Go! Go! Haskell Curry!

Matthew Retchin (mhr2145)

November 2022

1 Overview

I will implement the classic minimax search algorithm with alpha-beta pruning in Haskell, applying it to the ancient Chinese board game of Go. I will parallelize minimax to improve its performance.

2 Background

Go is a turn-based abstract strategy game that has been played for thousands of years. Until very recently, it was considered extremely difficult to beat a professional human Go player with a computer algorithm. However, DeepMind's AlphaGo was able to combine a version of minimax search with a neural network in 2016 to beat one of history's best Go players. [1]

While I won't be using a neural network for my project, I will be using minimax search. Minimax search is a strategy for adversarial turn-based games like Go that relies on the minimax decision rule. As we minimize our loss, we assume that our opponent's goal is to maximize our loss. And we assume that our opponent operates under the assumption that we are minimizing our loss. And so on; indeed, minimax is a recursive algorithm. Each possible move/board state exists within a tree, and our objective is to search this tree until we reach the leaves (completed games) with the minimum loss. If we can't reach the leaves in a reasonable amount of time, which often happens for games with a high branch factor like Go, then we use a heuristic on the incomplete board state to determine the state's value.

As mentioned, Go has an exceedingly high branch factor in its search tree, so we will try boards with dimensions ranging from 6x6 to 9x9 to keep the computation feasible and neural nets unnecessary. The write-up will describe what dimensions are most feasible.

Below is imperative Pythonic pseudocode for the sequential version of minimax (from Wikipedia):

```
def alphabeta(node, depth, alpha, beta, is_max):
    if depth == 0 or node is terminal:
        return heuristic_value(node)
```

```
    if is_max:
        value = -infinity
        for child in node.children:
            value = max(value, alphabeta(child, depth - 1, alpha, beta, False))
            if value >= beta:
                break
            alpha = max(alpha, value)
        return value
    else:
        value = infinity
        for child in node.children:
            value -= min(value, alphabeta(child, depth - 1, alpha, beta, True))
            if value <= alpha:
                break
            beta = min(beta, value)
        return value

alphabeta(root, depth, -infinity, infinity, True) # initial call like so
```

As is apparent by the sequential for-loops, what's tricky about parallelizing alpha-beta pruned minimax is that it's fundamentally a sequential algorithm. You save work by skipping branches of the search tree you've already determined aren't worth checking. One answer is to parallelize vanilla minimax (without pruning) up to a certain depth in the search tree, after which we switch to a sequential version and introduce alpha-beta pruning. It's rather like the Fibonnaci example in class where you use parallelism until a certain recursion depth, after which you switch to a vanilla sequential version.

3 Objectives

- Implement a Go game engine that determines the legality of moves and maintains a board state data structure. If I can introduce any parallelism into the game engine to speed it up, then I will, though I don't anticipate the opportunity arising.
- Provide an graphical, quantitative Threadscope-based analysis comparing parallel minimax with sequential minimax to see how much of a speedup parallelism provides.
- Implement a textual user interface (TUI) so a user can play against the AI. This is not a core feature, so I will likely drop it if I don't have time.
- Write a few tests to sanity-check whether parallel minimax outputs the same values as sequential minimax.

4 References

[1] https://www.theverge.com/2019/11/27/20985260/ai-go-alphago-lee-se-dol-retired-deepmind-d
[2] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#Pseudocode