Michael Lee
UNI: mhl2156

Chess Engine Report

This project was made to parallelize a mini-max algorithm for chess, a turn-based game. Mini-max is a turn-based algorithm, where one player (maximizer) must find the best move given an opposing player's (minimizer) optimal counter-play.

Mini-max works by traversing all potential game-states reachable from some original state, either maximizing or minimizing the values of these states, and returning this value to the parent so that they can then make a decision on how they would like to proceed.

My mini-max algorithm takes 3-5 parameters:

- Board - a representation of the current position being evaluated
- Depth - Depth to which we would like to recurse
- Player - which player is trying to maximize or minimize
- Alpha, Beta - parameter only used in alpha-beta pruning

Given that there is a high level of recursion occurring to traverse these trees, parallelization seems a good way to optimize this problem.

Firstly, there needed to be a simulation of chess so that the compiler could logically play it, and so that there was some logical representation of a board passable to the minimax algorithm. The board is set up as a list of lists of pieces, where pieces are denominated by their color and point value. Pieces must be differentiated so that the mini-max algorithm is able to tell which player is playing with which pieces.

A chess move is represented as a data type “Move” that is a tuple of two tuples: (Starting Position, Ending Position).

Each piece in chess has a unique set of move rules, and there are moves that are legal as well as moves that are not legal. As such, each different piece needed to have a separate move generation function that parsed their move rules for all legal rules. Along with this, there are separate edge cases for movement, such as capturing rules. Pieces cannot move onto a square that is occupied by their own color, but they can move to capture an enemy piece. Along with this, once a piece comes into contact with another piece, it can no longer move further in the same direction.

This required a separate algorithm for finding legal moves.

Taking a rook, for example:

A rook can move, at maximum, seven squares either forwards, backwards, left, or right. However, once a rook comes into contact with a piece or the edge of the board, this metric becomes smaller.

The `takeWhileInclusive` method addresses this, where it traverses seven moves away from the rook’s original position, stopping if the rook attempts to move onto a square that contains another piece. Then, using the `checkLastElemCap’ function, it checks if this last piece is capturable, so as to determine if this move was also legal or not and includable in the final legalMoves list. This logic was repeated for the bishop, queen, knight, and king.

Pawns, on the other hand, were a bit more tricky. Because pawns have a different set of rules for capturing (diagonal) versus movement (forward), two separate lists needed to be parsed: a capture move list, and a movement move list, due to the fact that a pawn cannot capture a piece

directly in front of it. After checking them for separate legality rules, they are concatenated, generating all legal moves for the pawn.

After this, I needed some type of evaluation heuristic to determine the value of a board state, so that there was a basis of comparison between two different board states. The implementation of evalBoard was straightforward: I parsed the entire [[Piece]] representation of the board, and added up all point values determined by the getPoints function for both white and black. The value was then calculated by subtracting black's total piece valuation by white's, meaning that a positive number was advantageous for white while a negative number was advantageous for black.

The function makeMove takes in a board and a move, and outputs an updated board. Using the coordinates from the move data type, The piece is removed from the starting index, replaced with Empty, and then replaces the target coordinate with the original piece. Afterwards, the new board is returned.

The function minimax accepts a given board state, and immediately checks if the depth is equal to 0. If it is equal to 0, it can just return the given board state without looking for more potential moves.

There are 5 separate implementations for minimax in the provided code:

- miniMaxSeq
    - A non-parallel, recursive DFS solution to mini-max
- miniMaxParMap
    - A parallel solution that just calls `parMap` on all nodes of the mini-max tree
- miniMaxParGran

    - A parallel solution that calls miniMaxSeq on each node of the original tree in parallel
- miniMaxSeqPrune
    - A sequential algorithm that implements alpha-beta pruning
- miniMaxParPrune
    - A parallel algorithm that calls miniMaxSeqPrune on each node of the original tree in parallel

