Gomokuku4KokoPuffs

Matthew Retchin (mhr2145)

December 2022

1 Overview

In this project, I implemented the classic minimax search algorithm with alpha-beta pruning in Haskell, applying it to the classic Japanese board game of Gomoku. I parallelize minimax to improve its performance.

Please note that some of the material in this writeup has been borrowed from my proposal, which is why some sentences herein may appear familiar to someone who has also read the proposal.

2 Background

Gomoku is a turn-based abstract strategy game that has been played for hundreds of years. Gomoku is played on a Go board, an even older game, but it has simpler rules than Go. Players take turns placing black and white stones on a grid, attempting to place five stones in a row of the same color while also preventing their opponent from doing the same. The first player is black and must place their stone in the middle of the board. So-called "overlines", which are lines longer than 5, do not win the game. The game only concludes when a row of five has been produced from either player. Lines may proceed up, down, or diagonally along the points of the grid. [1]

It is typically very difficult to beat a really good human Gomoku player with a computer algorithm due to the high branch factor of its game tree. One way to deal with this high branch factor is to employ DeepMind's approach with AlphaZero, which is to use a neural network combined with Monte Carlo (game) tree search, also known as MCTS. [2]

While I didn't use a neural network or MCTS for my project, I did use a simpler game tree search algorithm known as minimax search combined with some optimizations. Minimax search is a strategy for adversarial turn-based games like Gomoku 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 Gomoku, then I use a heuristic on the incomplete board state to determine the state's value. As we will see, the speed of heuristic has a highly significant effect on the AI's performance overall.

Below is imperative Pythonic pseudocode for the sequential version of minimax (from Wikipedia) [3]:

```
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 — this serial nature of alpha-beta pruning is what makes it an effective optimization for minimax. The solution I chose 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.

As mentioned, Gomoku has an exceedingly high branch factor in its game search tree, so to manage this branch factor, I came up with data structures uniquely suited to the game. This reduced the time taken per move and time per evaluation of the heuristic so that branch factor didn't present as an issue too much.

3 Method

I focused on speed instead of features for my project. In other words, I did not implement a way for a human to play against the AI. Instead, I just have the AI play against itself. However, to my eye, the moves the AI suggests are fairly decent, and it would probably be a fairly challenging opponent to a human player.

In my implementation, I took a great deal of inspiration from a previous years' project, Gomokururu [4]. Cleverly, they reduce the time taken by their is-terminal function (in other words, the function that determines if a game has finished) by only examining whether a 5-line can be found at the most recent stone placed on the board. I used this approach and extended its use in a further optimization.

This approach actually came in handy too with move ordering to optimize the alpha-beta minimax search algorithm. Once finding the children of a given node in the search tree, a good rule of thumb is to sort the children by using this most-recent-move heuristic before running minimax recursively on them one by one, since the alpha-beta optimization is dependent on whether we get lucky in a deeper level and reach a node that allows us to eschew searching the rest of the children. If we order the children to start with, we can perhaps increase our luck.

The data structures I used were as follows:

```haskell
data Element = Empty | Black | White deriving (Enum, Show, Eq)

data Move = Move
    { moveColor :: Element
    , movePosition :: StonePosition
    }

data Board = Board
    { matrix :: Matrix
    , blackStones :: StoneSet
    , whiteStones :: StoneSet
    , stones :: StoneSet
    , mostRecentMove :: Move
    }
```

The Element is an enum representing whether a space on the board's grid is empty or a black/white stone. The Move is a record of an Element and a StonePosition, which isn't shown but is simply a tuple of Ints. The most interesting data structures is, however, the Board. The Board is a Matrix (a vector of vectors containing Ints) and three HashSets representing the black stones, the white stones, and all the stones. Finally, the Board keeps track of its most recent move, which is used in the most-recent-move heuristic described previously.

Excluding the minimax function, my program is fast because the sets allow me to only consider the stones on the board, not the empty spaces that outnumber the stones, and the sets have near constant lookups and insertions, so any operations involving the sets have a low overhead.

These stone sets are incredibly useful because when I compute my heuristic, I can loop over the sets and have near-constant lookup to determine neighbors. In addition, determining the legality of potential moves in order to generate children of a board in the game tree is fast precisely because of the near-constant lookup.

My heuristic takes advantage of the fast neighbor-lookup by generating all possible combinations of directions (up, down, diagonal) and stones on the board. For each stone, we go in each possible direction (both forwards and reverse, since a stone could be in the middle of a line) until either a stone of a different color is reached or an empty space is reached. Along the way, we count the length of the line that is formed and associate those lengths with a range of numbers.

Smaller lengths have small numbers, while large lengths (like 5, the winning number) have huge numbers. We sum all these numbers together (being careful to make all black lines positive and white lines negative), and that sum represents the value of a given board. As we will see in the results section, this heuristic is already so fast that introducing parallelism doesn't help the speed. Below is an excerpt of the heuristic, the list comprehension generating all the combinations of black stones and possible directions (up, down, diagonal) for lines:

```
blackLines = [colorLine (pos, dir) Black | pos <- HSet.toList $
    blackStones board, dir <- halfDirections]
```

Although this one line is fairly dense, you can hopefully see in this list comprehension that I'm generating every combination of black stone position ("pos") and direction ("dir").

To aid understanding of the heuristic, I've provided some Pythonic pseudocode:

```
def color_heuristic(board, color):
    combinations = [(pos, dir) for pos in board.stones(color) for
    dir in all_directions]
    lines = map generate_line combinations
    counts = map generate_counts lines
    return sum(counts)

def heuristic(board):
    return 2*color_heuristic(board, Black) - color_heuristic(board,
     White)
```

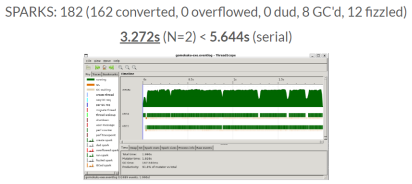
A minor detail to note is that I scale Black's count slightly when I subtract White's count from it because Black went first; Black has an advantage. As an illustrative example, suppose there is a board with four black stones and four white stones on it. It would be deceptive to claim that this board's heuristic should be 0 based on the fact that $4 - 4 = 0$. In fact, the first player to play (Black) has the advantage, because in the next move, Black could place one more stone and win. The heuristic for the board would then ideally be > 0, in that case. We wouldn't have this issue if our minimax game tree could be infinitely deep — in that case, we could eliminate the scalar term and have a truly zero-sum heuristic — but using infinite levels is an intractable approach.

3.1 Parallelism

After a lot of trial and error, I found that the optimal amount of parallelism (where sparking and managing threads didn't just introduce overhead) for this project is in parallelizing the first level of the minimax search tree, while leaving the rest of the search tree serial and using alpha-beta pruning. I did attempt to parallelize the heuristic too, even limiting the size of the buffer of sparks, but a parallel heuristic always hemorrhaged speed. Introducing parallelism into the heuristic translated into a lot of additional overhead for no demonstrable benefit.

4 Results

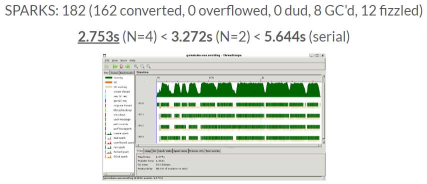
I've used the open source program Threadscope [5] to analyze how helpful parallelism is in improving the performance of my algorithm. Unfortunately, as is clear from the figures, the lion's share of the program runtime is dominated by serial processing. The part of the program that benefits from parallelize can only improve performance so much once parallelized, in other words. This truism is known as Amdahl's Law. Figures 1-3 are screenshots of the Threadscope program showing an event log for thread counts ranging from 2 to 6, confirming that, at least for this project, this truism is indeed true.



SPARKS: 182 (162 converted, 0 overflowed, 0 dud, 8 GC'd, 12 fizzled)

3.272s (N=2) < **5.644s** (serial)

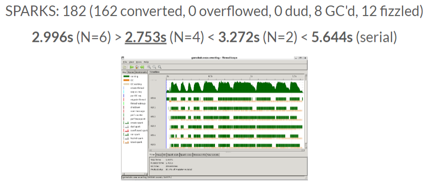
Figure 1: Threadscope with N=2 Threads



SPARKS: 182 (162 converted, 0 overflowed, 0 dud, 8 GC'd, 12 fizzled)

2.753s (N=4) < **3.272s** (N=2) < **5.644s** (serial)

Figure 2: Threadscope with N=4 Threads



SPARKS: 182 (162 converted, 0 overflowed, 0 dud, 8 GC'd, 12 fizzled)

2.996s (N=6) > **2.753s** (N=4) < **3.272s** (N=2) < **5.644s** (serial)

Figure 3: Threadscope with N=6 Threads

Figure 1 shows that using two threads does indeed increase the speed of the program, but it doesn't double the speed. As can be seen in Figure 2, increasing from two to four threads increases the speed a little more, but this trend is not linear. By the time we have six threads in Figure 3, the overhead of parallelism is hurting more than it helps. Threads are often left waiting. Amdahl's Law is upheld.

5 Tests

I ran some unit tests on various board states to ensure that my heuristic worked for lines ranging from 2 to 5, increasing in points. Importantly, Gomoku has the overline rule, where lines longer than 5 actually do not win the game and are worth 0 points. Thus, one of my tests confirmed my heuristic accounted for overlines even as it could successfully process shorter lines. I also ran tests to ensure that the children of a node in the game tree was correct, and that parallelizing the serial version of my code did not alter the output. It would be truly surprising if the latter test failed because Haskell's powerful functional purity guarantees that introducing parallelism should have no side effects.

6 Conclusion

My program's speed in its serial mode comes from data structures tailored for the domain of Gomoku, particularly the various sets, which I was able to profitably use to cheaply determine the legality of moves and also cheaply compute heuristics within the minimax algorithm.

Finally, Amdahl's Law rears its head within this project, empirically showing that parallelism is not a silver bullet. Because we can only parallelize a fraction of our code, increasing threads has no impact on the serial portion, which is really the bulk of the computation overall. This is why parallelizing had an unfortunately sublinear effect on performance.

7 Source Code

The following code was compiled/built with all warnings switched on. For further instructions, download the code and carefully follow the instructions in the README file.

The Main.hs file is as follows:

```haskell
module Main (main) where

import Lib

main :: IO ()
main = gomokuMain
```

The Lib.hs file, referenced by the Main and Spec modules, is as follows:

```haskell
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE PackageImports #-}

module Lib
    (
        -- app
        gomokuMain
        -- testing
        , Element (Empty, Black, White)
        , Board
        , showBoard
        , getChildren
        , initializeBoard
        , move
        , isTerminal
        , heuristic
        , scoreLine2
        , scoreLine3
        , scoreLine4
        , scoreLine5
        , loopSerial
        , loopPar
    ) where

import Data.List (sortBy)
import Data.Maybe
import qualified Data.HashSet as HSet
import qualified Data.Matrix as M
import Control.Parallel.Strategies
import Control.DeepSeq
import System.Environment (getArgs)
import System.Exit (die)

addTuple :: (Int, Int) -> (Int, Int) -> (Int, Int)
addTuple (a, b) (c, d) = (a + c, b + d)

multTuple :: Int -> (Int, Int) -> (Int, Int)
multTuple s (a, b) = (a*s, b*s)

generateNeighbors :: HSet.HashSet (Int, Int) -> Int -> (Int, Int) -> HSet.HashSet (Int, Int)
generateNeighbors availableSpaces amount position = HSet.filter (`HSet.member` availableSpaces) possibleNeighbors
  where possibleNeighbors = HSet.fromList $ map (addTuple position) directions ++
                                            map (addTuple position . multTuple amount) directions
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, 1), (1, -1), (-1, -1), (1, 1)]

data Element = Empty | Black | White deriving (Enum, Show, Eq)

toElement :: Int -> Element
toElement i = toEnum i :: Element

type StonePosition = (Int, Int)
type StoneSet = HSet.HashSet StonePosition
```

```haskell
type Matrix = M.Matrix Int

data Move = Move
    { moveColor :: Element
    , movePosition :: StonePosition
    }

data Board = Board
    { matrix :: Matrix
    , blackStones :: StoneSet
    , whiteStones :: StoneSet
    , stones :: StoneSet
    , mostRecentMove :: Move
    }

