Min-Max Kalah

Haruki Gonai and David Cendejas
{hg2541, dc3448}@columbia.edu

December 22, 2022

1 Abstract

Kalah is a two-player game involving seeds and a board with pits. Kalah is a solved game, meaning an optimal move can be found by traversing a tree of each player's moves using algorithms such as min-max and alpha-beta pruning. However, sequential implementations of these algorithms have large execution times due to Kalah's large search space. For this reason, we attempted various techniques at parallelizing min-max and alpha-beta pruning to speed up the execution times.

2 Introduction

Kalah is played with seeds and a board with pits. The board consists of 12 small pits, called houses, and two big pits, called stores, as shown in Figure 1. Initially, each house contains four seeds.

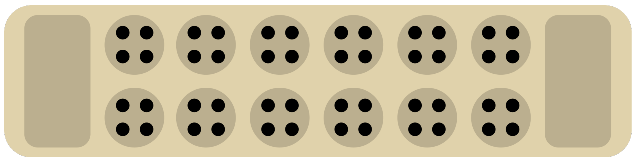


Figure 1: Initial board arrangement in Kalah

Player 0 owns the bottom row of six houses and the store to the right of them. Player 1 owns the top row of six houses and the store to the left of them. Each turn, a player picks one of their own houses that contains at least one seed. The player then removes the seeds from the house and deposits a seed into each pit (except the opponent's store), going counter-clockwise.

For example, suppose it is player 0's turn and they choose their third house from the right. Player 0 would remove the four seeds from this house and deposit a seed into each pit, going counter-clockwise, as depicted in Figure 2.

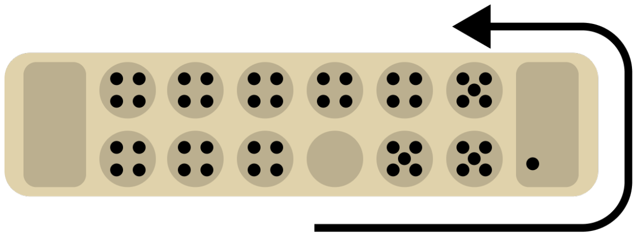


Figure 2: Kalah board after a move

Each player takes turns making a move, and the objective of the game is to have more total seeds in your houses and store than the opponent when the game ends.

Some additional rules are as follows:

- If the last pit into which the player deposits a seed during a move is their own store, they get another move.
- If the last pit into which the player deposits a seed during a move is an empty house that belongs to them, and the opponent's house across from it is not empty, then the player moves all the seeds in these two houses to the player's store.
- The game ends when one player has no seeds in any of their houses, at which point the winner is whoever has more total seeds in their houses and store [2].

3 Design and Implementation

3.1 Kalah State

To implement min-max and alpha-beta pruning for Kalah, a data structure storing the current state of the Kalah game was necessary. Hence, we created `State`:

```haskell
import qualified Data.Vector as V
...
data State = State (V.Vector Int) Bool
```

`V.Vector Int` contains 14 contiguous `Int`s and keeps track of the seeds in each of the Kalah board's pits. The `V.Vector Int` is used as a circular buffer, and each index of it represents a pit of the board, going counter-clockwise, i.e.:

- Indices 0 to 5 represent player 0's houses.

- Index 6 represents player 0's store.
- Indices 7 to 12 represent player 1's houses.
- Index 13 represents player 1's store.

Using a `Vector`, as opposed to other data structures such as a list, was crucial to access these indices in constant time.

`Bool`, which is also part of `State`, keeps track of whose turn it is. Its value is true if and only if it is player 0's turn.

