Parallelized Min-Max Chess Engine (PM)*

Feitong Qiao (flq2101), Yuanyuting Wang (yw3241)

December 2021

1 Introduction

The primary objective of a simple chess engine, given the state of a chess game as input, is to generate an optimal next move for the current player based on preset heuristics. To that end, algorithms like minimax and alpha-beta pruning are commonly used: the core idea is to construct a tree of possible next moves of the 2 players, and by recursively evaluating the chess boards at different nodes, to find the child that could lead to an end state with optimal accumulative "score" for the player. That node would then be the optimal next move that the program returns.

While conventionally this algorithm is sequentially executed, parts of the algorithm, especially the construction and evaluation of the game trees, are computationally heavy and could potentially improve in execution time through parallelization. In this project, we aimed to construct a simple sequential chess engine based on the aforementioned algorithms, and then to optimize its execution time through various parallelization strategies.

2 Problem Formulation

The basic data structure used throughout the program is a minimax tree where each node represents a potential game state (current board and player). A minimax tree has a conventional tree structure, except that each layer represents either a "maximizing" (White) or "minimizing" (Black) player. In this way, each node selects the child leading to the optimal end state by its standards, and in the end, the program simply outputs the first-level child node with the optimal score as the next move.

A program using pure minimiax algorithm would traverse the entirety of the tree of possible game states to an arbitrary depth, which could be highly computationally heavy, as the number of possible game states increase exponentially

*If it doesn't work so well, this name refers to Parallel Min-Max; otherwise, please call it Parallel Master (inspired by Grandmaster (GM) in the international chess ranking system) over the levels. In this project, we tested out three main approaches that could improve the computation time:

1. *Simple parallelization of the computation threads.* That is, utilize Haskell's parallel evaluation strategies to generate a thread/spark per node, which would be dedicated to the evaluation of the subtree under that node. With this method, we also experimented for the optimal ratio of sequential vs. parallel levels of the tree during evaluation.

2. *Adaption of alpha-beta pruning.* This sequential algorithm specializes in "pruning" the minimax tree by discarding nodes that represent obviously subpar options that are very unlikely to appear in actual gameplay. While this method significantly increases exec time by reducing the size of the minimax tree to traverse, its evaluation of each node is dependent on the *alpha* and *beta* values accumulated through evaluation of preceding nodes, which makes it hard to parallelize.

3. *A combination of both methods.* While we want to adapt alpha-beta pruning to construct the most time-efficient minimax trees, we also want to utilize the resources of multiple cores. Therefore, we experimented by creating parallel threads, and using a sequentially alpha-beta pruned minimax tree on each thread.

In this project, we aimed to implement the 3 strategies with two primary objectives: to provide maximum program execution time reduction, and also to compare the performances of the strategies in an investigation of the trade-off between sequential, algorithmic optimizations and multi-threaded workload distribution through parallelization.

3 Implementation

3.1 Chess

To implement a chess engine, we first needs to have an implementation of the chess game itself. The following data types are defined in the `Chess` module:

```haskell
data Game = Game
  { gamePlayer :: Player
  , gameBoard  :: Board
  }
  deriving (Read, Show, Eq)

data Player = Black | White deriving (Read, Show, Eq)

newtype Board = Board (Matrix BoardPiece) deriving Eq

type BoardPiece = Maybe (Player, Piece)

data Piece =
    Pawn
  | Knight
  | Bishop
  | Rook
  | Queen
  | King
  deriving (Read, Show, Eq)

type Position = (Int, Int)
```

In other components of the game engine, the interfaces mostly use the `Game` data type to represent the state of the game. The `Chess` module also defines useful helpers that, for example, gets the `BoardPiece` at a position, defines the default initial game state, pretty-prints the current game state, etc.

3.2 Chess Rules

The `Rules` module contains the following important functions:

- `isGameOver :: Game -> Bool`: determines whether the game has been won by a player
- `winner :: Game -> Maybe Player`: returns the winning Player, if there is one
- `legalMoves :: Game -> [Game]`: given a game state, return a list of next legal game states

The `legalMoves` function is especially important for this project, because it is the function that is used to produce the branches in a minimax search tree.

3.3 Board Score

To determine a best move in our chess engine algorithm, we need to assign score to a particular game board. The `Score` module provides the following function that evaluates the score of a `Game`:

- `gameScore :: Game -> Score`

Note that the `Score` type is defined to be a `Float`:

```
type Score = Float
```

The Game score is the sum of all BoardPiece scores. A BoardPiece score is calculated as this:

- an empty BoardPiece is 0
- a White BoardPiece is the sum of piece score and position score
- a Black BoardPiece is the negative value of the sum of piece score and position bonus
- piece score: Pawn = 10, Knight = 30, Bishop = 30, Rook = 50, Queen = 90, King = 900
- position bonus: given a chess piece, the position it is currently in on the chessboard also matters. For example, a Rook is more powerful in a central position, but quite weak in the corners. To reflect this, we use the `positionBonus :: BoardPiece -> Position -> Score` function to calculate the position bonus score. Note that this bonus can be negative to discourage disadvantageous positions.

3.4 Best-move Search

The best-move searching algorithm is the heart of the project. The base idea/algorithm that this chess engine builds upon is the minimax search algorithm. There are currently a total of 4 versions of the search algorithm:

- Sequential minimax (Minimax.Seq)
- Parallel minimax (Minimax.Par)
- Sequential minimax with alpha-beta pruning (Minimax.SeqAB)
- Parallel minimax with alpha-beta pruning (Minimax.ParAB)

Each of these 4 modules contains a submodule called `Move` that exports a `bestMove` function. However, since these searching strategy have different parameters (e.g. parallel depth), the outer `Minimax.Move` module defines the following function to have a cleaner interface:

```
bestMove :: PMStrategy -> Game -> Game

data PMStrategy
  = MinimaxSeq Depth -- depth
  | MinimaxPar Depth Depth -- parDepth, depth
  | MinimaxSeqAB Depth -- depth
  | MinimaxParAB Depth Depth -- parDepth, depth
  deriving (Read, Show, Eq)
```

Note that the `Depth` type here is defined as an `Integer` in the `Minimax.Common` module.