instance NFData Board where
  rnf b = b ‘seq‘ ()

showBoard :: Board -> Matrix
showBoard = matrix

isWithinBounds :: (Int, Int) -> Bool
isWithinBounds (a, b) = a >= 0 && a <= 8 && b >= 0 && b <= 8

isAvailable :: Board -> StonePosition -> Bool
isAvailable board position = (not $ HSet.member position $ stones board) && isWithinBounds position

move :: Board -> Element -> (Int, Int) -> Board
move board color pos@(x, y) = Board m’ b’ w’ s’ (Move color pos)
  where i = fromEnum color
        m = matrix board
        b = blackStones board
        w = whiteStones board
        s = stones board
        m’ = M.setElem i (x+1, y+1) m
        s’ = HSet.insert pos s
        b’ = if color == Black then HSet.insert pos b else b
        w’ = if color == White then HSet.insert pos w else w

initializeBoard :: (Int, Int) -> Board
initializeBoard = move (Board m b w s startMove) Black
  where b = HSet.fromList []
        w = HSet.fromList []
        s = HSet.fromList []
        m = M.fromList 15 15 (repeat 0)
        startMove = Move Empty (-1, -1)

getStoneChildren :: Board -> StonePosition -> HSet.HashSet (Int, Int)
getStoneChildren board position = HSet.filter (isAvailable board) $ generateNeighbors allSpaces 1 position
  where allSpaces = HSet.fromList allPositions
        allPositions = [(i, j) | i <- [0..8], j <- [0..8]]

childUnion :: [HSet.HashSet (Int, Int)] -> HSet.HashSet (Int, Int)
childUnion [] = HSet.fromList []
```

```haskell
childUnion (x:xs) = foldr HSet.union x xs

getChildren :: Board -> Element -> [Board]
getChildren board color = map (move board color) newPositions
  where setList b = map (getStoneChildren b) $ HSet.toList $ stones b
        newPositions = HSet.toList $ childUnion $ setList board

get :: Matrix -> (Int, Int) -> Maybe Int
get m (x, y) = M.safeGet (x+1) (y+1) m

oppositeColor :: Element -> Element
oppositeColor color = if color == Black then White else Black

goInDirHelper :: Matrix -> [Int] -> (Int, Int) -> (Int, Int) -> Element -> [Int]
goInDirHelper m l pos dir color
    | stop = r : l
    | stopBorder = l
    | otherwise = goInDirHelper m (r : l) (addTuple pos dir) dir color
  where stop = r == fromEnum (oppositeColor color) || r == 0
        stopBorder = r == -1
        r = fromMaybe (-1) $ get m pos

goInDir :: M.Matrix Int -> (Int, Int) -> (Int, Int) -> Element -> [Element]
goInDir m pos dir color = map toElement $ init (goInDirHelper m [] pos (multTuple (-1) dir) color) ++ reverse (goInDirHelper m [] pos dir color)

scoreLine2 :: Element -> [Element] -> Int
scoreLine2 color line
    | length line == 3 = helper3 line
    | length line == 4 = helper4 line
    | otherwise = 0
  where helper3 l
            | l == [Empty, color, color] || l == [color, color, Empty] = 50
            | otherwise = 0

        helper4 l
            | l == [Empty, color, color, Empty] = 100
            | l == [Empty, color, color, oppositeColor color] ||
              l == [oppositeColor color, color, Empty] = 50
            | otherwise = 0

scoreLine3 :: Element -> [Element] -> Int
scoreLine3 color line
    | length line == 4 = helper4 line
    | length line == 5 = helper5 line
    | otherwise = 0
  where helper4 l
            | l == [Empty, color, color, color] || l == [color, color, color, Empty] = 250
            | otherwise = 0

```

