I. The Game

Mancala is a two-player game with the goal to capture as many marbles as possible. A wooden board contains two rows of six holes each and two pockets on either side called mandalas that are used to store the marbles for each player as shown in Figure 1. The game starts with 6 marbles in each of the six holes. Players take turns choosing a hole on their side and distributing the marbles to the holes in a counter-clockwise direction, making sure to drop a marble in their designated mancala as they pass it. If the last marble lands on the opposing player's side, their turn ends. If it lands in an empty pocket on their own side and there is at least one marble in the hole directly across from it, the player gets to capture both holes’ marbles. If the last marble lands in their store, they get to choose another hole. The game ends when either player has an empty row. Any marbles that are not captured at this time, go to the player whose side they were left on.

Figure 1. Diagram of Mancala game board
II. Minimax

Minimax is a search algorithm usually used in game-solving to find the best next move. The algorithm works by using a minimizer and a maximizer where the player tries to minimize and maximize their score respectively [1]. This score is calculated by evaluating the current game state and determining which new game state will minimize/maximize the score based on possible moves.

A tree with nodes of next possible games states and evaluates those nodes to see which move has the highest evaluation score (Figure 2). Because there are so many possible moves, a depth limit is passed in to indicate how far down the tree we want to search [1].

![Minimax Tree Image](image.png)

Figure 2. Steps in example minimax tree
III. Alphabeta Pruning

The alphabeta technique is used to optimize the Minimax algorithm. Using this technique, minimax is able to search the nodes of the tree faster. The algorithm will know not to search certain branches of the tree because it will not hold the minimal or maximum value [2]. It works by passing in alpha and beta values to the Minimax algorithm. Alpha is the maximized value and beta is the minimized value. The maximizer updates the alpha value to the maximum value found so far while the minimizer updates the beta value to the minimum value found [3].

IV. Implementation and Parallelism

Below is a snippet of the sequential minimax implementation. The function takes in the current game state, a boolean indicating whether or not we want to minimize or maximize the score, a starting depth, and the depth limit of the tree. It then returns a tuple of the score which is a result of the game board evaluation and the best move to get that score (which is represented by a number on the board).

```haskell
minimax :: (GameState a) => a -> Bool -> Int -> Int -> (Int, Maybe Int)
minimax gs _ depth depthlimit | depth == depthlimit || gameOver gs = (evaluate gs, Nothing)
minimax gs minimize depth depthlimit =
    let minOrMax = (if minimize then minimumBy else maximumBy) (comparing fst)
        possibilities = (possibleMoves gs)
        scores = map fst $ map (\poss -> (minimax (makePossibility gs poss) (not minimize) (depth+1) depthlimit)) possibilities
        wrappedPossibilities = map Just possibilities
        scorePossPairs = zip scores wrappedPossibilities
    in minOrMax scorePossPairs
```
The parallelized minimax algorithm is similar to the sequential, except that it implements parList from the Control.Parallel.Strategies library to evaluate each list element in parallel based on a given strategy. A strategy takes a data structure as input and parallelizes it using rpar and rseq to then return the original value [4]. In this case, the strategy is rseq which evaluates an argument to its Weak Head Normal Form (WHNF) [4]. WHNF is defined as when the outermost part has been evaluated to the lambda abstraction [5].

\[
\text{minimaxPar} \quad :: \quad (\text{GameState}\ a) \rightarrow a \rightarrow \text{Bool} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int}, \text{Maybe} \ \text{Int})
\]

\[
\text{minimaxPar} \ gs \ _ \ \text{depth} \ \text{depthLimit} \ | \ \text{depth} == \ \text{depthLimit} | | \ \text{gameOver} \ gs =
\]

\[
(\text{evaluate} \ gs, \ \text{Nothing})
\]

\[
\text{minimaxPar} \ gs \ \text{minimize} \ \text{depth} \ \text{depthLimit} =
\]

