a higher probability distribution. The random nature of this algorithm creates parallelism since we can assign a different random walk to each processor. A regular best first search would only expand a leaf node if it had to highest current heuristic, this introduces a problem as the initial best move does not always lead to a favorable outcome for the entire game. RBFM fixes this problem by descending to all children with some probability. This method of expanding nodes seems to fit the randomization that is preferable in Go as well as outperforming the alpha-beta method of selecting leaf nodes.

All parallel game tree algorithms will speculate at the outcome of a series of moves in order to achieve asynchronous parallelism. The only time when contention happens in RBFM is if two or more threads are attempting to update the value of a single node, once there are many nodes on the tree this becomes less of a concern. When a thread finishes expanding a node path it backs up the score and starts over again at the root, this way the processors remain busy. Using our shared-memory model implementation all processes access a single search tree, these processes will lock nodes when they traverse the tree but since the tree only grows without transforming there may only need to be active lock per process.

B. Monte Carlo

III. Results

Talk about execution times and present charts

IV. Conclusion

Summary of results and ideas for future tests.

References

[1] Y. Shoham, S. Toledo, *Parallel Randomized Best-First Minimax Search*, Artificial Intelligence 137 (2002) 165–196

[2] J. Tromp, G. Farneback, *Combinatorics of Go*, http://tromp.github.io/go/legal.html (2007)

[3] V. Allis, *Searching for Solutions in Games and Artificial Intelligence*, http://fragrieu.free.fr/SearchingForSolutions.pdf (1994)