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 could span 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 Monte-Carlo tree in terms of search times, since its use of heuristics should assist in 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. Although the rules of Go are relatively simple, the dynamic nature of the game allows for much variation.

Each player, black and white, has an infinite amount of stones of that player's respective color. Black player will always be first to place a stone, the adjacent intersections are the most important positions on the board for a stone after it has been placed. Each stone placed at the center of the board initially has four adjacent intersections which will decide the fate of the 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. If stones of the opponents color occupy all of the adjacent intersections of your stone or group of stones than those stones are captured by the opponent and removed from the board. Since stones can be removed, placement of a new stone is forbidden if it recreates a previous board state, this rule prevents an infinite loop of capturing stones. Also suicide in Go is illegal, if placement of your stone would result in one or more stones of your color being removed then that move is not allowed.

A. Complexity

Each location on the board has only 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 if we choose to implement a standard game tree, we would need to optimize it with pruning and only expand those nodes which we feel will lead to a positive outcome. Standard alpha-beta pruning 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 results in a cut-off, 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 cut-off.

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

B. Game Trees

The traditional approach of combining alpha-beta search with a heuristic position evaluator is very difficult to do for the game of Go. Creation of an adequate static position evaluator is one of the main hurdles for writing an artificial player for Go. Even if we reduce the standard board size to 9 x 9, which in terms of complexity is inferior to Chess, the traditional minimax game tree approach does not create a strong player [4]. One of the only times the computer player is aware that a move is good is when the opponent is about to have some of their stones captured. Most of the strategic positions in Go are dynamic in nature which lends itself to the randomized branch selection inherent in a Monte-Carlo evaluation. Since Go is deterministic we can generate actions at random until

a terminal board state is reached, some if not most of these random play-outs will result in a better outcome than trying to abide by a heuristic that leads to predictable play.

Monte-Carlo Tree Search (MCTS) can be used in any combinatorial game and is based on many play-outs, the final game result of each play-out is then used to weigh the nodes in the game tree so that those nodes which lead to more favourable results are chosen during actual game play. 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 much less space complexity than a breadth or depth first approach, since MCTS does not attempt to expand every leaf of the game tree. 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 between threads.

II. Implementation

Since we are well aware of the computational difficulty inherent in games where the sheer number of potential moves can overwhelm an artificial intelligence's ability to determine the correct course of action, it follows that the more nodes we are able to compute in a time-frame that humans find reasonable to take a turn in a game, then the more accurate our program will be at playing said game. This is why the next step in developing an artificial intelligence that is proficient at the game of Go is dependent on the processing power of the computer running the artificial intelligence software. Expanding our game tree concurrently will allow us to explore more nodes than we could in in same time sequentially, ideally with a lock-free implementation we would expect a speed-up equal to the number of threads that our implementation has availability to. This speed-up will obviously not be achieved as our solutions cannot completely run concurrently, instead they will implement a various locks as needed to ensure that redundant work is not done and data integrity is upheld. Both of our chosen implementations have best-first search methodology, which means we will not actively try to expand all nodes but instead choose nodes to expand based upon some reasoning. Traditional game theory suggests that most games should be winnable using a minimax tree, although we understand the difficulty of the implementation in Go, with a high level of parallelization we believe that our implementation can overcome the difficulty of intermediate evaluation and produce a bot that exhibits strategic game play throughout the game with little to no possibility of skipping the most desirable move at a specific board position. As noted by Chaslot, Winands and Jaap van den Herik, currently a Monte-Carlo tree search program named *Crazy Stone* holds the highest rating amongst all the other programs recognized on a particular Go site as well as won the second Computer Go University of Electro-Communications cup, since this implementation is state-of-the-art it seems important for us to parallelize it as well.

A. RBFM

Our implementation of a parallel minimax tree search comes in the form of a Randomized Best-First Minimax Search (RBFM). Shoham and Toledo introduce this efficient selective search algorithm which 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 the highest current heuristic, this introduces a problem as the initial best move does not always lead to a favourable outcome for the entire game. RBFM fixes this problem by expanding all children with some probability down to a specified depth. 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 one active lock per process.