\[
\quad \quad \text{let minOrMax} = (\text{if minimize then minimumBy else maximumBy}) \ (\text{comparing} \ \text{fst})
\]

\[
\quad \quad \text{possibilities} = (\text{possibleMoves} \ gs)
\]

\[
\quad \quad \text{scores} = (\text{map} \ \text{fst} \ \text{map} \ (\\text{poss} \rightarrow (\text{minimaxPar} \ \text{makePossibility} \ gs \ \text{poss}) \ (\text{not minimize}) \ (\text{depth+1}) \ \text{depthLimit})) \ \text{possibilities} \ \text{`using` parList rseq}
\]

\[
\quad \quad \text{wrappedPossibilities} = \text{map} \ \text{Just} \ \text{possibilities}
\]

\[
\quad \quad \text{scorePossPairs} = \text{zip} \ \text{scores} \ \text{wrappedPossibilities} \ \text{in}
\]

\[
\quad \quad \text{minOrMax} \ \text{scorePossPairs}
\]

V. Results and Conclusion

In order to test the results, two boards were tested with a depth limit that was kept constant across all functions in order to accurately compare the data. It is expected that the results will vary when tested with “harder” or more complicated boards. Tests were conducted with the starting mancala board defined as

\[
\text{let board} = \text{Board} \ \text{V.fromList} \ [6,6,6,6,6,6,0,6,6,6,6,6,6,0]
\]

and a fixed depth limit of 8.
The average running times of the sequential minimax and alphabeta pruning implementations are shown in Figure 3. The alpha beta pruning algorithm, even without parallelism, was 3.485 seconds faster than the standard minimax. This is equivalent to a 82.02% decrease in time.

<table>
<thead>
<tr>
<th>Core</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Running Time (s)</td>
<td>4.456</td>
<td>2.438</td>
<td>1.344</td>
<td>1.916</td>
<td>2.173</td>
<td>2.073</td>
</tr>
</tbody>
</table>

Figure 4 displays the results of the parallel minimax algorithm using different amounts of cores. The largest difference in time is seen with 4 cores at 1.344 seconds which is 68.37% faster than the sequential minimax. There is an increase in the total time at 6, 8, and 10 cores which contradicts the idea that parallelism is used to increase performance. However, with the increase in new threads also comes an increase in the time needed to create all those threads. This starts to outweigh the benefits of parallelism especially since the algorithm was already running pretty quickly to begin with. This can also be the result of hardware limitations when it comes to how many threads the computer can make. With these results it can be inferred that alphabeta is
the most optimized when it comes to the starting game board and an alphabeta parallel could possibly be even faster.

**Parallel Minimax Threadscope and Runtime Data**

![Figure 5. Performance data of parallel minimax on 1 core](image)

**Figure 5. Performance data of parallel minimax on 1 core**

![Figure 6. Parallel minimax on 1 core](image)

**Figure 6. Parallel minimax on 1 core**
Figure 7. Performance data of parallel minimax on 2 cores

Figure 8. Parallel minimax on 2 cores
Figure 9. Performance data of parallel minimax on 4 cores

Figure 10. Parallel minimax on 4 cores
Figure 11. Performance data of parallel minimax on 6 cores

Figure 12. Parallel minimax on 6 cores
Figure 13. Performance data of parallel minimax on 8 cores

Figure 14. Parallel minimax on 8 cores
Figure 15. Performance data of parallel minimax on 10 cores

Figure 16. Parallel minimax on 10 cores
From these Threadscope graphs, we can see that the effects of parallelism on the load balancing between cores tapers off after 4 cores are used.