3.5 Sequential Minimax (MinimaxSeq)

The `MinimaxSeq` search strategy is the base minimax algorithm. It takes the following parameter:

- depth: the number of moves to look into the future; in other words, it is the depth of the game tree to search

The minimax algorithm is naturally recursive. Without loss of generality, suppose the minimax algorithm is determining the best move for Black player. It tries to evaluate the score of the game state of each possible next move, and it would select the move that yields the minimum score. It assumes that White player is also trying to optimize their move; hence White would consider all possible next moves and select the move that yields the maximum score. But White player would hold the same assumptions about Black, and the same reasoning occurs again. Each of these turns is a level in our game tree, and when we reach the level of the parametrized depth, the score is calculated using the `gameScore` function instead of recursing on minimax.

The implementation of the algorithm is rather straightforward and can be found in the `Minimax.Seq.Move` module.

3.6 Parallel Minimax (MinimaxPar)

The MinimaxPar search strategy is the parallel version of minimax algorithm. It takes the following parameters:

- parDepth: the depth of spark generating minimax
- depth: the total depth of the game tree

Similar to the depth parameter, we decrease the parDepth parameter by 1 each time we recurse. The main difference in this version is that, depending on whether parDepth is greater than 0, different Eval Strategies are applied to the evaluation of the scores variable:

```
minimax :: Depth -> Depth -> Game -> Score
minimax parDepth depth g
  | depth > 0
  = let
      evalStrat = if parDepth > 0 then parList rseq else rseq
      scores =
        map (minimax (parDepth - 1) (depth - 1)) (legalMoves g)
          `using` evalStrat
      optimalScore =
        if shouldMaximize g then maximum scores else minimum scores
    in
      optimalScore
  | otherwise
  = gameScore g
```

If parDepth is greater than 0, the evaluations of each subtree is sparked and run in parallel; otherwise, they are run sequentially.

3.7 Sequential Minimax with Alpha-Beta Pruning (MinimaxSeqAB)

The MinimaxSeqAB search strategy add alpha-beta pruning optimization to the base minimax algorithm. It takes the same depth parameter as MinimaxSeq:

- depth: the number of moves to look into the future; in other words, it is the depth of the game tree to search

The addition of alpha-beta pruning does not change the output of the minimax algorithm; it only takes advantage of an observation to prevent unneeded exploration of the game tree. The observation is that when the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta ¡ alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play. [1]

3.8 Parallel Minimax with Alpha-Beta Pruning (Minimax-ParAB)

The MinimaxParAB search strategy is the parallel version of minimax with alpha-beta pruning algorithm. It takes the following parameters:

[1] https://en.wikipedia.org/wiki/Alpha-beta_pruning

- parDepth: the depth of spark generating minimax
- depth: the total depth of the game tree

For this implementation, we essentially run the parallel minimax for parDepth levels, and run the remaining (depth - parDepth) levels with the sequential minimax with alpha-beta pruning.

4 Evaluation

4.1 Experiment Settings

All the following measurements are performed on a Macbook Pro (16-inch, 2019) with the 2.3 GHz 8-Core Intel Core i9, and a memory of 16 GB 2667 MHz DDR4.

We ran the program using a series of different strategies with the following codenames and specifications:

1. `Seq,depth::Int`: Simple sequential minimax tree traversal until `depth`.
2. `Par,parDepth::Int,depth::Int`: Parallel threads generated for each node until `parDepth`, and then simple sequential minimax tree traversal until the fixed overall depth reaches `depth`.
3. `SeqAB,depth::Int`: Alpha-beta pruned sequential minimax tree until `depth`.
4. `ParAB,parDepth::Int,depth::Int`: Parallel thread generation until `parDepth`, and then sequential traversal of alpha-beta pruned minimax tree until overall depth of `depth`.

Additionally, each test is run with a given starting chess board, and completes after the program executes twice (i.e. generate the optimal move for 2 consecutive player turns). The starting chess board can be one of the following:

1. *B1*: Default clean chess board.
2. *B2*: The Sicilian Defense opening[2].
3. *B3*: A combination by Phillip Stamma[3].
4. *B4*: The Ruy Lopez opening[4].

4.2 Execution Time Analysis

We ran the program on different chess boards using the various aforementioned strategies, with a fixed tree depth of 4. The resultant execution time statistics is shown in Table 1.

Based on the results, several observations can be made:

[2]https://www.chess.com/openings/Sicilian-Defense
[3]https://thechessworld.com/articles/problems/7-most-famous-chess-combinations/
[4]https://www.chess.com/openings/Ruy-Lopez-Opening

Total Time (s)	*B1*	*B2*	*B3*	*B4*	Avg. % of Seq time
`Seq,4`	12.5	28.7	111	54.1	NA
`Par,1,4`	2.21	4.98	15.6	8.32	16.1%
`Par,2,4`	2.13	4.10	14.5	7.28	14.5%
`SeqAB,4`	1.96	3.90	17.9	12.83	17.3%
`ParAB,1,4`	0.69	1.26	3.48	2.42	4.38%
`ParAB,2,4`	0.81	1.49	3.94	2.86	5.13%

Table 1: Execution time comparison of different stratgies.

1. All 3 optimizing methods provided substantial improvement to the execution time of the program, with a exec time reduction of 85% − 95%.

2. Regarding the `Par` strategy, increasing the levels of parallelization from 1 to 2 only offered insignificant improvements. This is potentially because with the workload of multiple levels of tree nodes being put into parallel threads, the workload of nodes in the upper levels become insignificant (simply calling spark generation on all of its child nodes).

3. Comparing `Par` and `SeqAB` strategies, we discovered that their exec time improvements are on a similar magnitude, which went to suggest that a strong sequential optimization to the algorithm was on the same par performance-wise as a straightforward parallelization of work threads for this program.

4. Regarding the `ParAB` strategy, it offered the most exec time improvement out of all strategies (95%), proving that this combination of sequential and parallel strategies did improve the outcome by quite a margin, instead of being counterproductive.