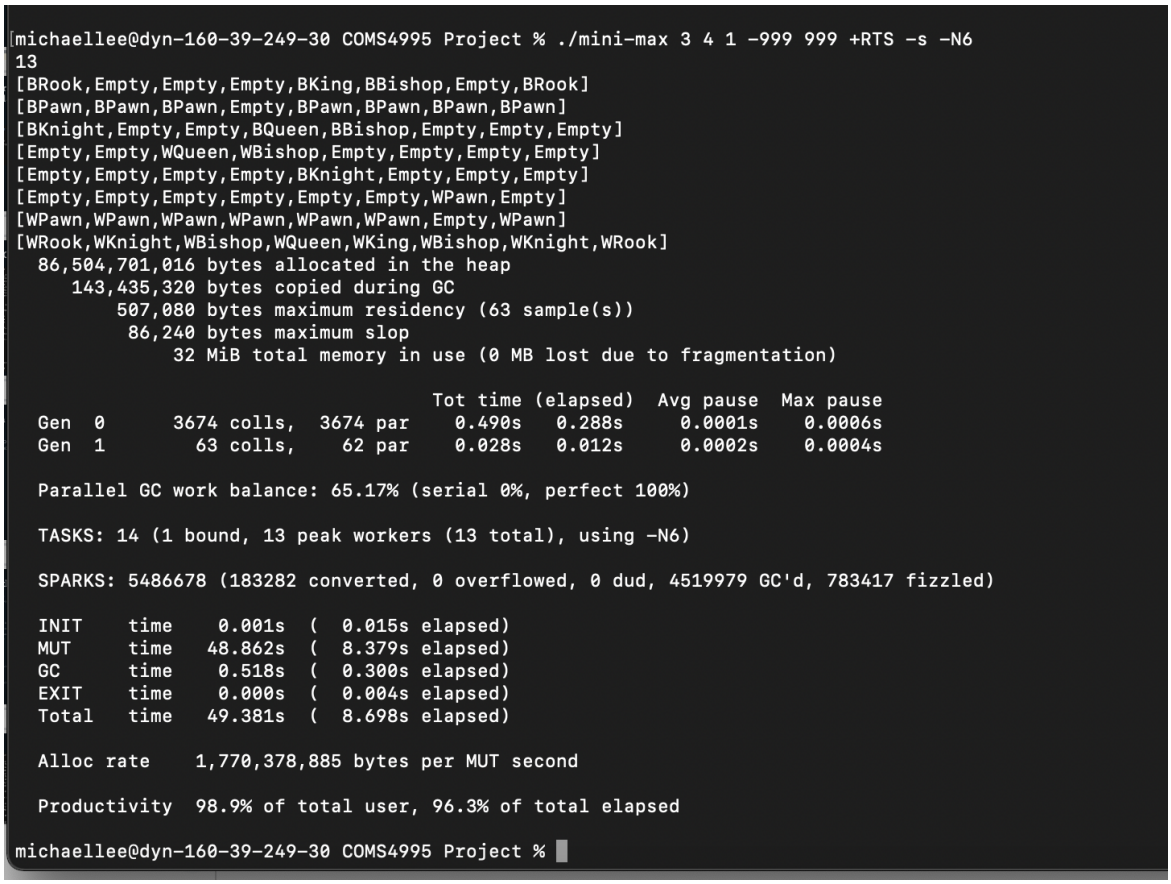
Each was designed as an attempt to increase the efficiency of the previous.

miniMaxSeq outPut at depth == 4:

```
michaellee@dyn-160-39-249-30 COMS4995 Project % ./mini-max 1 4 1 -999 999 +RTS -s -N1
13
[BRook,Empty,Empty,Empty,BKing,BBishop,Empty,BRook]
[BPawn,BPawn,BPawn,Empty,BPawn,BPawn,BPawn,BPawn]
[BKnight,Empty,Empty,BQueen,BBishop,Empty,Empty,Empty]
[Empty,Empty,WQueen,WBishop,Empty,Empty,Empty,Empty]
[Empty,Empty,Empty,Empty,BKnight,Empty,Empty,Empty]
[Empty,Empty,Empty,Empty,Empty,Empty,WPawn,Empty]
[WPawn,WPawn,WPawn,WPawn,WPawn,WPawn,Empty,WPawn]
[WRook,WKnight,WBishop,WQueen,WKing,WBishop,WKnight,WRook]
  79,504,113,048 bytes allocated in the heap
      65,579,200 bytes copied during GC
          79,872 bytes maximum residency (6 sample(s))
          29,952 bytes maximum slop
               7 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     19868 colls,     0 par    0.200s   0.237s     0.0000s    0.0004s
  Gen  1         6 colls,     0 par    0.002s   0.002s     0.0003s    0.0003s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.001s  (  0.006s elapsed)
  MUT     time   51.627s  ( 51.979s elapsed)
  GC      time    0.201s  (  0.239s elapsed)
  EXIT    time    0.000s  (  0.008s elapsed)
  Total   time   51.829s  ( 52.232s elapsed)

  Alloc rate    1,539,968,483 bytes per MUT second

  Productivity  99.6% of total user, 99.5% of total elapsed
```