3.2 Kalah Gameplay

After creating `State`, we wrote several functions to implement the rules of Kalah. For example, the `makeMove` function takes in the current `State`, `s`, and an `Int`, `a`, representing the index of the house picked for the move. The function returns the successor `State` wrapped in the `Maybe` monad if the move is valid.

```haskell
import qualified Data.Vector as V
...
-- Given a State and a house to pick, outputs the resulting State
makeMove :: State -> Int -> Maybe State
makeMove s@(State board isP0Turn) a = do
  if isValidMove s a
    then do
      let boardFunc i
            | i == a = 0
            | otherwise = board V.! i
      let n = board V.! a
      return $ deposit (a + 1) (-1) boardFunc n isP0Turn
  else do
    Nothing

-- Given a state and a move, is the move valid?
isValidMove :: State -> Int -> Bool
isValidMove (State board isP0Turn) a
  | isP0Turn = 0 <= a && a <= 5 && houseNotEmpty
  | otherwise = 7 <= a && a <= 12 && houseNotEmpty
    where houseNotEmpty = houseIsNotEmpty board a

-- Is the house empty?
houseIsNotEmpty :: (V.Vector Int) -> Int -> Bool
houseIsNotEmpty board i = board V.! i /= 0

-- Deposit seeds around the Kalah board
deposit :: Int -> Int -> (Int -> Int) -> Int -> Bool -> State
deposit i prev f n isP0Turn
  | n == 0 = State (V.generate 14 $ steal f prev isP0Turn) nxtIsP0Turn
  | (i == 6 && not isP0Turn) || (i == 13 && isP0Turn) =
      deposit iNxt prev f n isP0Turn
  | otherwise = deposit iNxt i fNew (n - 1) isP0Turn
    where fNew i'
            | i == i' = (f i') + 1
            | otherwise = f i'
          iNxt = mod (i + 1) 14
          nxtIsP0Turn = (prev == 6 && isP0Turn) ||
                        (prev /= 13 && not isP0Turn)

-- Steal seeds from opponent if possible
steal :: (Int -> Int) -> Int -> Bool -> (Int -> Int)
steal f lastPit isP0Turn
  | lastPitOurs && canSteal = fNew
  | otherwise = f
    where lastPitOurs = (isP0Turn && (0 <= lastPit && lastPit <= 5)) ||
                        (not isP0Turn && (7 <= lastPit && lastPit <= 12))
          seedsInLast = f lastPit
          pitAcross = 12 - lastPit
          seedsAcross = f pitAcross
          canSteal = seedsInLast == 1 && seedsAcross /= 0
          fNew i
            | i == lastPit || i == pitAcross = 0
            | (isP0Turn && i == 6) || (not isP0Turn && i == 13) =
                (f i) + seedsInLast + seedsAcross
            | otherwise = f i
```

makeMove first calls the isValidMove function to check whether a is a valid move for the current player (i.e. if the player owns house a, and house a is not empty). If this is the case, makeMove calls the deposit function, which simulates depositing a seed into each pit, going counter-clockwise. Once there are no more seeds to deposit, deposit calls steal. steal checks if the last seed landed in one of the player's empty houses and if the opponent's house across from it is non-empty, in which case the player moves the seeds in these two houses to their store.

We also wrote isTerm to determine whether the game has ended (i.e. either player has no seeds in any of their houses).

```haskell
-- Is the game over?
isTerm :: State -> Bool
isTerm (State board _) = (V.sum $ V.slice 0 6 board) == 0 ||
                         (V.sum $ V.slice 7 6 board) == 0
```

Additionally, we wrote `util`, which is player 0's score minus player 1's score, where each player's score is the total number of seeds in their houses and store.

```haskell
-- Player 0's score - Player 1's score
util :: State -> Int
util (State board _) = (V.sum $ V.slice 0 7 board) -
                       (V.sum $ V.slice 7 7 board)
```

When the game ends, a positive return value from `util` means that player 0 has won. A negative return value means that player 1 has won, etc. Since the function determines the winner at the end of the game, we also decided to use it as the heuristic to evaluate non-terminal states when the search depth limit is reached in min-max and alpha-beta pruning.

3.3 Kalah AI

3.3.1 Sequential Versions

After implementing the rules of Kalah, we wrote a basic Kalah AI by implementing sequential min-max (`SeqMM`) and sequential alpha-beta pruning (`SeqAB`). To ensure that the algorithms terminate in a reasonable amount of time, we imposed a depth limit `k` on the search.