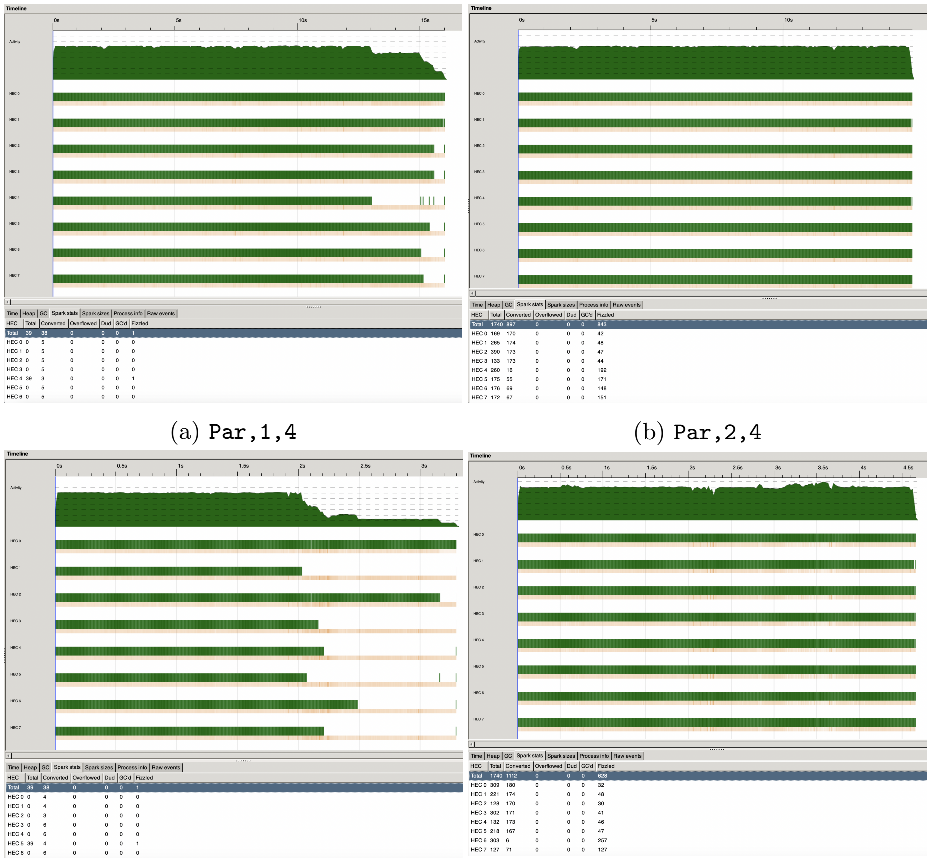
   However, we noticed having 2 parallel levels was suboptimal compared to having only 1. This is potentially due to 1) the same reason as for the mild difference between `Par,1,4` and `Par,2,4` and 2) the fact that increasing a parallel level means "losing" a level that could have been utilized by the pruning algorithm, which is more potent when it has access to more information in the tree.

4.3 Spark Generation and Load-Balancing Analysis

With a given starting board (*B3*), we ran the program using `Par` and `ParAB` strategies; the resultant event log and visualization via Threadscope[5] is shown in Figure 1.

Based on the logs, several observations regarding load-balancing efficacy of the strategies:

[5] https://github.com/haskell/ThreadScope



(a) `Par,1,4` (b) `Par,2,4`

(c) `ParAB,1,4` (d) `ParAB,2,4`

Figure 1: Threadscope visualizations of program run.

1. Both `Par,1,4` and `ParAB,1,4` resulted in the generation of 39 sparks, which is supposedly the number of legal next moves given the start state (i.e. immediate child nodes of the tree roo). `Par,2,4` and `ParAB,2,4` resulted in the generation of 1740 sparks, which is supposedly the number of nodes on the second AND third layers of the tree. This matches our expectation.

2. Given the greater number of sparks generated, strategies with 2 parallel levels did better in terms of load-balancing than those with 1, as workload is divided more granularly over a larger number of sparks. Throughout the course of the execution, the work done on the 8 cores for `Par,2,4` and `ParAB,2,4` remained consistent without obvious gaps (except for at the end, small gaps to make up for minor exec time differences). There was also no significant surges of garbage collection.

3. Regarding strategies with only 1 parallel level: given only 39 sparks in total, there was only 4-6 sparks of heavy computation distributed to each core. More susceptible to the fluctuations of the computation time of each thread, load balancing for `Par,1,4` and `ParAB,1,4` was thus worse.

   This phenomenon is especially pronounced in `ParAB,1,4`: depending on the actual outcome of alpha-beta pruning on different subtrees, the computation time for each spark (one sequential traversal of a pruned tree) could vary greatly. This leads to a general difficulty to reasonably balance workload without more sophisticated preemptive workload-estimating mechanisms to guide the balancing.

4.4 Conclusion and Discussion

As the above analysis has shown, the combination of parallel threading and sequential alpha-beta pruning strategies was successfully in reducing the program execution time by around 95%. Otherwise, pure parallelization and sequential alpha-beta pruning methods yielded similar improvements of around 85%. It can then be argued that a careful combination of parallel and sequential strategies could yield the maximal efficacy of optimization for this problem of constructing a chess engine.

However, considering mainly the parallel strategies `Par` and `ParAB`, we noticed that there's an interesting trade-off between having 1 or 2 parallel levels: having 1 level results in suboptimal load-balancing, whereas having 2 levels trivializes the work done in most sparks, and hence unnecessarily creates a large number of sparks without actually increasing the efficiency of each core.

To address this shortcoming, one possible method would be having a user-input parameter that acts as the cap for threadcounts to more precisely control the number of sparks to generate and distribute onto the cores. In the traversal of the minimax tree, we could enforce the threadcount by dividing up the nodes to be traversed into "chunks", assigning each chunk to a dedicated thread. We can then do a breadth-first-search, instead of depth-first-search, of the tree, where we list the nodes in each level, evaluate all the nodes in chunks, and then synchronize and collapse the resultant "child scores" back into the root node through our maximizing/minimizing heuristics.

Another potential method that could optimize the engine is Principle Variation Splitting, an advance and more chess-specific parallel algorithm[6]. This method approximates a "parallel version" of alpha-beta pruning and is another way of bringing together the sequential pruning process and efficient parallelization. In the future, it would be interesting to try out both of these methods and compare the results to the current strategies.