```
        helper5 l
            | l == [Empty, color, color, color, Empty] = 500
            | l == [Empty, color, color, color, oppositeColor color] ||
              l == [oppositeColor color, color, color, color, Empty] = 250
            | otherwise = 0

scoreLine4 :: Element -> [Element] -> Int
scoreLine4 color line
    | length line == 5 = helper5 line
    | length line == 6 = helper6 line
    | otherwise = 0
  where helper5 l
            | l == [Empty, color, color, color, color] || l == [color, color, color, color, Empty] = 500000
            | otherwise = 0

        helper6 l
            | l == [Empty, color, color, color, color, Empty] = 1000000
            | l == [Empty, color, color, color, color, oppositeColor color] ||
              l == [oppositeColor color, color, color, color, color, Empty] = 500000
            | otherwise = 0

scoreLine5 :: Element -> [Element] -> Int
scoreLine5 color line
    | length line >= 5 && length line <= 7 = helper line
    | otherwise = 0
  where helper [] = 0
        helper [_] = 0
        helper [_, _] = 0
        helper [_, _, _] = 0
        helper [_, _, _, _] = 0
        helper l@(a:b:c:d:e:_)
            | [a, b, c, d, e] == [color, color, color, color, color] = 10000000000
            | otherwise = scoreLine5 color (tail l)

halfDirections :: [(Int, Int)]
halfDirections = [(1, 0), (0, 1), (1, 1), (1, -1)]

reduce :: [Int] -> [Int] -> [Int] -> [Int] -> Int
reduce two three four five = ((sum two) `div` 2) + ((sum three) `div` 3) + ((sum four) `div` 4) + ((sum five) `div` 5)

heuristic :: Board -> Bool -> Int
heuristic board isSerial = 2*blackCount - whiteCount
  where m = matrix board
        colorLine (pos, dir) = goInDir m pos dir

        blackLines = [colorLine (pos, dir) Black | pos <- HSet.toList $ blackStones board, dir <- halfDirections]
        black2Serial = map (scoreLine2 Black) blackLines
        black3Serial = map (scoreLine3 Black) blackLines
```

```haskell
        black4Serial = map (scoreLine4 Black) blackLines
        black5Serial = map (scoreLine5 Black) blackLines

        black2Par = parMap (rpar . force) (scoreLine2 Black) blackLines
        black3Par = parMap (rpar . force) (scoreLine3 Black) blackLines
        black4Par = parMap (rpar . force) (scoreLine4 Black) blackLines
        black5Par = parMap (rpar . force) (scoreLine5 Black) blackLines

        blackCount = if isSerial
                        then reduce black2Serial black3Serial black4Serial black5Serial
                        else reduce black2Par black3Par black4Par black5Par

        whiteLines = [colorLine (pos, dir) White | pos <- HSet.toList $ whiteStones board, dir <- halfDirections]

        white2Serial = map (scoreLine2 White) whiteLines
        white3Serial = map (scoreLine3 White) whiteLines
        white4Serial = map (scoreLine4 White) whiteLines
        white5Serial = map (scoreLine5 White) whiteLines

        white2Par = parMap (rpar . force) (scoreLine2 White) whiteLines
        white3Par = parMap (rpar . force) (scoreLine3 White) whiteLines
        white4Par = parMap (rpar . force) (scoreLine4 White) whiteLines
        white5Par = parMap (rpar . force) (scoreLine5 White) whiteLines

        whiteCount = if isSerial
                        then reduce white2Serial white3Serial white4Serial white5Serial
                        else reduce white2Par white3Par white4Par white5Par

isTerminal :: Board -> Bool
isTerminal board = elem 10000000000 $ map (scoreLine5 color) colorLines
  where m = matrix board
        r = mostRecentMove board
        (p, color) = (movePosition r, moveColor r)
        colorLine (pos, dir) = goInDir m pos dir
        colorLines = [colorLine (p, dir) color | dir <- halfDirections]

infinity :: Int
infinity = maxBound :: Int

-- Inspired by the "star lines" of http://www.cs.columbia.edu/~sedwards/classes/2021/4995-fall/reports/Gomokururu.pdf
recentMoveHeuristic :: Board -> Int
```