To implement `SeqMM`, we wrote the following `runMM` function:

```haskell
-- Sequential MM
runMM :: Int -> State -> (Int, Int)
runMM k s
  | k == 0 || isTerm s = (util s, -1)
  | getIsP0Turn s = maximumBy valMoveOrdering childrenMMRes
  | otherwise = minimumBy valMoveOrdering childrenMMRes
    where succs = genSuccs s
          childrenMMRes = [(fst $ runMM (k - 1) s', a) | (a, s') <- succs]

genSuccs :: State -> [(Int, State)]
genSuccs s = [(a, unwrapState $ makeMove s a) | a <- getValidMoves s]
  where unwrapState (Just s') = s'
        unwrapState Nothing = error "Something went wrong during MinMax"

valMoveOrdering :: (Int, Int) -> (Int, Int) -> Ordering
valMoveOrdering (v0,_) (v1,_) = compare v0 v1
```

This implementation of min-max differs from that of most min-max algorithms, in which `max` calls `min` and then `min` calls `max`. The reason for this is that in Kalah, a player is granted another move if their last seed lands in their store. Hence, we abstracted `min` and `max` via `runMM`. `runMM` will find whose turn it is using the `isP0Turn` field of `State`, and use it to determine whether to maximize or minimize the utility of the children states accordingly.

The algorithm for `SeqAB` is similar, but slightly more involved since branches must be pruned:

```haskell
-- Sequential AB
runAB :: Int -> State -> (Int, Int) -> (Int, Int)
runAB k s (a, b)
  | k == 0 || isTerm s = (util s, -1)
  | getIsP0Turn s = maxAB (k - 1) (a, b) (negInfty, -1) succs
  | otherwise = minAB (k - 1) (a, b) (posInfty, -1) succs
    where succs = genSuccs s

maxAB :: Int -> (Int, Int) -> (Int, Int) -> [(Int, State)] -> (Int, Int)
maxAB _ _ (v, move) [] = (v, move)
maxAB k (a, b) (v, move) ((sMove, s):succs)
  | v' > b = (v', move')
  | otherwise = maxAB k (a', b) (v', move') succs
    where (v2, _) = runAB k s (a, b)
          (v', move', a')
            | v2 > v = (v2, sMove, max a v)
            | otherwise = (v, move, a)

minAB :: Int -> (Int, Int) -> (Int, Int) -> [(Int, State)] -> (Int, Int)
minAB _ _ (v, move) [] = (v, move)
minAB k (a, b) (v, move) ((sMove, s):succs)
  | v' <= a = (v', move')
  | otherwise = minAB k (a, b') (v', move') succs
    where (v2, _) = runAB k s (a, b)
          (v', move', b')
            | v2 < v = (v2, sMove, min b v)
            | otherwise = (v, move, b)
```

3.3.2 Parallel Versions

To speed up our sequential AIs, we attempted to parallelize min-max and alpha-beta pruning in various ways.

Our first parallel implementation was a parallelized version of min-max (`ParThenMM`).

```haskell
-- Run par MM for parDepth layers, then seq MM for deeper layers
```

```
runParThenMM :: Int -> Int -> State -> (Int, Int)
runParThenMM parDepth k s
  | k == 0 || isTerm s = (util s, -1)
  | parDepth == 0 = runMM k s
  | getIsP0Turn s = maximumBy valMoveOrdering childrenMMRes
  | otherwise = minimumBy valMoveOrdering childrenMMRes
    where succs = genSuccs s
          runMMOnSucc (a, s') =
            (fst $ runParThenMM (parDepth - 1) (k - 1) s', a)
          childrenMMRes = parMap rdeepseq runMMOnSucc succs
```

In ParThenMM, we perform the search in parallel for parDepth layers, and then use SeqMM for the deeper layers. We decided to limit the depth to which we use parallelism to avoid too many sparks from being created, which would slow down the execution time.