Here, the runtime of the sequential miniMax algorithm is 51.829 seconds.

miniMaxParMap output at depth == 4 with 6 cores:



```
michaellee@dyn-160-39-249-30 COMS4995 Project % ./mini-max 3 4 1 -999 999 +RTS -s -N6
13
[BRook,Empty,Empty,Empty,BKing,BBishop,Empty,BRook]
[BPawn,BPawn,BPawn,Empty,BPawn,BPawn,BPawn,BPawn]
[BKnight,Empty,Empty,BQueen,BBishop,Empty,Empty,Empty]
[Empty,Empty,WQueen,WBishop,Empty,Empty,Empty,Empty]
[Empty,Empty,Empty,Empty,BKnight,Empty,Empty,Empty]
[Empty,Empty,Empty,Empty,Empty,Empty,WPawn,Empty]
[WPawn,WPawn,WPawn,WPawn,WPawn,WPawn,Empty,WPawn]
[WRook,WKnight,WBishop,WQueen,WKing,WBishop,WKnight,WRook]
  86,504,701,016 bytes allocated in the heap
     143,435,320 bytes copied during GC
         507,080 bytes maximum residency (63 sample(s))
          86,240 bytes maximum slop
              32 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      3674 colls,  3674 par    0.490s   0.288s     0.0001s    0.0006s
  Gen  1        63 colls,    62 par    0.028s   0.012s     0.0002s    0.0004s

  Parallel GC work balance: 65.17% (serial 0%, perfect 100%)

  TASKS: 14 (1 bound, 13 peak workers (13 total), using -N6)

  SPARKS: 5486678 (183282 converted, 0 overflowed, 0 dud, 4519979 GC'd, 783417 fizzled)

  INIT    time    0.001s  (  0.015s elapsed)
  MUT     time   48.862s  (  8.379s elapsed)
  GC      time    0.518s  (  0.300s elapsed)
  EXIT    time    0.000s  (  0.004s elapsed)
  Total   time   49.381s  (  8.698s elapsed)

  Alloc rate    1,770,378,885 bytes per MUT second

  Productivity  98.9% of total user, 96.3% of total elapsed

michaellee@dyn-160-39-249-30 COMS4995 Project %
```

A strong increase, with completion in 8.698 seconds. However, the spark count for the current board state being evaluated is 5,486,678. With better granularity, this could likely be optimized.

miniMaxParGran output at depth == 4 with 6 cores:

```
michaellee@dyn-160-39-249-30 COMS4995 Project % ./mini-max 4 4 1 -999 999 +RTS -s -N6
13
[BRook,Empty,Empty,Empty,BKing,BBishop,Empty,BRook]
[BPawn,BPawn,BPawn,Empty,BPawn,BPawn,BPawn,BPawn]
[BKnight,Empty,Empty,BQueen,BBishop,Empty,Empty,Empty]
[Empty,Empty,WQueen,WBishop,Empty,Empty,Empty,Empty]
[Empty,Empty,Empty,Empty,BKnight,Empty,Empty,Empty]
[Empty,Empty,Empty,Empty,Empty,Empty,WPawn,Empty]
[WPawn,WPawn,WPawn,WPawn,WPawn,WPawn,Empty,WPawn]
[WRook,WKnight,WBishop,WQueen,WKing,WBishop,WKnight,WRook]
  79,504,549,432 bytes allocated in the heap
      71,260,944 bytes copied during GC
         361,536 bytes maximum residency (9 sample(s))
          82,960 bytes maximum slop
              32 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      3591 colls,  3591 par    0.369s   0.265s     0.0001s    0.0003s
  Gen  1         9 colls,     8 par    0.004s   0.002s     0.0003s    0.0003s

  Parallel GC work balance: 74.79% (serial 0%, perfect 100%)

  TASKS: 14 (1 bound, 13 peak workers (13 total), using -N6)

  SPARKS: 92 (92 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.001s  (  0.015s elapsed)
  MUT     time   55.638s  (  9.947s elapsed)
  GC      time    0.373s  (  0.267s elapsed)
  EXIT    time    0.000s  (  0.009s elapsed)
  Total   time   56.012s  ( 10.238s elapsed)

  Alloc rate    1,428,968,142 bytes per MUT second

  Productivity  99.3% of total user, 97.2% of total elapsed

michaellee@dyn-160-39-249-30 COMS4995 Project %
```

Slight slowdown to 10.238 seconds, but much better on spark count.