Appendix A Important Code Listings

`app/Main.hs`:

[6] http://worldcomp-proceedings.com/proc/p2011/SER3956.pdf

```haskell
module Main where

import           System.Console.GetOpt           ( ArgDescr
                                                   ( NoArg
                                                   , ReqArg
                                                   )
                                                 , ArgOrder
                                                   ( RequireOrder
                                                   )
                                                 , OptDescr(..)
                                                 , getOpt
                                                 , usageInfo
                                                 )
import           System.Environment              ( getArgs
                                                 , getProgName
                                                 )
import           System.Exit                     ( exitSuccess )
import           System.IO                       ( hFlush
                                                 , hPutStrLn
                                                 , print
                                                 , readFile
                                                 , stderr
                                                 , stdout
                                                 )

import           Chess                           ( Board(..)
                                                 , Game(..)
                                                 , Player(..)
                                                 , defaultBoard
                                                 , defaultGame
                                                 , parseBoard
                                                 , prettyBoard
                                                 )
import           Control.Monad                   ( unless )
import           Data.Char                       ( isSpace )
import           Data.List.Split                 ( splitOn )
import           Data.Monoid                     ( Alt(getAlt) )
import           Minimax.Common                  ( Depth )
import           Minimax.Move                    ( PMStrategy(..)
                                                 , bestMove
                                                 )
import           Rules                           ( isGameOver )


data Mode
  = Interactive
  | Test
  deriving (Read, Show, Eq)

data Options = Options
  { optPMStrategy :: PMStrategy
  , optMode       :: Mode
  , optPlayer     :: Player
  , optBoardSrc   :: String
  }
  deriving (Show, Eq)

defaultOptions :: Options
defaultOptions = Options { optPMStrategy = MinimaxSeq 5
                         , optMode       = Interactive
                         , optPlayer     = Black
                         , optBoardSrc   = ""
                         }

usage :: IO Options
usage = do
  prg <- getProgName
  let header = "Usage: " ++ prg ++ " [option]... [player] [file]"
  hPutStrLn stderr (usageInfo header options)
  exitSuccess

options :: [OptDescr (Options -> IO Options)]
options =
  [ Option "m"
           ["mode"]
           (ReqArg (\mode opt -> return opt { optMode = read mode }) "<mode>")
           "Mode to run the engine"
  , Option
    "s"
    ["strategy"]
    (ReqArg
      (\pmStrat opt ->
        let splitStrat = splitOn "," pmStrat
            pmStrat'   = case head splitStrat of
              "MinimaxSeq" -> MinimaxSeq (read $ splitStrat !! 1)
              "MinimaxPar" ->
                MinimaxPar (read $ splitStrat !! 1) (read $ splitStrat !! 2)
              "MinimaxSeqAB" -> MinimaxSeqAB (read $ splitStrat !! 1)
              "MinimaxParAB" -> MinimaxParAB (read $ splitStrat !! 1) (read $
                   splitStrat !! 2)
              _              -> error "Invalid PMStrategy"
```

```haskell
        in  return opt { optPMStrategy = pmStrat' }
      )
      "<strategy>"
    )
    "Strategy for minimax"
  , Option
    "p"
    ["player"]
    (ReqArg (\player opt -> return opt { optPlayer = read player }) "<player>")
    "Player that the engine is playing as"
  , Option
    "b"
    ["boardSrc"]
    (ReqArg (\boardSrc opt -> return opt { optBoardSrc = boardSrc })
            "<boardSrc>"
    )
    "File path specifying custom initial board layout"
  , Option "h" ["help"] (NoArg (const usage)) "Print help"
  ]

main :: IO ()
main = do
  args <- getArgs
  let (actions, filenames, errors) = getOpt RequireOrder options args
  opts <- foldl (>>=) (return defaultOptions) actions
  mapM_ putStrLn filenames
  print opts
  startGame opts
 where
  startGame opts@Options { optBoardSrc = src, optPlayer = player } = do
    g <- initGame src player
    loop 1 g opts
  initGame src player
    | null src || all isSpace src = return Game { gamePlayer = player
                                                , gameBoard  = defaultBoard
                                                }
    | otherwise = do
      contents <- readFile src
      return Game { gamePlayer = player, gameBoard = parseBoard contents }
  loop turn g opts = do
    unless (optMode opts == Test) $ do
      putStrLn
        $  "> Turn "
        ++ show turn
        ++ ", "
        ++ show (gamePlayer g)
        ++ "'s move:"
      putStrLn $ prettyBoard $ gameBoard g
      putStrLn ""
    unless ((turn >= 3 && optMode opts == Test) || isGameOver g) $ do
      let g' = bestMove (optPMStrategy opts) g
      loop (turn + 1) g' opts
```