After implementing ParThenMM, we recognized that instead of performing SeqMM for these deeper layers, we could use SeqAB. We created this implementation, called ParThenAB, by just changing the use of runMM above with runAB:

```
-- Run par MM for parDepth layers, then seq AB for deeper layers
runParThenAB :: Int -> Int -> State -> (Int, Int)
runParThenAB parDepth k s
  | k == 0 || isTerm s = (util s, -1)
  | parDepth == 0 = runAB k s (negInfty, posInfty)
  | getIsP0Turn s = maximumBy valMoveOrdering childrenMMRes
  | otherwise = minimumBy valMoveOrdering childrenMMRes
    where succs = genSuccs s
          runMMOnSucc (a, s') =
            (fst $ runParThenAB (parDepth - 1) (k - 1) s', a)
          childrenMMRes = parMap rdeepseq runMMOnSucc succs
```

To further optimize our implementation, we turned toward the "Younger Brothers Wait Concept" algorithm [3]. The goal of this algorithm is to fix the lack of pruning in the first parDepth layers of ParThenAB. It accomplishes this by running SeqAB on the first child, updating the α or β value, pruning if possible, and then running the "Young Brothers Wait Concept" algorithm recursively in parallel on the other children. After parDepth layers, SeqAB is used to evaluate the deeper layers.

This implementation, which we refer to as YoungBros, was written as follows:

```
-- Run seq AB on first child, prune, then recurse on rest of children
-- Once parDepth is reached, only seq AB is used
runYoungBros :: Int -> Int -> State -> (Int, Int) -> (Int, Int)
runYoungBros parDepth k s (a, b)
```

```
  | k == 0 || isTerm s = (util s, -1)
  | parDepth == 0 = runAB k s (a, b)
  | getIsP0Turn s = runYoungBrosMax (parDepth - 1) (k - 1) (a, b) succs
  | otherwise = runYoungBrosMin (parDepth - 1) (k - 1) (a, b) succs
    where succs = genSuccs s

runYoungBrosMax :: Int -> Int -> (Int, Int) -> [(Int, State)] -> (Int, Int)
runYoungBrosMax _ _ _ [] = error "Something went wrong during MinMax"
runYoungBrosMax parDepth k (a, b) ((house, s):succs)
  | v > b = (v, move)
  | otherwise = maximumBy valMoveOrdering ((v, house):childrenMMRes)
    where (v, move) = runAB k s (a, b)
          a' = max a v
          runYoungBrosOnSucc (house', s') =
            (fst $ runYoungBros parDepth k s' (a', b), house')
          childrenMMRes = parMap rdeepseq runYoungBrosOnSucc succs

runYoungBrosMin :: Int -> Int -> (Int, Int) -> [(Int, State)] -> (Int, Int)
runYoungBrosMin _ _ _ [] = error "Something went wrong during MinMax"
runYoungBrosMin parDepth k (a, b) ((house, s):succs)
  | v <= a = (v, move)
  | otherwise = minimumBy valMoveOrdering ((v, house):childrenMMRes)
    where (v, move) = runAB k s (a, b)
          b' = min b v
          runYoungBrosOnSucc (house', s') =
            (fst $ runYoungBros parDepth k s' (a, b'), house')
          childrenMMRes = parMap rdeepseq runYoungBrosOnSucc succs
```

4 Results

Before measuring the speed-up of the parallel implementations compared to the sequential ones, we decided to find the best configurations for the parallel implementations, in terms of number of cores and parallelism depth limit (`parDepth`).

To standardize measurements across the different implementations, we fixed the search depth limit (`k`) to 9 for all of our tests. We chose this number because for all the implementations, the search terminated in a reasonable amount of time for this search depth (i.e. neither too fast nor too slow).

We performed the benchmark tests using the Criterion package for Haskell. For each parallel AI, we varied the number of cores between 2 and 8 and `parDepth` between 1 and 9. For each of these configurations, we ran the AI on the initial Kalah board using the default configuration for Criterion and benchmarked the mean execution time.