Board 2

\[
\text{let board} = \text{Board}\ V.\text{fromList [1,2,7,4,0,1,32,1,0,2,1,1,2,18]}
\]

<table>
<thead>
<tr>
<th>Minimax</th>
<th>3.707</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaBeta Pruning</td>
<td>1.823</td>
</tr>
</tbody>
</table>

*Figure 17. Average running times of minimax and alphabeta pruning algorithms on Board 2*

<table>
<thead>
<tr>
<th>Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Running Time (s)</td>
<td>3.840</td>
<td>1.822</td>
<td>1.217</td>
<td>1.660</td>
<td>1.620</td>
<td>1.797</td>
</tr>
</tbody>
</table>

*Figure 18. Parallel Minimax running on increasing cores on Board 2*

In Figure 17 we can see that the alphabeta function again ran faster than the minimax algorithm, but with a significantly less speedup of 50.82%. Figure 18 also displays the results of the parallel minimax algorithm on Board 2 with the increasing number of cores. Again we see the least amount of time taken with 4 cores at 1.217 seconds which is 67.17% faster than the sequential minimax. In this case, the speedup was more consistent in comparison to the results with the starting board. This can lead us to think that the alphabeta function’s optimization is dependent on the board and the
efficiency of the original sequential minimax algorithm which already ran quickly to begin with.

VI. Code

Play.hs

module Main where

import Data.List
import qualified Data.Vector as V
import Mancala
import Minimax
import System.Exit
import System.IO
import Text.Printf

{-
Starting game board

    A  B  C  D  E  F
P  6   6  6  6  6  6
0     0
6   6  6  6  6  6  C
L K J I H G
-}

-- translate letters into numeric spaces
--valid computer moves
getComputerMove :: String -> Int
getComputerMove "L" = 0
getComputerMove "K" = 1
getComputerMove "J" = 2
getComputerMove "I" = 3
getComputerMove "H" = 4
getComputerMove "G" = 5
getComputerMove "q" = 13 --quit
getComputerMove "Q" = 13 --quit
getComputerMove _ = 14 --invalid input

--valid player moves
getPlayerMove :: String -> Int
getPlayerMove "A" = 12
getPlayerMove "B" = 11
getPlayerMove "C" = 10
getPlayerMove "D" = 9
getPlayerMove "E" = 8
getPlayerMove "F" = 7
getPlayerMove "q" = 13 --quit
getPlayerMove "Q" = 13 --quit
getPlayerMove _ = 14 --invalid input

-- translate numeric spaces into letters
getComputerLetter :: Int -> String
getComputerLetter 0 = "L"
getComputerLetter 1 = "K"
getComputerLetter 2 = "J"
getComputerLetter 3 = "I"
getComputerLetter 4 = "H"
getComputerLetter 5 = "G"
getComputerLetter _ = error "Invalid move"

-- input move, check if move is valid
getMove :: Player -> Board -> IO Int
getMove p (Board b) = do
    str <- getLine
    let move = if p == Computer then getComputerMove str else getPlayerMove str
    if (move == 14 || ((b V.! move) == 0 && move /= 13)) --invalid letter or hole that is empty
       then do
            putStrLn "Invalid Move. Try again:"
            hFlush stdout
            getMove p (Board b)
       else do
            if move == 13 --quit key
               then do
                    putStrLn "Quitting."
                    exitWith ExitSuccess
            else return move

-- identify who's turn it is
printPlayer :: Player -> IO ()
printPlayer Computer = putStrLn "Computer: "

printPlayer Player2 = putStrLn "You: 

--print marbles in each hole
printMarbles :: Board -> [Int] -> IO String
printMarbles (Board b) xs = do
  lineStr <-
    return
      (foldl (\str n -> str ++ (printf "%3d" n)) "" (map (\i -> b V.! i) xs))
  return lineStr

--print hole letters on top along with marbles
printTopRow :: Board -> IO ()
printTopRow b = do
  str <- printMarbles b [12, 11 .. 7]
  putStrLn $ "        
  putStrLn $ "        

--print hole letters on bottom along with marbles
printBottomRow :: Board -> IO ()
printBottomRow b = do
  str <- printMarbles b [0 .. 5]
  putStrLn $ "        
  putStrLn $ "        

