Othello

Noam Hirschorn (nyh2111) & Dan Ivanovich (dmi2115)

Overview

We aim to optimize the performance of an Othello AI opponent that uses the minimax decision-making algorithm. We will attempt to do this by optimally parallelizing the minimax decision tree. We will explore various strategies for parallelizing the minimax tree, particularly focusing on balancing workload distribution across threads while preserving the benefits of pruning.

Background

Othello, a board game derived from Reversi, is played on an 8x8 grid with a fixed starting layout. Throughout the game, players take turns placing disks of their side's color onto empty spaces. After a player places a disk, any disks of the opponent's color that lie between the one that was just placed and another one of the same color are flipped. At the end of the game (when the board is filled), whichever player has the majority of disks showing their color wins the game. The game can end in a tie if both players have the same number of disks when there are no remaining moves to be made.

Minimax (AKA minmax) is a back-tracking decision-making algorithm that is often used to determine the optimal move in turn-based games (eg. chess, checkers, tic-tac-toe, or othello). It assumes that the opponent will play optimally and recursively evaluates all future states of the board, then selects the move that eventually leads to the most favorable outcome. By developing an algorithm to assign a score to each board state that reflects which player holds the most advantageous position, one can use minimax to create an intelligent opponent that optimally plays towards a winning result. While minimax is a great framework for decision-making, it is limited by the exponential growth of possible game states as search depth increases, making deeper searches computationally expensive, particularly for games like Othello with high branching factors. To address this, practical implementations often limit the search depth and rely on heuristic evaluation functions to approximate the desirability of non-terminal board states. In the case of Othello, an example simple heuristic is the number of stones of the chosen color in a given state to estimate desirability.

Architecture of Code

The initial codebase allows the user to play othello against an AI opponent in a command-line GUI. The AI opponent picks moves using a minimax tree, capped at a search depth of 3 moves ahead. We have already modified the code so that the user can input a game board and desired maximum search depth, and the program will output the agent's next move (the optimal move, as determined by minimax). These modifications will allow us to input a series of different starting game states in order to easily test our results across different positions. This is important since randomness is difficult to achieve in Haskell, so we otherwise risk optimizing on one specific set of calculations which then becomes slower when the search tree has different numbers. We also are considering adding alpha-beta pruning to the minimaxing algorithm, which would significantly reduce the number of paths to be evaluated by

minimax by skipping branches of board states that would not affect the algorithm's final decision. One key note about Othello is that each game has 64 moves, so the search tree should be balanced. This may make it easier to figure out the optimal search depth.

Architecture of Parallelization

We plan on parallelizing the first few rows of the minimax search tree, and then leaving the bottom parts to be sequential. This is desirable as the bottom layers should have enough work in them to avoid underutilizing threads. Additionally, this will allow each thread to utilize the benefits of alpha beta pruning (if we enable it) which relies on sequential knowledge and therefore can get broken up if a process is overly parallelized. We will need to parallelize the top nodes of the tree so each subtree can be processed in parallel in each thread/core. The main optimizations we hope to investigate are:

a) What is the minimum amount of work (i.e. size of subtree) needed for a thread to be sufficiently utilized and to allow for some of the benefits of pruning?

b) How many cores/threads should be utilized (depending on the number of cores on the machine and the depth of the minimax tree)?

We might also want to check:

c) Is it better to use sequential logic at the top of the decision tree, and only parallelize in the middle (so then the middle of each subtree can be parallelized with all threads, before they return and to then repeat the process by the next subtree)? Or is it better to parallelize from the very top of the decision tree (so each thread just deals with one subtree, and when they all finish the program overall is nearly done)? A key point to emphasize is that since it is Othello, we can assume the search tree will be balanced, which makes the latter option more viable.

d) How is the %speedup affected if we turn on/off alpha-beta pruning? (the program may speed up overall, but the %speedup may be lower if the program is more parallelized, since it prevents pruning). Overall, it may just change the answers we find for parts a-c.

References

- Initial codebase: https://arttuys.fi/coding/2022/05/othello-haskell/