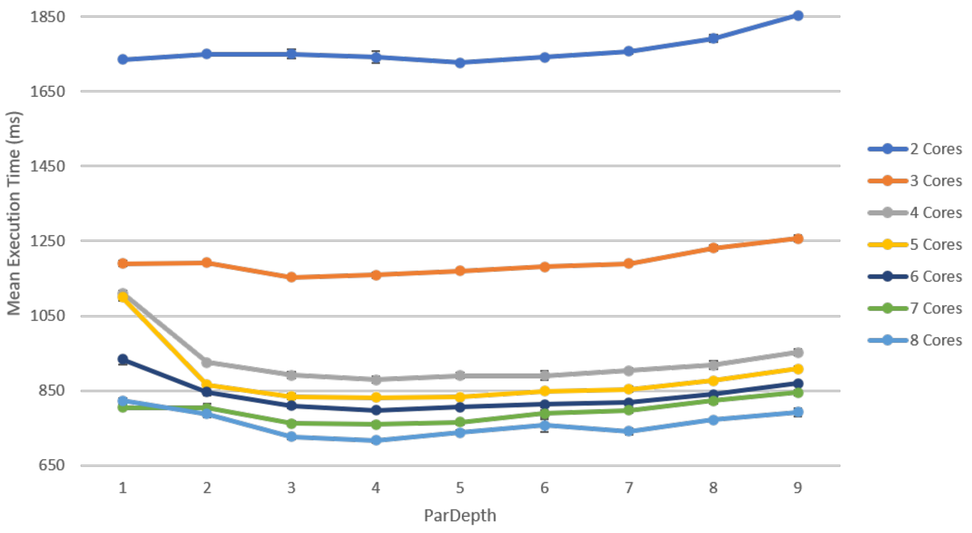


Figure 3: Mean Execution Time Vs. `parDepth` for `ParThenMM`

Figure 3 shows a graph of mean execution time vs. `parDepth` on various numbers of cores for the `ParThenMM` AI. We observed that as the number of cores increased, the execution time for a given `parDepth` improved. For 4 to 9 cores, we also observed that from a `parDepth` of 1 to 3, the execution time improved. However, as `parDepth` increased beyond that, the execution time gradually worsened, likely due to too many sparks being created. The fastest mean execution time was on 8 cores around a `parDepth` of 3.

Figure 4 shows mean execution time vs. `parDepth` for various numbers of cores for `ParThenAB`. We observed that as the `parDepth` increased, the performance worsened, likely due to fewer opportunities to prune. Generally, the performance improved as the number of cores in-

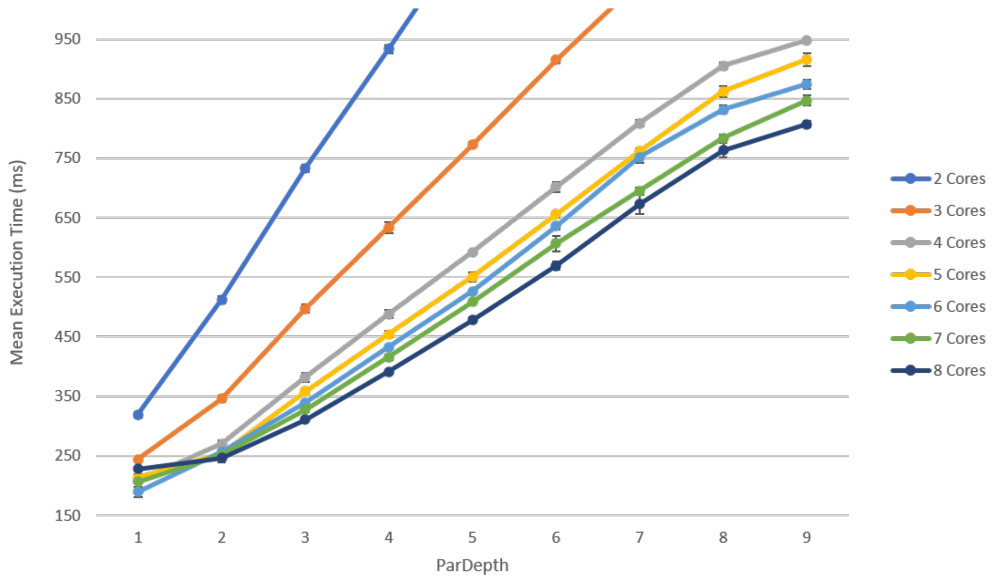


Figure 4: Mean Execution Time Vs. `parDepth` for `ParThenAB`

creased. However, at a `parDepth` of 1, the execution time on 6 cores was faster than that on 7 or 8 cores. A potential cause for this issue may have been due to the increased overhead required to coordinate parallelism as the number of cores increased past 6. Overall, the ideal configuration for `ParThenAB` was on 6 cores with a `parDepth` of 1.