src/Chess.hs:

```haskell
-- | Chess representations

module Chess
  ( Board(..)
  , board
  , BoardPiece(..)
  , Game(..)
  , Piece(..)
  , Position
  , Player(..)
  , atPos
  , getBoardMatrix
  , setBoardPiece
  , setPlayer
  , prettyGame
  , prettyBoard
  , parseBoard
  , defaultGame
  , defaultBoard
  ) where
import           Data.Bifunctor                            ( first )
import           Data.Char                                 ( toLower )
import           Data.List                                 ( intercalate )
import           Data.Matrix                               ( (!)
                                                           , Matrix
                                                           , fromLists
                                                           , matrix
                                                           , setElem
                                                           , toLists
                                                           , nrows
                                                           , ncols
                                                           )

data Game = Game
  { gamePlayer :: Player
```

```haskell
  , gameBoard  :: Board
  }
  deriving (Read, Show, Eq)

data Player = Black | White deriving (Read, Show, Eq)

newtype Board = Board (Matrix BoardPiece) deriving Eq

type BoardPiece = Maybe (Player, Piece)

instance Show Board where
  show (Board b) = show $ toLists b

instance Read Board where
  readsPrec prec s = map (first (Board . fromLists)) (readsPrec prec s)

-- checks dimension on the 2D list
board :: [[BoardPiece]] -> Board
board b = if isValidBoard
  then Board $ fromLists b
  else error "Dimension of board is not 8*8"
 where
  validNumOfRows    = length b == 8
  validNumOfColumns = all (\row -> length row == 8) b
  isValidBoard      = validNumOfRows && validNumOfColumns

data Piece =
   Pawn
  | Knight
  | Bishop
  | Rook
  | Queen
  | King
  deriving (Read, Show, Eq)

type Position = (Int, Int)


-- unsafe: get piece at position
atPos :: Game -> Position -> BoardPiece
atPos g pos = getBoardMatrix g ! pos

getBoardMatrix :: Game -> Matrix BoardPiece
getBoardMatrix Game { gameBoard = Board b } = b

-- update game board
setBoardPiece :: Game -> Position -> BoardPiece -> Game
setBoardPiece g@Game { gameBoard = Board b } pos bp =
  g { gameBoard = Board $ setElem bp pos b }

setPlayer :: Game -> Player -> Game
setPlayer g p = g { gamePlayer = p }


-- pretty print Game
prettyGame :: Game -> String
prettyGame g =
  "> Player: " ++ show (gamePlayer g) ++ "\n" ++ prettyBoard (gameBoard g)

prettyBoard :: Board -> String
prettyBoard (Board b) = intercalate "\n" . map prettyRow . toLists $ fmap
  prettyBoardPiece
  b
 where
  prettyRow row = "|" ++ intercalate "|" row ++ "|"
  prettyBoardPiece Nothing  = "  "
  prettyBoardPiece (Just p) = prettyPiece p
  prettyPiece (player, piece) =
    let player' = toLower . head . show $ player
        piece'  = toLower . head . show $ piece
    in  [player', piece']

parseBoard :: String -> Board
parseBoard text = case (nrows board, ncols board) of
                    (8, 8) -> Board board
                    _ -> error "ill-formatted initial board"
  where board = fromLists $ map parseRow (lines text)
        parseRow line = do w <- wordsWhen (==',') line
                           return $ parsePiece w
        parsePiece word = case word of
          " " -> Nothing
          (pl:pi:_) ->
            let player = case toLower pl of
                           'b' -> Black
                           'w' -> White
                           _ -> error ("invalid piece " ++ word) in
            let piece = case toLower pi of
                          'p' -> Pawn
                          'k' -> Knight
                          'b' -> Bishop
                          'r' -> Rook
```

```haskell
                              'q' -> Queen
                              'x' -> King
                              _ -> error ("invalid piece " ++ word) in
            Just (player, piece)
          _ -> error ("invalid piece " ++ word)

-- reference: https://stackoverflow.com/questions/4978578/how-to-split-a-string-in-haskell
wordsWhen     :: (Char -> Bool) -> String -> [String]
wordsWhen p s =  case dropWhile p s of
                      "" -> []
                      s' -> w : wordsWhen p s''
                            where (w, s'') = break p s'

-- default start game state
defaultGame :: Game
defaultGame = Game { gamePlayer = Black, gameBoard = defaultBoard }

defaultBoard :: Board
defaultBoard = board b
 where
  b =
    [ [ Just (Black, Rook)
      , Just (Black, Knight)
      , Just (Black, Bishop)
      , Just (Black, Queen)
      , Just (Black, King)
      , Just (Black, Bishop)
      , Just (Black, Knight)
      , Just (Black, Rook)
      ]
    , [ Just (Black, Pawn)
      , Just (Black, Pawn)
      , Just (Black, Pawn)
      , Just (Black, Pawn)
      , Just (Black, Pawn)
      , Just (Black, Pawn)
      , Just (Black, Pawn)
      , Just (Black, Pawn)
      ]
    , [Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing]
    , [Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing]
    , [Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing]
    , [Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing]
    , [ Just (White, Pawn)
      , Just (White, Pawn)
      , Just (White, Pawn)
      , Just (White, Pawn)
      , Just (White, Pawn)
      , Just (White, Pawn)
      , Just (White, Pawn)
      , Just (White, Pawn)
      ]
    , [ Just (White, Rook)
      , Just (White, Knight)
      , Just (White, Bishop)
      , Just (White, Queen)
      , Just (White, King)
      , Just (White, Bishop)
      , Just (White, Knight)
      , Just (White, Rook)
      ]
    ]
```