```haskell
recentMoveHeuristic board = colorCount
  where m = matrix board
        r = mostRecentMove board
        (p, color) = (movePosition r, moveColor r)
        colorLine (pos, dir) = goInDir m pos dir
        colorLines = [colorLine (p, dir) color | dir <- halfDirections]
        color2 = map (scoreLine2 color) colorLines
        color3 = map (scoreLine3 color) colorLines
        color4 = map (scoreLine4 color) colorLines
        color5 = map (scoreLine5 color) colorLines
        colorCount = reduce color2 color3 color4 color5

orderMoves :: Bool -> [Board] -> [Board]
orderMoves isSerial moves = result
  where hmoves = zip heuristics moves
        sortedMoves = sortBy compareHeuristic hmoves
        compareHeuristic (ha, _) (hb, _)
            | ha > hb = LT
            | otherwise = GT
        extractMoves (_, m) = m
        heuristics = if isSerial
                        then map recentMoveHeuristic moves
                        else parMap (rpar . force) recentMoveHeuristic moves
        result = if isSerial
                    then map extractMoves sortedMoves
                    else parMap (rpar . force) extractMoves sortedMoves

minimax :: Board -> Int -> Int -> Int -> Element -> Bool -> (Int, Board)
minimax board depth alpha beta color isSerial
    | depth == 0 || isTerminal board = (h, board)
    | color == Black = playBlack (-infinity) board alpha beta children
    | otherwise = playWhite infinity board alpha beta children
  where children = orderMoves isSerial $ getChildren board color
        h = heuristic board isSerial

        playBlack maxValue maxChild _ _ [] = (maxValue, maxChild)
        playBlack maxValue maxChild a b (c:cs) =
            let (pvalue, _) = minimax c (depth-1) a b White isSerial
                comparison = pvalue > maxValue
                (maxValue', maxChild') = if comparison then (pvalue, c) else (maxValue, maxChild)
                a' = max a maxValue'
            in if maxValue >= b
                then (maxValue', maxChild') -- break loop
                else playBlack maxValue' maxChild' a' b cs -- continue loop

        playWhite minValue minChild _ _ [] = (minValue, minChild)
        playWhite minValue minChild a b (c:cs) =
            let (pvalue, _) = minimax c (depth-1) a b Black isSerial
```

```haskell
                comparison = pvalue < minValue
                (minValue', minChild') = if comparison then (pvalue, c) else (minValue, minChild)
                b' = min b minValue'
            in if minValue <= a
                then (minValue', minChild') -- break loop
                else playWhite minValue' minChild' a b' cs -- continue loop

chooseMove :: Element -> [(Int, Board)] -> (Int, Board)
chooseMove color moves = if color == Black then last sortedMoves else head sortedMoves
  where sortedMoves = sortBy compareHeuristic moves
        compareHeuristic (ha, _) (hb, _)
            | ha > hb = GT
            | otherwise = LT

parmapMinimax :: Int -> Board -> Element -> [(Int, Board)]
parmapMinimax depth board color
    | depth == 0 = parMap (rpar . force) play children
    -- playP was used during debugging, but I found that partial parallelization beyond one level didn't help
    | otherwise = parMap (rpar . force) playP children
  where children = getChildren board color
        play child = (fst $ minimax child 4 (-infinity) infinity (oppositeColor color) True, child)
        -- playP was used during debugging, but I found that partial parallelization beyond one level didn't help
        playP child = (fst $ chooseMove color $ parmapMinimax (depth-1) child $ oppositeColor color, child)

mapMinimax :: Board -> Element -> [(Int, Board)]
mapMinimax board color = map play children
  where children = getChildren board color
        play child = (fst $ minimax child 4 (-infinity) infinity (oppositeColor color) True, child)

loopNoMap :: Board -> Element -> Int -> [Board] -> [Board]
loopNoMap board color n boards
    | n == 0 = reverse boards
    | otherwise = loopNoMap next (oppositeColor color) (n-1) (next : boards)
  where next = snd $ minimax board 5 (-infinity) infinity color True

loopSerial :: Board -> Element -> Int -> [Board] -> [Board]
loopSerial board color n boards
    | n == 0 = reverse boards
    | otherwise = loopSerial next (oppositeColor color) (n-1) (next : boards)
  where next = snd $ chooseMove color $ mapMinimax board color

loopPar :: Board -> Element -> Int -> [Board] -> [Board]
loopPar board color n boards
    | n == 0 = reverse boards
    | otherwise = loopPar next (oppositeColor color) (n-1) (next : boards)
```