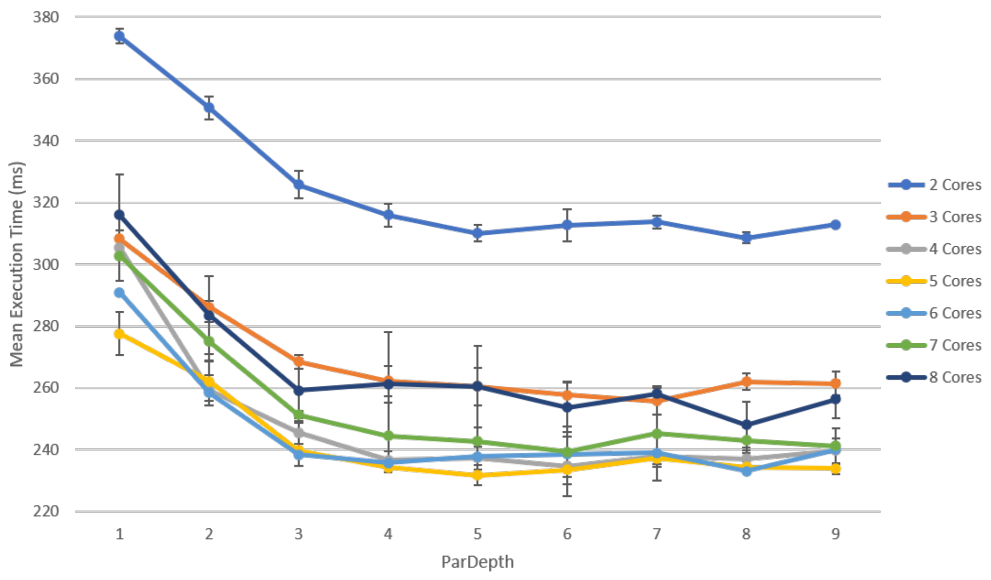


Figure 5: Mean Execution Time Vs. `parDepth` for `YoungBros`

Figure 5 shows mean execution time vs. parDepth for different numbers of cores for YoungBros. In general, the execution time decreased as the number of cores were varied from 2 to 6, but increased as the number of cores were changed from 6 to 8. We again believe that this was due to increased overhead in coordinating parallelism when using more than 6 cores. We also observed that from a parDepth of 1 to around 3, the execution time decreased. Past a parDepth of 3, the execution time stayed relatively constant. Overall, we determined that the ideal configuration for YoungBros was on 6 cores with a parDepth of 3.

After finding the ideal configurations for our parallel implementations, we compared their execution times with those of the sequential implementations.