From these, we can see that the introduction of parallelizing the algorithm clearly increases the efficiency of the algorithm. With over 5.48 million board states to evaluate, the algorithm is able to complete traversal using this algorithm in ~10 seconds.

Another way to further improve this algorithm would be with the inclusion of alpha-beta pruning.

The implementation of alpha-beta pruning is incomplete, as the expected return value does not match the actual return value.

Using miniMaxParPrune at depth == 4 with 8 cores, the output is:

```
michaellee@dyn-160-39-249-30 COMS4995 Project % ./mini-max 5 4 1 -999 999 +RTS -s -N8
-999
[BRook,Empty,Empty,Empty,BKing,BBishop,Empty,BRook]
[BPawn,BPawn,BPawn,Empty,BPawn,BPawn,BPawn,BPawn]
[BKnight,Empty,Empty,BQueen,BBishop,Empty,Empty,Empty]
[Empty,Empty,WQueen,WBishop,Empty,Empty,Empty,Empty]
[Empty,Empty,Empty,Empty,BKnight,Empty,Empty,Empty]
[Empty,Empty,Empty,Empty,Empty,Empty,WPawn,Empty]
[WPawn,WPawn,WPawn,WPawn,WPawn,WPawn,Empty,WPawn]
[WRook,WKnight,WBishop,WQueen,WKing,WBishop,WKnight,WRook]
  79,508,362,584 bytes allocated in the heap
      72,331,296 bytes copied during GC
         470,904 bytes maximum residency (10 sample(s))
         114,504 bytes maximum slop
              43 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2868 colls,  2868 par    0.377s   0.224s     0.0001s    0.0004s
  Gen  1        10 colls,     9 par    0.006s   0.003s     0.0003s    0.0006s

  Parallel GC work balance: 68.90% (serial 0%, perfect 100%)

  TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

  SPARKS: 92 (92 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.001s  (  0.008s elapsed)
  MUT     time   54.944s  (  7.897s elapsed)
  GC      time    0.383s  (  0.227s elapsed)
  EXIT    time    0.000s  (  0.003s elapsed)
  Total   time   55.328s  (  8.135s elapsed)

  Alloc rate    1,447,077,164 bytes per MUT second

  Productivity  99.3% of total user, 97.1% of total elapsed

michaellee@dyn-160-39-249-30 COMS4995 Project %
```

While there is a speedup, the evaluation heuristic is wrong, meaning the the implementation itself is incorrect.

Alpha beta pruning, in this project, would be a definite way to improve the efficiency of the algorithm. Alpha beta pruning works by comparing each child node for a given game state to a maximum or minimum value, and then using this value to decide whether or not the child is worth traversing.

Pruning branches by saving a maximum value to compare the evaluation metric cuts down the trees in the node by a significant degree. However, implementing this feature is difficult in the frame of parallel haskell, as separate threads may need to communicate a maximum value so that another thread knows to prune a game tree.

There are a multitude of ways in which this program can be optimized further:

- There is lots of tedious code designed to implement the rules of moves, board evaluation, and move-making. Many of these can be implemented with a similar parallel algorithm, where lists can be traversed in parallel in order to parse and sort data quicker.
- The granularity can be increased to optimize sparks. If parallelism is implemented until depth == 2, and then the rest of the algorithm was performed sequentially, this may be able to increase speed and spark count by a favorable amount.
- The chess implementation is bare; there are no castling, checkmates, or double-square pawn moves.

Thank you for taking the time to read this and view my project.