```haskell
  where next = snd $ chooseMove color $ parmapMinimax 0 board color

gomokuMain :: IO ()
gomokuMain = do
    putStrLn "BEGIN GAME"

    let startStone = (7, 7)
    let board = initializeBoard startStone

    args <- getArgs
    if length args /= 1
        then do die $ "Usage: stack exec gomokuku-exe <argument>\n<argument> may be serial, parallel, or no-map"
    else if head args == "serial"
        then do
            putStrLn "SERIAL"
            let solutions = loopSerial board White 10 []
            mapM_ putStrLn $ map (show . (‘heuristic‘ True)) solutions
            mapM_ print $ map showBoard solutions
    else if head args == "parallel"
        then do
            putStrLn "PARALLEL"
            let solutions = loopPar board White 10 []
            mapM_ putStrLn $ map (show . (‘heuristic‘ True)) solutions
            mapM_ print $ map showBoard solutions
    else do
        putStrLn "NO MAP"
        let solutions = loopNoMap board White 10 []
        mapM_ putStrLn $ map (show . (‘heuristic‘ True)) solutions
        mapM_ print $ map showBoard solutions
```

```haskell
import Lib

initialBoard :: Board
initialBoard = initializeBoard (7, 7)

evaluateTest :: String -> Bool -> IO ()
evaluateTest testName test = if test then putStrLn $ "Test {" ++ testName ++ "} passed." else putStrLn $ "Test {" ++ testName ++ "} failed."

testGetChildren :: Bool
testGetChildren = (length $ getChildren initialBoard White) == 8

testOverline :: Bool
testOverline = ((‘heuristic‘ True) $ (move (move (move (move (move initialBoard Black (7, 8)) Black (7, 9)) Black (7, 10)) Black (7, 11)) Black (7, 12))) == 0

testScore2 :: Bool
testScore2 = ((‘heuristic‘ True) $ (move initialBoard Black (7, 8))) == 200

testScore3 :: Bool
testScore3 = ((‘heuristic‘ True) $ (move (move initialBoard Black (7, 8)) Black (7, 9))) == 1000
```

```

testScore4 :: Bool
testScore4 = ((`heuristic` True) $ (move (move (move initialBoard
    Black (7, 8)) Black (7, 9)) Black (7, 10))) == 2000000

testScore5 :: Bool
testScore5 = ((`heuristic` True) $ (move (move (move (move
    initialBoard Black (7, 8)) Black (7, 9)) Black (7, 10)) Black
    (7, 11))) == 20000000000

testIsTerminal :: Bool
testIsTerminal = isTerminal $ (move (move (move (move initialBoard
    Black (7, 8)) Black (7, 9)) Black (7, 10)) Black (7, 11))

testParSerialMatch :: Bool
testParSerialMatch = (map showBoard $ serialSolutions) == (map
    showBoard $ parallelSolutions)
  where serialSolutions = loopSerial initialBoard White 10 []
        parallelSolutions = loopPar initialBoard White 10 []

main :: IO ()
main = do
    putStrLn "BEGIN TESTING"
    evaluateTest "Get Children of Board" testGetChildren -- initial
     board should have eight children
    evaluateTest "Overline" testOverline -- lines with length
    greater than 5 actually have a heuristic of 0
    evaluateTest "Score 2" testScore2
    evaluateTest "Score 3" testScore3
    evaluateTest "Score 4" testScore4
    evaluateTest "Score 5" testScore5
    evaluateTest "Termination" testIsTerminal
    evaluateTest "Parallel = Serial Output" testParSerialMatch
```

8 References

[1] http://gomokuworld.com/gomoku/2

[2] https://www.theverge.com/2019/11/27/20985260/ai-go-alphago-lee-se-dol-retired-deepmin

[3] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#Pseudocode

[4] http://www.cs.columbia.edu/~sedwards/classes/2021/4995-fall/reports/Gomokururu.pdf

[5] https://wiki.haskell.org/ThreadScope

[6] https://huggingface.co/spaces/stabilityai/stable-diffusion

9 Project Mascot

Generated courtesy of Stability AI's Stable Diffusion 2 [6]:

Figure 4: The meaning of ”Gomokuku for Koko Puffs” translated into pixels. Determining the true prompt that generated this image is left as an exercise for the reader.