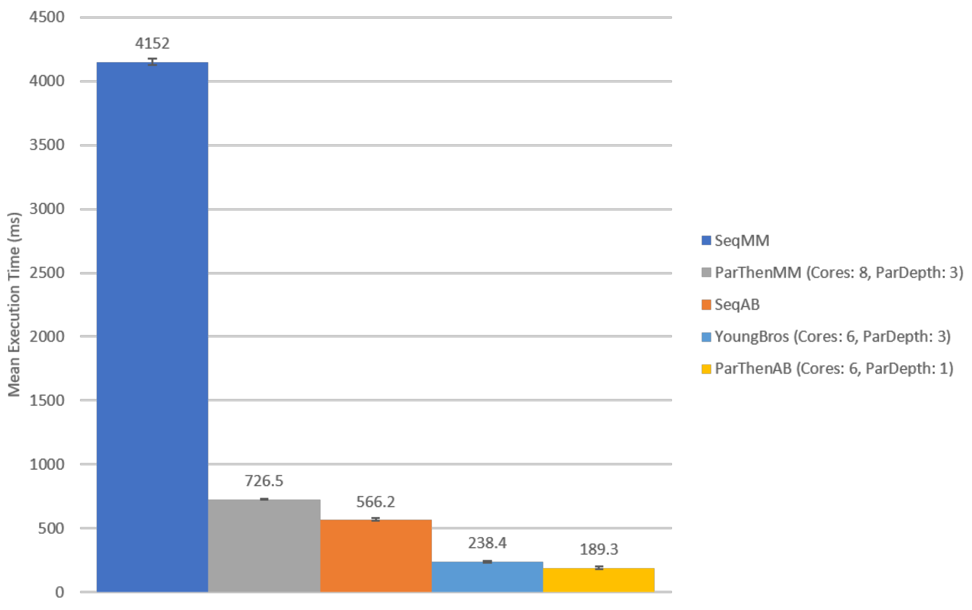


Figure 6: Mean Execution Time by Implementation

Figure 6 is a chart of the mean execution time of all implementations on the initial Kalah board. We found that SeqMM had the slowest execution time. This was expected because SeqMM runs the min-max algorithm with no optimization. ParThenMM, on 8 cores and a parDepth of 3, had the next slowest execution time, followed by SeqAB. We were surprised that SeqAB had a faster execution time than ParThenMM, which reflected the importance of pruning nodes for better performance.

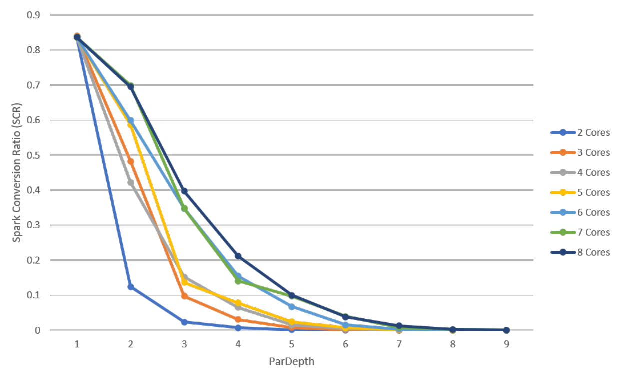
On the other hand, YoungBros and ParThenAB had the fastest execution times. This matched our expectations since these implementations were the most optimized out of the 5. However, we did not expect that ParThenAB would be faster than YoungBros, since the purpose of YoungBros was to introduce more opportunities for pruning to improve the execution time beyond that of ParThenAB.

To summarize the total speed up achieved by introducing parallelism, ParThenAB's execution time was roughly 21.933 times faster than that of SeqMM and 2.991 times faster than that of

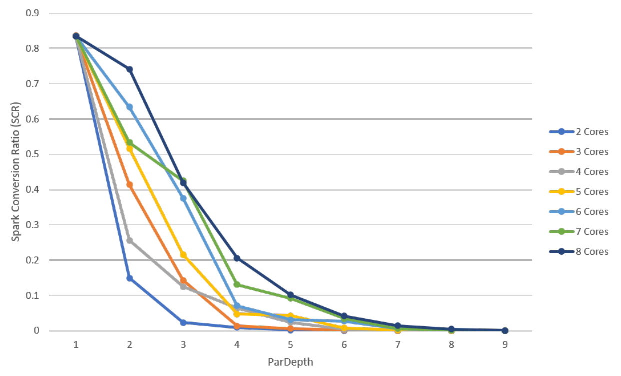
SeqAB.

4.1 Spark Conversion Ratios

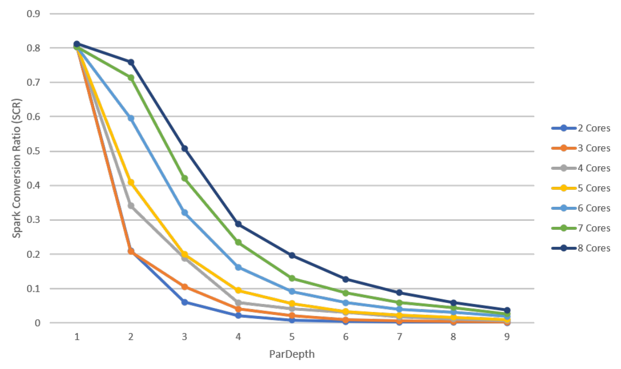
To further evaluate our parallel implementations, we also investigated their spark conversion ratios (i.e. the number of sparks converted divided by the number of sparks created). For each parallel implementation, we once again varied the number of cores between 2 and 8 and parDepth between 1 and 9. For each of these configurations, we ran the implementation on the initial Kalah board using the default configuration for Criterion and calculated the average spark conversion ratio (SCR).