--print both stores
printStores :: Board -> IO ()
printStores (Board b) =
  putStrLn $ "        " ++ (show $ b V.! 13) ++ (replicate 20 ' ') ++ (show $ b V.! 6)

--print board
printBoard :: Board -> IO ()
printBoard b = do
  printTopRow b
  printStores b
  printBottomRow b

--print board and get game input
printGameState :: MancalaGameState -> IO ()
printGameState (MancalaGameState b p _) = do
  printPlayer p
printBoard b

applyMove :: MancalaGameState -> Int -> IO ()
applyMove gs move = return (distributeMarbles gs move) >>= playGame

humanMoveGS :: MancalaGameState -> IO Int
humanMoveGS (MancalaGameState board player _) = do
  m <- getMove player board
  putStrLn ""
  return m

makeMoveGS :: MancalaGameState -> IO Int
makeMoveGS gs = do
  let (score, move) = minimaxPar gs False 0 8
  let Just x = move
  if move == Just x then do
    printf "Computer move: \%s.\n\n" (getComputerLetter x)
    return x
  else error "Invalid move: Nothing"

playGame :: MancalaGameState -> IO()
-- if game is over print results
playGame (MancalaGameState board computer player) | rowEmpty board Computer ||
rowEmpty board Player2 = do
  putStrLn $ "Game over. " ++ winString
  printGameState (MancalaGameState board computer player)
  where
    other | player == Computer = Player2
    | otherwise = Computer
    winString | (evaluate (MancalaGameState board computer player) > 0) = "Winner is " ++ (show player)
    | (evaluate (MancalaGameState board computer player) < 0) = "Winner is " ++ (show other)
    | otherwise = "Tie."
  -- else print current game state and get next move
playGame (MancalaGameState board Computer Computer) = do
  printGameState (MancalaGameState board Computer Computer)
  putStrLn "Computer's turn"
  move <- makeMoveGS (MancalaGameState board Computer Computer)
  applyMove (MancalaGameState board Computer Computer) move
playGame (MancalaGameState board Player2 x) = do
  printGameState (MancalaGameState board Player2 x)
  putStrLn "Enter move: ">
  hFlush stdout
  move <- humanMoveGS (MancalaGameState board Player2 x)
  applyMove (MancalaGameState board Player2 x) move

startGameState = MancalaGameState initialBoard Computer Computer
main = do
  startGS <- return startGameState
  playGame startGS

Mancala.hs

module Mancala where

import Data.List
import qualified Data.Vector as V
--import Minimax

class GameState a where
  evaluate :: a -> Int
  gameOver :: a -> Bool
  possibleMoves :: a -> [Int]
  makePossibility :: a -> Int -> a
  isMaximizing :: a -> Bool

data Player = Computer | Player2
  deriving (Eq, Show)

data Board = Board (V.Vector Int)
  deriving (Show)

initialBoard = Board $ V.fromList [6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
makePossibility = distributeMarbles

isMaximizing (MancalaGameState _ computer player) = computer == player

rowEmpty :: Board -> Player -> Bool
rowEmpty (Board board) player =
  rowTotal == 0
  where rowTotal | player == Computer = board V.! 0 + board V.! 1 + board V.! 2 +
    board V.! 3 + board V.! 4 + board V.! 5
    | otherwise = board V.! 7 + board V.! 8 + board V.! 9 + board V.! 10 + board V.! 11 + board V.! 12

mancalaTotal :: Board -> Player -> Int
mancalaTotal (Board board) player = board V.! (storePos player)