src/Rules.hs:

```haskell
-- | Chess rules

module Rules
    ( isGameOver
    , winner
    , legalMoves
    , legalMovesForPos
    ) where

import           Chess                                    ( Board(..)
                                                          , Game(..)
                                                          , Piece(..)
                                                          , Player(..)
                                                          , Position
                                                          )
import           Data.Foldable                            ( find )
import           Data.Matrix                              ( (!)
                                                          , setElem
                                                          )
import           Data.Maybe                               ( fromJust )

isGameOver :: Game -> Bool
isGameOver Game { gameBoard = Board b } = not $ hasBlackKing && hasWhiteKing
  where
```

```haskell
    hasBlackKing = Just (Black, King) `elem` b
    hasWhiteKing = Just (White, King) `elem` b

winner :: Game -> Maybe Player
winner g@Game { gameBoard = Board b } = if isGameOver g
    then Just . fst . fromJust . fromJust $ find isKing b
    else Nothing
  where
    isKing (Just (_, King)) = True
    isKing _                = False

legalMoves :: Game -> [Game]
legalMoves g =
    let allPositions = [ (r, c) | r <- [1 .. 8], c <- [1 .. 8] ]
    in  concatMap (legalMovesForPos g) allPositions


legalMovesForPos :: Game -> Position -> [Game]
legalMovesForPos g@Game { gamePlayer = player, gameBoard = Board b } pos@(r, c)
    = case b ! pos of
        Nothing -> []
        Just (piecePlayer, piece) ->
            if piecePlayer == player then movesForPiece piece else []
  where
    movesForPiece Pawn = case player of
        Black ->
            let normalStep = validEmpty [(r + 1, c)]
                doubleStep = if r == 2 && isEmpty (b ! (3, c))
                    then validEmpty [(r + 2, c)]
                    else []
                takes = validTake [(r + 1, c - 1), (r + 1, c + 1)]
                poses = normalStep ++ doubleStep ++ takes
                newBoards newpos@(r, c)
                    | r == 8
                    = let promotions = [Pawn, Knight, Bishop, Rook, Queen]
                      in  map (makeMove pos newpos) promotions
                    | otherwise
                    = [makeMove pos newpos Pawn]
            in  map makeGame $ concatMap newBoards poses
        White ->
            let normalStep = validEmpty [(r - 1, c)]
                doubleStep = if r == 7 && isEmpty (b ! (6, c))
                    then validEmpty [(r - 2, c)]
                    else []
                takes = validTake [(r - 1, c - 1), (r - 1, c + 1)]
                poses = normalStep ++ doubleStep ++ takes
                newBoards newpos@(r, c)
                    | r == 1
                    = let promotions = [Pawn, Knight, Bishop, Rook, Queen]
                      in  map (makeMove pos newpos) promotions
                    | otherwise
                    = [makeMove pos newpos Pawn]
            in  map makeGame $ concatMap newBoards poses
    movesForPiece Knight =
        let poses = validEmptyOrTake
                [ (r + 1, c + 2)
                , (r + 1, c - 2)
                , (r + 2, c + 1)
                , (r + 2, c - 1)
                , (r - 1, c + 2)
                , (r - 1, c - 2)
                , (r - 2, c + 1)
                , (r - 2, c - 1)
                ]
        in  map (\newpos -> makeGame $ makeMove pos newpos Knight) poses
    movesForPiece Bishop =
        let dir   = [(1, 1), (1, -1), (-1, 1), (-1, -1)]
            poses = concatMap (allPosInDirection 1) dir
        in  map (\newpos -> makeGame $ makeMove pos newpos Bishop) poses
    movesForPiece Rook =
        let dir   = [(1, 0), (-1, 0), (0, 1), (0, -1)]
            poses = concatMap (allPosInDirection 1) dir
        in  map (\newpos -> makeGame $ makeMove pos newpos Rook) poses
    movesForPiece Queen =
        let
            dir =
                [ (1 , 0)
                , (-1, 0)
                , (0 , 1)
                , (0 , -1)
                , (1 , 1)
                , (1 , -1)
                , (-1, 1)
                , (-1, -1)
                ]
            poses = concatMap (allPosInDirection 1) dir
        in
            map (\newpos -> makeGame $ makeMove pos newpos Queen) poses
    movesForPiece King =
        let poses = validEmptyOrTake
                [ (r', c')
```

```haskell
                    | r' <- [(r - 1) .. (r + 1)]
                    , c' <- [(c - 1) .. (c + 1)]
                    , (r', c') /= (r, c)
                    ]
            in  map (\newpos -> makeGame $ makeMove pos newpos King) poses
    validEmpty = filter (\pos -> isEmpty $ b ! pos) . filter inRange
    validTake  = filter (\pos -> isEnemy $ b ! pos) . filter inRange
    validEmptyOrTake =
        filter (\pos -> isEmpty (b ! pos) || isEnemy (b ! pos)) . filter inRange
    inRange (r, c) = r >= 1 && r <= 8 && c >= 1 && c <= 8
    isEmpty (Just _) = False
    isEmpty Nothing  = True
    isMine (Just (player', _)) = player' == player
    isMine Nothing             = False
    isEnemy (Just (player', _)) = player' == otherPlayer player
    isEnemy Nothing             = False
    allPosInDirection mult (rDir, cDir)
        | not (inRange newPos) = []
        | isEnemy dest         = [newPos]
        | isMine dest          = []
        | otherwise = newPos : allPosInDirection (mult + 1) (rDir, cDir)
      where
        newPos = (r + mult * rDir, c + mult * cDir)
        dest   = b ! newPos
    makeMove oldpos newpos newpiece =
        let b1 = setElem Nothing oldpos b
            b2 = setElem (Just (player, newpiece)) newpos b1
        in  Board b2
    makeGame b = Game { gamePlayer = otherPlayer player, gameBoard = b }


otherPlayer :: Player -> Player
otherPlayer Black = White
otherPlayer White = Black
```