Our approach iteratively expands a leaf node based upon a probability distribution and propagates the score up to the top of the tree. There will be some cases where expanding a node and updating its minimax score cannot change the decision at the root, in this case RBFM avoids expanding these nodes. One thread of execution will always be exploring the moves that spawn from the best play but others will continue to search for moves in leaf nodes with less desirable initial results, in this way we are not guaranteed that our algorithm will always choose the greedy move.

B. Monte-Carlo

Research indicated that there were several different valid techniques for accomplishing concurrency. All of these stemmed from the same four steps which are the basis of a MCTS implementation. The first of which is *selection*, during this phase a leaf node is chosen at random from the

root of our game tree. Second is the *expansion* phase which examines the leaf node chosen in step one and determines if it is represents a game board where the maximum number of moves has been reached. Assuming that the leaf node selected does not represent a completed game board, we proceed to the *simulation* stage, where random plays occur until the maximum number of game moves have been played. Finally we score the board and take note of the wins and loses, during the *backpropagation* step we update the corresponding nodes from the leaf node chosen in step one to the root of our game tree.

Our implementation of a Monte-Carlo Tree Search is mostly based off tree parallelization from G.M.J-B Chaslot. This method uses a shared game tree in which each thread is simultaneously playing a random game and expanding that specific branch of the game tree. In designing our implementation we had to consider whether to focus on exploration or exploitation. If several threads start from the root of our game tree at the same time, there is a chance that they will traverse the tree in a similar fashion. Since heuristic values are hard to compute for intermediate game board states in Go, we decided to focus our design on exploration. Thus when a thread plays a game it only looks to explore nodes from the current root node's children only, once an expansion reaches a game board state with only one move left we explore all possible moves before propagating our wins and losses back to the root, this is our exploitation step and helps us to get a better idea of the destination's value. It is worth noting that since we explore all leaf nodes final scores we only need to lock the leaf node's parent in order to reduce redundant work done by other threads, thus terminal nodes will only be visited once by one thread. Since multiple threads will still have the potential to explore games from a specific child of the root node, we use atomic integers to represent the number of wins and loses for a node. After a time period of thirty seconds has passed, we will not continue to expand random child nodes, once all threads have terminated we look at the difference between the wins and loses for each child node and choose the one that is most fit. After a turn is complete our game tree trims all non-selected children and after the opponents move, will choose this node as the new root node.

III. Results

Talk about complexity of 9 x 9 search space. Talk about execution times and present charts.

IV. Conclusion

Summary of results and ideas for future tests.

Appendix A
Challenges

1) Implementation of a 2 parallel game trees in Java from pseudo-code.
2) Grouping of neighbouring stones on our board for easier position evaluation.
3) Determining which parallelized algorithm has the greatest chance to be efficient at a deterministic, zero-sum, combinatorial, perfect information game.
4) Getting a minimax algorithm to play efficiently on a game where static evaluation of board position is most of the time useless.

Appendix B
Completed Tasks

1) Wrote a Randomized Best First Minimax game tree for Go.
2) Wrote a Java Go board class which only allows legal moves and provides a score at every stage of the game, which is used as our static position evaluator.
3) Located many resources to make sure our implementation is cutting-edge.

Appendix C
Remaining Tasks

1) Parallelize our implementation of RBFM.
2) Writing a Monte-Carlo Tree Search to compare our RBFM implementation to.
3) Parallelize our implementation of Monte-Carlo.
4) Testing with multiple threads.
5) Testing with a 9x9 Go board that is initially empty.
6) Testing with a 19x19 Go board that has a specific set up of initial stones and a correct solution.
7) Finishing formalizing our research for the final paper.

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)

[4] R. Coulom. *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search*, 5th International Conference on Computer and Games, May 2006, Turin, Italy. (2006)