Here is the implementation for my code:

mini-max.hs:

```
import System.Exit
import System.Environment
import Control.Parallel.Strategies
import Data.List
import Control.Monad




data Piece =  WKing | WQueen | WBishop | WKnight | WRook | WPawn
           | BKing | BQueen | BBishop | BKnight | BRook | BPawn
           | Empty deriving (Eq, Show)

-- Define the type for a chess board, represented as a list of lists of pieces
type Board = [[Piece]]

-- Define the type for a move, represented as the starting and ending positions on the
board
type Move = ((Int, Int), (Int, Int))

outPutBoard :: Board -> IO ()
outPutBoard board = do
   mapM_ print (reverse board)

getPoints :: Piece -> Int
getPoints piece
| piece == WKing   || piece == BKing   = 9999
| piece == WQueen  || piece == BQueen  = 9
| piece == WBishop || piece == BBishop = 3
| piece == WKnight || piece == BKnight = 3
| piece == WRook   || piece == BRook   = 5
| piece == WPawn   || piece == BPawn   = 1
| otherwise = 0

sumPieceList :: [Piece] -> Int
sumPieceList [] = 0
sumPieceList (x:xs) = (getPoints x) + (sumPieceList xs)
```

```haskell
-- Function to evaluate the value of a given board
-- Higher values indicate a better position for the player
-- Negative (better for -1 black)
-- positive (better for +1 white)
--Possible algorithm, how much total material is there on the board for each player?
King is negligible
-- pawn is 1
-- knight and bishop are 3
-- Rook is 5
-- Queen is 9
--Basically just counts the piece count.
evalBoard :: Board -> Int
evalBoard board = whiteCount - blackCount
   where
       whiteList = [board !! x !! y | x <- [0..7], y <- [0..7], snd (isOccupied board
(x, y)) == 1]
       blackList = [board !! x !! y | x <- [0..7], y <- [0..7], snd (isOccupied board
(x, y)) == (-1)]
       whiteCount = sumPieceList whiteList
       blackCount = sumPieceList blackList



--Checks if an index on the board is occupied and if its within bounds.
-- IF its not in bounds, a tuple will be returned with the int as code 999, which
signifies an out-of bounds
-- call without stopping the board evaluation.

isOccupied :: Board -> (Int, Int) -> (Bool, Int)
isOccupied board (x, y)
| x < 0 || x > 7 || y < 0 || y > 7 = (True, 999)
| (board !! x) !! y == Empty = (False, 0)
| head (show piece) == 'W' = (True, 1)
| otherwise = (True, -1)
   where
       piece = board !! x !! y


--Custom function that traverses a list, and returns a tuple of the list and the index
at which it had to stop.
takeWhileInclusive :: (a -> Bool) -> [a] -> [a]
```

```
takeWhileInclusive _ []  = []
takeWhileInclusive p (x:xs) = x : if p x then takeWhileInclusive p xs
                                         else []

checkLastElemCap :: Board -> Int -> [Move] -> [Move]
checkLastElemCap _ _ [] = []
checkLastElemCap board player list
| snd (isOccupied board (snd (last list))) == (-player) = list
| otherwise = fst (splitAt (length list - 1) list)


--Wrote this while working on rook moves, is applicable to any list of moves for any
piece (including knight). Takes board, int for player designation,
getListOfMoves :: Board -> Int -> [Move] -> [Move]
getListOfMoves board player moves = checkLastElemCap board player ((takeWhileInclusive
(\((r, c), (r', c')) -> (fst (isOccupied board (r', c')) == False)) moves))


--Generates legal moves for Pawns, regardless of whether or not the pawn is white or
black.
--This is essentially a filter function that takes a position and returns the legal
moves for the piece at a set of coordinates.
-- Either a WPawn or a BPawn will be passed in.


pawnLegalMoves :: Board -> Int -> Piece -> (Int, Int) -> [Move]
pawnLegalMoves board player piece (row, col)
| piece == WPawn = [x | x <- wMoveList, let check = snd (isOccupied board (snd x)),
check == 0 && check /= 999] ++
                   [y | y <- wCapList, let check = snd (isOccupied board (snd y)),
check == (-player) && check /= 999]
| piece == BPawn = [x | x <- bMoveList, let check = snd (isOccupied board (snd x)),
check == 0 && check /= 999] ++