src/Score.hs:

```haskell
-- | Chess board evaluation

module Score
    ( gameScore
    , boardScore
    , Score
    ) where
import           Chess                                          ( Board(..)
                                                                , BoardPiece(..)
                                                                , Game(..)
                                                                , Piece(..)
                                                                , Player(..)
                                                                , Position(..)
                                                                )
import           Data.Matrix                                    ( (!)
                                                                , Matrix(..)
                                                                , fromLists
                                                                , switchRows
                                                                )

type Score = Float

gameScore :: Game -> Score
gameScore g = boardScore $ gameBoard g

boardScore :: Board -> Score
boardScore (Board b) = foldl
    (\score pos -> score + positionScore (b ! pos) pos)
    0
    indicies
    where indicies = [ (r, c) | r <- [1 .. 8], c <- [1 .. 8] ]

positionScore :: BoardPiece -> Position -> Score
positionScore bp pos = score + bonus
  where
    score = boardPieceScore bp
    bonus = positionBonus bp pos

boardPieceScore :: BoardPiece -> Score
boardPieceScore Nothing            = 0
boardPieceScore (Just (Black, p)) = -1 * pieceScore p
boardPieceScore (Just (White, p)) = pieceScore p

pieceScore :: Piece -> Score
pieceScore Pawn   = 10
pieceScore Knight = 30
pieceScore Bishop = 30
pieceScore Rook   = 50
pieceScore Queen  = 90
pieceScore King   = 900

positionBonus :: BoardPiece -> Position -> Score
positionBonus Nothing                 _   = 0
```

```haskell
positionBonus (Just (player, piece)) pos = bonusMap player piece ! pos

bonusMap :: Player -> Piece -> Matrix Score
bonusMap White piece = pieceBonusMap piece
bonusMap Black piece = fmap negate $ reflectOverX $ pieceBonusMap piece
  where
    reflectOverX mat =
        switchRows 1 8 $ switchRows 2 7 $ switchRows 3 6 $ switchRows 4 5 mat
pieceBonusMap Pawn = fromLists
    [ [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    , [5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0]
    , [1.0, 1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0]
    , [0.5, 0.5, 1.0, 2.5, 2.5, 1.0, 0.5, 0.5]
    , [0.0, 0.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0]
    , [0.5, -0.5, -1.0, 0.0, 0.0, -1.0, -0.5, 0.5]
    , [0.5, 1.0, 1.0, -2.0, -2.0, 1.0, 1.0, 0.5]
    , [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    ]
pieceBonusMap Knight = fromLists
    [ [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
    , [-4.0, -2.0, 0.0, 0.0, 0.0, 0.0, -2.0, -4.0]
    , [-3.0, 0.0, 1.0, 1.5, 1.5, 1.0, 0.0, -3.0]
    , [-3.0, 0.5, 1.5, 2.0, 2.0, 1.5, 0.5, -3.0]
    , [-3.0, 0.0, 1.5, 2.0, 2.0, 1.5, 0.0, -3.0]
    , [-3.0, 0.5, 1.0, 1.5, 1.5, 1.0, 0.5, -3.0]
    , [-4.0, -2.0, 0.0, 0.5, 0.5, 0.0, -2.0, -4.0]
    , [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
    ]
pieceBonusMap Bishop = fromLists
    [ [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
    , [-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0]
    , [-1.0, 0.0, 0.5, 1.0, 1.0, 0.5, 0.0, -1.0]
    , [-1.0, 0.5, 0.5, 1.0, 1.0, 0.5, 0.5, -1.0]
    , [-1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, -1.0]
    , [-1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0]
    , [-1.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.5, -1.0]
    , [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
    ]
pieceBonusMap Rook = fromLists
    [ [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    , [0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5]
    , [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5]
    , [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5]
    , [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5]
    , [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5]
    , [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5]
    , [0.0, 0.0, 0.0, 0.5, 0.5, 0.0, 0.0, 0.0]
    ]
pieceBonusMap Queen = fromLists
    [ [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
    , [-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0]
    , [-1.0, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -1.0]
    , [-0.5, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -0.5]
    , [0.0, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -0.5]
    , [-1.0, 0.5, 0.5, 0.5, 0.5, 0.5, 0.0, -1.0]
    , [-1.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, -1.0]
    , [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
    ]
pieceBonusMap King = fromLists
    [ [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0]
    , [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0]
    , [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0]
    , [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0]
    , [-2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0]
    , [-1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0]
    , [2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 2.0, 2.0]
    , [2.0, 3.0, 1.0, 0.0, 0.0, 1.0, 3.0, 2.0]
    ]
```

src/Minimax/Common.hs:

```haskell
-- | Common minimax definitions

module Minimax.Common
  ( Depth
  ) where

type Depth = Int
```

src/Minimax/Move.hs:

```haskell
module Minimax.Move
  ( PMStrategy(..)
  , bestMove
  ) where

import           Chess                              ( Game(..) )
import           Minimax.Common                     ( Depth )
import qualified Minimax.Par.Move         as P
```

```haskell
import qualified Minimax.ParAB.Move            as PAB
import qualified Minimax.Seq.Move              as S
import qualified Minimax.SeqAB.Move            as SAB

data PMStrategy
  = MinimaxSeq Depth
  | MinimaxPar Depth Depth -- parDepth, depth
  | MinimaxSeqAB Depth
  | MinimaxParAB Depth Depth
  deriving (Read, Show, Eq)


bestMove :: PMStrategy -> Game -> Game
bestMove pmStrat g = case pmStrat of
  MinimaxSeq depth            -> S.bestMove depth g
  MinimaxPar parDepth depth   -> P.bestMove parDepth depth g
  MinimaxSeqAB depth          -> SAB.bestMove depth g
  MinimaxParAB parDepth depth -> PAB.bestMove parDepth depth g
```

src/Minimax/Seq/Move.hs:

```haskell
-- | Move generation by sequential minimax algorithm

module Minimax.Seq.Move
  ( bestMove
  ) where

import           Chess                                  ( Game(..)
                                                        , Player(..)
                                                        )
import           Minimax.Common                         ( Depth )
import           Rules                                  ( legalMoves )
import           Score                                  ( Score
                                                        , gameScore
                                                        )

bestMove :: Depth -> Game -> Game
bestMove d g =
  let movesWithScores = [ (move, minimax (d - 1) move) | move <- legalMoves g ]
      comparator      = if shouldMaximize g
        then \x@(_, xscore) y@(_, yscore) -> if xscore >= yscore then x else y
        else \x@(_, xscore) y@(_, yscore) -> if xscore <= yscore then x else y
      optimalMove = fst $ foldr1 comparator movesWithScores
  in  optimalMove

shouldMaximize :: Game -> Bool
shouldMaximize Game { gamePlayer = White } = True
shouldMaximize Game { gamePlayer = Black } = False

minimax :: Depth -> Game -> Score
minimax d g
  | d <= 0
  = gameScore g
  | otherwise
  = let scores = [ minimax (d - 1) move | move <- legalMoves g ]
        optimalScore =
          if shouldMaximize g then maximum scores else minimum scores
    in  optimalScore
```

src/Minimax/Par/Move.hs:

```haskell
-- | Move generation by parallel minimax algorithm

module Minimax.Par.Move
  ( bestMove
  ) where

import           Chess                                  ( Game(..)
                                                        , Player(..)
                                                        )
import           Minimax.Common                         ( Depth )
import           Rules                                  ( legalMoves )
import           Score                                  ( Score
                                                        , gameScore
                                                        )

import           Control.Parallel.Strategies            ( evalTuple2
                                                        , parList
                                                        , rseq
                                                        , using
                                                        )

bestMove :: Depth -> Depth -> Game -> Game
bestMove parDepth depth g =
  let evalStrat = if parDepth > 0 then parList (evalTuple2 rseq rseq) else rseq
      movesWithScores =
        map (\move -> (move, minimax (parDepth - 1) (depth - 1) move))
            (legalMoves g)
          `using` evalStrat
```

```haskell
        comparator = if shouldMaximize g
          then \x@(_, xscore) y@(_, yscore) -> if xscore >= yscore then x else y
          else \x@(_, xscore) y@(_, yscore) -> if xscore <= yscore then x else y
        optimalMove = fst $ foldr1 comparator movesWithScores
  in  optimalMove

shouldMaximize :: Game -> Bool
shouldMaximize Game { gamePlayer = White } = True
shouldMaximize Game { gamePlayer = Black } = False

minimax :: Depth -> Depth -> Game -> Score
minimax parDepth depth g
  | depth > 0
  = let
        evalStrat = if parDepth > 0 then parList rseq else rseq
        scores =
          map (minimax (parDepth - 1) (depth - 1)) (legalMoves g)
            `using` evalStrat
        optimalScore =
          if shouldMaximize g then maximum scores else minimum scores
    in
        optimalScore
  | otherwise
  = gameScore g
```