/-
Game is over when both rows are empty
Total 72 marbles are caught by players
-}

isGameOver :: Board -> Bool
isGameOver board
  | totalPoints == 72 = True
  | otherwise = False
  where totalPoints = mancalaTotal board Computer + mancalaTotal board Player2

storePos :: Player -> Int
storePos p | p == Computer = 6
  | otherwise = 13

getScore :: Board -> Player -> Int
getScore board player = (if (rowEmpty board Computer || rowEmpty board Player2) then 100 else 1) * (mancalaTotal board player - mancalaTotal board otherPlayer)
  where otherPlayer
    | player == Computer = Player2
    | otherwise = Computer

getPossibleMoves :: Board -> Player -> [Int]
getPossibleMoves (Board board) player =
  filter (\i -> (board V.! i) /= 0) rows
  where rows | player == Computer = [0..5]
    | otherwise = [7..12]

distributeMarbles :: MancalaGameState -> Int -> MancalaGameState
distributeMarbles (MancalaGameState (Board b) computer player) pos = (MancalaGameState
finalNewBoard nextPlayer player)

where
count = (b V.! pos)
boardGetMarbles = Board (b V.// [(pos, 0)])
(newBoard, nextPlayer) = placeStones boardGetMarbles computer (pos + 1) count
finalNewBoard | (rowEmpty newBoard Computer || rowEmpty newBoard Player2) =
endGameMove newBoard
| otherwise = newBoard

endGameMove :: Board -> Board
endGameMove (Board b) =
  Board
  $ b
  V.// ( [(6, (b V.! 6) + computerTotal), (13, (b V.! 13) + playerTotal)]
    ++ computerZeros
    ++ playerZeros
  ) where
totalFunc = \l -> sum $ map (\i -> b V.! i) l
computerTotal = totalFunc [0 .. 5]
playerTotal = totalFunc [7 .. 12]
zeroFunc = map (\i -> (i, 0))
computerZeros = zeroFunc [0 .. 5]
playerZeros = zeroFunc [7 .. 12]

nextPos :: Player -> Int -> Int
nextPos player pos | (player == Computer && pos == 12) = 0
| (player == Player2 && pos == 5) = 7
| pos == 13 = 0
| otherwise = pos + 1

placeLastStone :: Board -> Player -> Int -> (Board, Player)
placeLastStone (Board board) player pos
  | board V.! pos == 0 && board V.((-) 12 pos) /= 0 && playerHole == player
  = (Board $ board V.// [(holeAcross, 0), (storePos player, newCount)], otherPlayer)
where
  holeAcross = (-) 12 pos
  holeAcrossCount = (board V.! holeAcross)
  newCount = (mancalaTotal (Board board) player) + holeAcrossCount + 1
  otherPlayer = player == Computer = Player2
  | otherwise = Computer
  playerHole = pos >= 0 && pos <= 6 = Computer
placeLastStone (Board board) player pos =
  (newBoard, nextPlayer)
  where newBoard = Board $ board V.// [(pos, (board V.! pos) + 1)]
  otherPlayer | player == Computer = Player2
               | otherwise = Computer
  nextPlayer   | pos == storePos player = player
               | otherwise = otherPlayer

-- get only changes updates to update score
takeReverse :: [Int] -> Int -> [Int]
takeReverse listUpdates count = takeReverse' listUpdates count []
  where takeReverse' (x _ ) 1 acc = x : acc
        takeReverse' (x : xs) count acc = takeReverse' xs (count - 1) (x : acc)

placeStones :: Board -> Player -> Int -> Int -> (Board, Player)
placeStones (Board b) player pos count = placeLastStone intermediateBrd player newPos
  where allUpdates = iterate (\i -> nextPos player i) pos
       currUpdates = takeReverse allUpdates count
       intermediateBrd = Board $ b V.// (map (\l -> (head l, (b V.! head l) + length l)) $ group . sort . tail currUpdates)
       newPos = head currUpdates

Minimax.hs

module Minimax
where
import Debug.Trace
import Data.List
import Data.Ord
import Mancala
import qualified Data.Vector as V
import Control.Parallel.Strategies