                   [y | y <- bCapList, let check = snd (isOccupied board (snd y)),
check == (-player) && check /= 999]
| otherwise = []
   where
       wMoveList = [((row, col),(row + 1, col))]
       wCapList =  [((row, col),(row + 1, col + 1)), ((row, col),(row + 1, col - 1))]
       bMoveList = [((row, col),(row - 1, col))]
       bCapList =  [((row, col),(row - 1, col - 1)), ((row, col),(row - 1, col + 1))]
```

```
rookLegalMoves :: Board -> Int -> Piece -> (Int, Int) -> [Move]
rookLegalMoves board player piece (row, col)
| piece == WRook || piece == WQueen = (getListOfMoves board player vertNegMoveList) ++
(getListOfMoves board player vertPosMoveList) ++ (getListOfMoves board player
horiNegMoveList) ++ (getListOfMoves board player horiPosMoveList)
| piece == BRook || piece == BQueen = (getListOfMoves board player vertNegMoveList) ++
(getListOfMoves board player vertPosMoveList) ++ (getListOfMoves board player
horiNegMoveList) ++ (getListOfMoves board player horiPosMoveList)
| otherwise = []
   where
       vertNegMoveList = [((row, col),(row - 1, col)), ((row, col),(row - 2, col)),
((row, col),(row - 3, col)), ((row, col),(row - 4, col)), ((row, col),(row - 5, col)),
((row, col),(row - 6, col)), ((row, col),(row - 7, col))]
       vertPosMoveList = [((row, col),(row + 1, col)), ((row, col),(row + 2, col)),
((row, col),(row + 3, col)), ((row, col),(row + 4, col)), ((row, col),(row + 5, col)),
((row, col),(row + 6, col)), ((row, col),(row + 7, col))]
       horiNegMoveList = [((row, col),(row, col - 1)), ((row, col),(row, col - 2)),
((row, col),(row, col - 3)), ((row, col),(row, col - 4)), ((row, col),(row, col - 5)),
((row, col),(row, col - 6)), ((row, col),(row, col - 7))]
       horiPosMoveList = [((row, col),(row, col + 1)), ((row, col),(row, col + 2)),
((row, col),(row, col + 3)), ((row, col),(row, col + 4)), ((row, col),(row, col + 5)),
((row, col),(row, col + 6)), ((row, col),(row, col + 7))]

--This move list can just use a simple list comprehension: If the move is on a square
that is empty or the opposite color or inbounds, it counts.
knightLegalMoves :: Board -> Int -> Piece -> (Int, Int) -> [Move]
knightLegalMoves board player piece (row, col)
| piece == WKnight || piece == BKnight = [x | x <- moveList, let check = snd
(isOccupied board (snd x)), check == 0 || check == (-player)]
| otherwise = []
   where
       moveList = [((row, col),(row + 2, col + 1)), ((row, col),(row + 2, col - 1)),
((row, col),(row - 2, col + 1)), ((row, col),(row - 2, col + 1)),
                   ((row, col),(row + 1, col + 2)), ((row, col),(row - 1, col + 2)),
((row, col),(row + 1, col - 2)), ((row, col),(row - 1, col - 2))]

bishopLegalMoves :: Board -> Int -> Piece -> (Int, Int) -> [Move]
bishopLegalMoves board player piece (row, col)
| piece == WBishop || piece == WQueen = (getListOfMoves board player ppMoveList) ++
(getListOfMoves board player pnMoveList) ++ (getListOfMoves board player npMpveList)
++(getListOfMoves board player nnMoveList)
```

```
| piece == BBishop || piece == BQueen = (getListOfMoves board player ppMoveList) ++
(getListOfMoves board player pnMoveList) ++ (getListOfMoves board player npMpveList)
++(getListOfMoves board player nnMoveList)
| otherwise = []
   where
       ppMoveList = [((row, col),(row + 1, col + 1)), ((row, col),(row + 2, col + 2)),
((row, col),(row + 3, col + 3)), ((row, col),(row + 4, col + 4)), ((row, col),(row +
5, col + 5)), ((row, col),(row + 6, col + 6)), ((row, col),(row + 7, col + 7))]
       pnMoveList = [((row, col),(row + 1, col - 1)), ((row, col),(row + 2, col - 2)),
((row, col),(row + 3, col - 3)), ((row, col),(row + 4, col - 4)), ((row, col),(row +
5, col - 5)), ((row, col),(row + 6, col - 6)), ((row, col),(row + 7, col - 7))]
       npMpveList = [((row, col),(row - 1, col + 1)), ((row, col),(row - 2, col + 2)),
((row, col),(row - 3, col + 3)), ((row, col),(row - 4, col + 4)), ((row, col),(row -
5, col + 5)), ((row, col),(row - 6, col + 6)), ((row, col),(row - 7, col + 7))]
       nnMoveList = [((row, col),(row - 1, col - 1)), ((row, col),(row - 2, col - 2)),