src/Minimax/SeqAB/Move.hs:

```haskell
-- | Move generation by sequential minimax algorithm with alpha-beta pruning

module Minimax.SeqAB.Move
  ( bestMove
  ) where

import           Chess                          ( Game(..)
                                                , Player(..)
                                                )
import           Minimax.Common                 ( Depth )
import           Rules                          ( legalMoves )
import           Score                          ( Score
                                                , gameScore
                                                )

bestMove :: Depth -> Game -> Game
bestMove d g =
  let movesWithScores =
        [ (move, minimax (d - 1) (-10000) 10000 move) | move <- legalMoves g ]
      comparator = if shouldMaximize g
        then \x@(_, xscore) y@(_, yscore) -> if xscore >= yscore then x else y
        else \x@(_, xscore) y@(_, yscore) -> if xscore <= yscore then x else y
      optimalMove = fst $ foldr1 comparator movesWithScores
  in  optimalMove

shouldMaximize :: Game -> Bool
shouldMaximize Game { gamePlayer = White } = True
shouldMaximize Game { gamePlayer = Black } = False

minimax :: Depth -> Score -> Score -> Game -> Score
minimax d alpha beta g
  | d <= 0
  = gameScore g
  | shouldMaximize g
  = let optimalScore _ prevBest [] = prevBest
        optimalScore alpha' prevBest (move : moves) =
          let currBest = max prevBest (minimax (d - 1) alpha' beta move)
              alpha''  = max alpha' currBest
          in  if beta <= alpha''
                then currBest
                else optimalScore alpha'' currBest moves
    in  optimalScore alpha (-9999) (legalMoves g)
  | otherwise
  = let optimalScore _ prevBest [] = prevBest
        optimalScore beta' prevBest (move : moves) =
          let currBest = min prevBest (minimax (d - 1) alpha beta' move)
              beta''   = min beta' currBest
          in  if beta'' <= alpha
                then currBest
                else optimalScore beta'' currBest moves
    in  optimalScore beta 9999 (legalMoves g)
```

src/Minimax/ParAB/Move.hs:

```haskell
-- | Move generation by parallel minimax algorithm with alpha-beta pruning

module Minimax.ParAB.Move
  ( bestMove
  ) where

import           Chess                          ( Game(..)
                                                , Player(..)
```

```haskell
                                                            )
import              Control.Parallel.Strategies             ( evalTuple2
                                                            , parList
                                                            , rseq
                                                            , using
                                                            )
import              Minimax.Common                          ( Depth )
import              Rules                                   ( legalMoves )
import              Score                                   ( Score
                                                            , gameScore
                                                            )

bestMove :: Depth -> Depth -> Game -> Game
bestMove parDepth d g
  | parDepth <= 1
  = let
      movesWithScores =
        [ (move, minimaxAB (d - 1) (-10000) 10000 move) | move <- legalMoves g ]
        `using` parList (evalTuple2 rseq rseq)
      comparator = if shouldMaximize g
        then \x@(_, xscore) y@(_, yscore) -> if xscore >= yscore then x else y
        else \x@(_, xscore) y@(_, yscore) -> if xscore <= yscore then x else y
      optimalMove = fst $ foldr1 comparator movesWithScores
    in
      optimalMove
  | otherwise
  = let movesWithScores =
          [ (move, minimax (parDepth - 1) (d - 1) move) | move <- legalMoves g ]
          `using` parList (evalTuple2 rseq rseq)
        comparator = if shouldMaximize g
          then \x@(_, xscore) y@(_, yscore) -> if xscore >= yscore then x else y
          else \x@(_, xscore) y@(_, yscore) -> if xscore <= yscore then x else y
        optimalMove = fst $ foldr1 comparator movesWithScores
    in  optimalMove

shouldMaximize :: Game -> Bool
shouldMaximize Game { gamePlayer = White } = True
shouldMaximize Game { gamePlayer = Black } = False

minimax :: Depth -> Depth -> Game -> Score
minimax parDepth d g
  | d <= 0
  = gameScore g
  | parDepth <= 1
  = let scores =
          [ minimaxAB (d - 1) (-10000) 10000 move | move <- legalMoves g ]
          `using` parList rseq
        optimalScore =
          if shouldMaximize g then maximum scores else minimum scores
    in  optimalScore
  | otherwise
  = let scores =
          [ minimax (parDepth - 1) (d - 1) move | move <- legalMoves g ]
          `using` parList rseq
        optimalScore =
          if shouldMaximize g then maximum scores else minimum scores
    in  optimalScore

minimaxAB :: Depth -> Score -> Score -> Game -> Score
minimaxAB d alpha beta g
  | d <= 0
  = gameScore g
  | shouldMaximize g
  = let optimalScore _ prevBest [] = prevBest
        optimalScore alpha' prevBest (move : moves) =
          let currBest = max prevBest (minimaxAB (d - 1) alpha' beta move)
              alpha''  = max alpha' currBest
          in  if beta <= alpha''
                then currBest
                else optimalScore alpha'' currBest moves
    in  optimalScore alpha (-9999) (legalMoves g)
  | otherwise
  = let optimalScore _ prevBest [] = prevBest
        optimalScore beta' prevBest (move : moves) =
          let currBest = min prevBest (minimaxAB (d - 1) alpha beta' move)
              beta''   = min beta' currBest
          in  if beta'' <= alpha
                then currBest
                else optimalScore beta'' currBest moves
    in  optimalScore beta 9999 (legalMoves g)
```