{-
--Test:
main :: IO()
main = do
  let board = Board $ V.fromList [6,6,6,6,6,6,0,6,6,6,6,6,6,0]
  let gs = MancalaGameState board Computer Player2
  --print (minimax gs False 0 8)
--print (minimaxPar gs False 0 8)
p
print (alphabeta gs 0 8 (-1000) 1000)
--
-- ghc -threaded -rtsopts -eventlog --make -main-is Minimax Minimax.hs -package
vector
-- time ./Minimax +RTS -ls -s
-- time ./Minimax +RTS -N2 -ls -s

minimax :: (GameState a) => a -> Bool -> Int -> Int -> (Int, Maybe Int)
minimax gs depth depthlimit | depth == depthlimit || gameOver gs = (evaluate gs, Nothing)
minimax gs minimize depth depthlimit =
  let minOrMax = (if minimize then minimumBy else maximumBy) (comparing fst)
      possibilities = (possibleMoves gs)
      scores = map fst $ map (\poss -> (minimax (makePossibility gs poss) (not minimize) (depth+1) depthlimit)) possibilities
      wrappedPossibilities = map Just possibilities
      scorePossPairs = zip scores wrappedPossibilities in
  minOrMax scorePossPairs

minimaxPar :: (GameState a) => a -> Bool -> Int -> Int -> (Int, Maybe Int)
minimaxPar gs depth depthlimit | depth == depthlimit || gameOver gs = (evaluate gs, Nothing)
minimaxPar gs minimize depth depthlimit =
  let minOrMax = (if minimize then minimumBy else maximumBy) (comparing fst)
      possibilities = (possibleMoves gs)
      scores = (map fst $ map (\poss -> (minimaxPar (makePossibility gs poss) (not minimize) (depth+1) depthlimit)) possibilities) `using` parList rseq
      wrappedPossibilities = map Just possibilities
      scorePossPairs = zip scores wrappedPossibilities in
  minOrMax scorePossPairs

{-
alphabeta :: (GameState a) => a -> Int -> Int -> Int -> Int -> (Int, Maybe Int)
alphabeta gs _ _ _ _ | gameOver gs = (evaluate gs, Nothing)
alphabeta gs depth depthlimit _ _ | depth == depthlimit = (evaluate gs, Nothing)
alphabeta gs depth depthlimit alpha beta =
  alphabetaFold possibilities alpha beta (-1)
  where possibilities = possibleMoves gs
    alphabetaFold [] a _ bestChild = (a, Just bestChild)
    alphabetaFold (x:xs) a b bestChild =
let child = makePossibility gs x

    newAlpha = (if (isMaximizing child) then alphabetamax else alphabetamin) child (depth+1) depthlimit a b in

    if (newAlpha >= b)
        then (newAlpha, Just x)
        else alphabetafold xs (max a newAlpha) b (if newAlpha > a then x else bestChild)

alphabetamax :: (GameState a) => a -> Int -> Int -> Int -> Int -> Int
alphabetamax gs _ _ _ _ | gameOver gs = evaluate gs
alphabetamax gs depth depthlimit _ _ | depth == depthlimit = evaluate gs
alphabetamax gs depth depthlimit alpha beta =
    alphabetaHelper gs possibilities alpha beta depth depthlimit
    where possibilities = possibleMoves gs

alphabetamin :: (GameState a) => a -> Int -> Int -> Int -> Int -> Int
alphabetamin gs _ _ _ _ | gameOver gs = evaluate gs
alphabetamin gs depth depthlimit _ _ | depth == depthlimit = evaluate gs
alphabetamin gs depth depthlimit alpha beta =
    alphabetaHelper gs possibilities alpha beta depth depthlimit
    where possibilities = possibleMoves gs

alphabetaHelper :: (GameState a) => a -> [Int] -> Int -> Int -> Int -> Int -> Int
alphabetaHelper _ [] _ _ b _ = b
alphabetaHelper gs (x:xs) a b depth depthlimit =
    let child = makePossibility gs x
        newBeta = (if (isMaximizing child) then alphabetamax else alphabetamin) child (depth+1) depthlimit a b in
    if (newBeta <= a)
        then newBeta
        else alphabetaHelper gs xs a (min b newBeta) depth depthlimit
->}
References