((row, col),(row - 3, col - 3)), ((row, col),(row - 4, col - 4)), ((row, col),(row -
5, col - 5)), ((row, col),(row - 6, col - 6)), ((row, col),(row - 7, col - 7))]

queenLegalMoves :: Board -> Int -> Piece -> (Int, Int) -> [Move]
queenLegalMoves board player piece (row, col)
| piece == WQueen = (rookLegalMoves board player piece (row, col)) ++
(bishopLegalMoves board player piece (row, col))
| piece == BQueen = (rookLegalMoves board player piece (row, col)) ++
(bishopLegalMoves board player piece (row, col))
| otherwise = []

kingLegalMoves :: Board -> Int -> Piece -> (Int, Int) -> [Move]
kingLegalMoves board player piece (row, col)
| piece == WKing || piece == BKing = [x | x <- moveList, let check = snd (isOccupied
board (snd x)), check == 0 || check == (-player)]
| otherwise = []
   where
       moveList = [((row, col), (row + 1, col)), ((row, col), (row - 1, col)), ((row,
col), (row, col + 1)),
                   ((row, col), (row, col - 1)), ((row, col), (row + 1, col + 1)),
((row, col), (row - 1, col - 1))]

--This one is tricky, because the king cannot move into a square that is being
attacked.
--you may have to generate the list of all acceptable legal moves makeable by the
enemy,
```

```haskell
-- and then check if the list of king moves is in that list. --Or, just don't move the king at all.
-- that's the easiest way hahaha
-- maybe make the king the last piece you'd ever want to move. like, the engine would never think about `
--Just don't move the king. It's so much easier if you just don't touch it.


-- Modifies the board to make a new board, with a move being made.
makeMove :: Board -> Move -> Board
makeMove board ((r, c), (dr, dc)) =
   let piece = board !! r !! c
       (r', c') = splitAt c (board !! r)
       (moda, modb) = splitAt r board
       modPreRow =  r' ++ [Empty] ++ (snd (splitAt 1 c'))
       modPreBoard = moda ++ [modPreRow] ++ (snd (splitAt 1 modb))

       (dr', dc') = splitAt dc (modPreBoard !! dr)
       (modc, modd) = splitAt dr modPreBoard
       modPostRow = dr' ++ [piece] ++ (snd (splitAt 1 dc'))
       modPostBoard = modc ++ [modPostRow] ++ (snd (splitAt 1 modd))
   in
       modPostBoard

generateMovesHelper :: Board -> Int -> [(Piece , (Int, Int))] -> [Move]
generateMovesHelper _ _ [] = []
generateMovesHelper board player (x:xs) =
                            (pawnLegalMoves board player (fst x) (snd x)) ++
                            (knightLegalMoves board player (fst x) (snd x)) ++
                            (bishopLegalMoves board player (fst x) (snd x)) ++
                            (rookLegalMoves board player (fst x) (snd x)) ++
                            (queenLegalMoves board player (fst x) (snd x)) ++
                            (kingLegalMoves board player (fst x) (snd x)) ++
                            (generateMovesHelper board player xs)


--Generates all possible moves for a player (white or black) on a given board.

--Try to make it all one continuous list.
generateMoves :: Board -> Int -> [Move]
generateMoves board player = generateMovesHelper board player pieceList
   where
```

```
      pieceList = [((board !! x !! y), (x, y)) | x <- [0..7], y <- [0..7], snd
(isOccupied board (x, y)) == player]



-- Minimax function, which takes the current board, the depth of the search,
-- and the player who is currently making a move (1 for player, -1 for opponent)
-- Structure is that for each move, the minimax algorithm is called. you don't
actually have to change the minimax algo
-- For each move that is created by generate Moves, run minimax. This is where
parallelization occurs.

--Struggles heavily past a depth of 3.
--for correct usage, depth must be an odd number. This way, the function evaluates the
correct player, and returns
--the evaluation for the result of the corresponding player's move as opposed to the
opposite player.

minimaxSeq :: Board -> Int -> Int -> Int
minimaxSeq board depth player =
 if depth == 0
   then evalBoard board
   else
     let moves = generateMoves board player
         -- Recursively evaluate each possible move, using the opposite player.
         -- this applies minimax to every move.
         values = map (\move -> minimaxSeq (makeMove board move) (depth-1) (-player))
moves
     in
       -- If the current player is maximizing, return the maximum value
       if player == 1
         then maximum values
         else minimum values

--Minimax SEQ with ALPHA-BETA pruning
minimaxSeqPrune :: Board -> Int -> Int -> Int -> Int -> Int
minimaxSeqPrune board depth player alpha beta =
 if depth == 0
   then evalBoard board
   else
     let moves = generateMoves board player
         maxBestVal = minBound :: Int
```

```
        minBestVal = maxBound :: Int
        values = map (\move -> minimaxSeqPrune (makeMove board move) (depth-1)
(-player) alpha beta) moves
        value = maximum values
        --minBest = min minBestVal value
    in
      if player == 1
        then
          let alphBest = (max alpha (max maxBestVal value))
          in
              if alphBest >= beta
                  then beta
              else
                  max maxBestVal value
        else
          let minBest = min minBestVal value
              betaBest = min beta minBest
          in
              if betaBest >= alpha
                  then alpha
              else
                  minBest



--Parallel Function
--SPARKS for every single node (move) generated.
-- Granularity is incredibly high, too costly to be so much less efficient.

minimaxParMap :: Board -> Int -> Int -> Int
minimaxParMap board depth player =
 if depth == 0
   then evalBoard board
   else
     let moves = generateMoves board player
         -- Recursively evaluate each possible move, using the opposite player.
         -- this applies minimax to every move.
         values = parMap rpar (\move -> minimaxParMap (makeMove board move) (depth-1)
(-player)) moves
     in
       -- If the current player is maximizing, return the maximum value
       if player == 1
```

```
        then maximum values
        else minimum values

minimaxParGran :: Board -> Int -> Int -> Int
minimaxParGran board depth player=
   let
       moves = generateMoves board player
       values = parMap rpar (\move -> minimaxSeq (makeMove board move) (depth-1)
(-player)) moves
   in
       if player == 1
        then maximum values
        else minimum values


minimaxParPrune :: Board -> Int -> Int -> Int -> Int -> Int
minimaxParPrune board depth player alpha beta =
   let
       moves = generateMoves board player
       values = parMap rpar (\move -> minimaxSeqPrune (makeMove board move) (depth-1)
(-player) alpha beta) moves
   in
       if player == 1
        then maximum values
        else minimum values



main :: IO ()
main = do
   --print("Usage: minimax type (1 - minimaxSeq, 2-minimaxSeqPrune, 3-minimaxParMap,
4-minimaxParGran) boardname depth player alpha beta")
   argNames <- getArgs

   let minimax = read (argNames !! 0) :: Integer
       board = [[WRook, WKnight, WBishop, WQueen, WKing, WBishop, WKnight, WRook],
                [WPawn, WPawn, WPawn, WPawn, WPawn, WPawn, Empty, WPawn],
                [Empty, Empty, Empty, Empty, Empty, Empty, WPawn, Empty],
                [Empty, Empty, Empty, Empty, BKnight, Empty, Empty, Empty],
                [Empty, Empty, WQueen, WBishop, Empty, Empty, Empty, Empty],
                [BKnight, Empty, Empty, BQueen, BBishop, Empty, Empty, Empty],
                [BPawn, BPawn, BPawn, Empty, BPawn, BPawn, BPawn, BPawn],
```

```
              [BRook, Empty, Empty, Empty, BKing, BBishop, Empty, BRook]]
    depth = read (argNames !! 1) :: Int
    player = read (argNames !! 2) :: Int
    alpha = read (argNames !! 3) :: Int
    beta = read (argNames !! 4) :: Int

when(minimax == 1) $ print (minimaxSeq board depth player)
when(minimax == 2) $ print (minimaxSeqPrune board depth player alpha beta)
when(minimax == 3) $ print (minimaxParMap board depth player)
when(minimax == 4) $ print (minimaxParGran board depth player)
when(minimax == 5) $ print (minimaxParPrune board depth player alpha beta)

outPutBoard board
```