(a) SCR vs. parDepth for ParThenMM



(b) SCR vs. parDepth for ParThenAB



(c) SCR vs. parDepth for YoungBros

Figure 7: SCR for ParThenMM, ParThenAB, YoungBros

For each parallel implementation, we found that as the number of cores increased, the SCR increased for a fixed parDepth. Additionally, as the parDepth increased, the SCR decreased dramatically, likely due to too many sparks being created and most sparks fizzling or being garbage collected.

5 Conclusion and Next Steps

We observed that the performance of the Kalah AIs improved greatly when using parallelism. In particular, the fastest AI, `ParThenAB`, exhibited a mean execution time that was several times greater than those of the sequential `SeqMM` and `SeqAB` AIs.

We also observed that increasing the number of cores improves the ratio of sparks converted to total sparks created. However, for `ParThenAB` and `YoungBros`, the execution time worsens as the number of cores increases past 6, most likely due to too much overhead required to coordinate parallelism for more cores.

In the future, we intend to investigate the Kalah AIs further and improve them in several ways. For example, we could seek ways to improve the AI by modifying the underlying Kalah game implementation. For instance, we could use `Repa` instead of `Vector` and use its `computeP` function to compute the next state in parallel inside of the `makeMove` function. Additionally, we created parallel variants of `isTerm` and `util` that compute the two `sum` calls in parallel, but they have yet to be tested and benchmarked. We could also repeat the analysis on different sized Kalah boards and different initial seed configurations.

6 Code Listing

6.1 app/Main.hs

```haskell
module Main (main) where

import Text.Read
import MinMax
import Play
import Benchmark
import System.Environment
import System.Exit

main :: IO ()
main = do
  args <- getArgs
  case args of
    ["pve", mode, params] -> do
      minMaxConfig <- parseModeAndParams mode params
      pVsAI minMaxConfig
    ["eve", mode1, params1, mode2, params2] -> do
      minMaxConfig1 <- parseModeAndParams mode1 params1
      minMaxConfig2 <- parseModeAndParams mode2 params2
      aIVsAI minMaxConfig1 minMaxConfig2
      exitFailure
    ["onemove", mode, params] -> do
      minMaxConfig <- parseModeAndParams mode params
      runOneMove minMaxConfig
    ("benchmark":benchmarkName) -> do
      runBench benchmarkName
    _ -> do
      pn <- getProgName
      let usage = "Usage: " ++ pn ++ " pve AI_1_mode AI_1_params\n\
        \Usage: " ++ pn ++ " eve AI_1_mode AI_1_params \
        \AI_2_mode AI_2_params\n\
        \Usage: " ++ pn ++ " onemove AI_1_mode AI_1_params"
      die $ usage

parseModeAndParams :: String -> String -> IO MinMaxConfig
parseModeAndParams mode' params' = do
  case mode' of
    "seq_mm" -> do
```

```haskell
      depth <- parseOne params'
      return $ MinMaxConfig SeqMM depth (-1)
    "seq_ab" -> do
      depth <- parseOne params'
      return $ MinMaxConfig SeqAB depth (-1)
    "par_then_mm" -> do
      (depth, parDepth) <- parseTwo params'
      return $ MinMaxConfig ParThenMM depth parDepth
    "par_then_ab" -> do
      (depth, parDepth) <- parseTwo params'
      return $ MinMaxConfig ParThenAB depth parDepth
    "young_bros" -> do
      (depth, parDepth) <- parseTwo params'
      return $ MinMaxConfig YoungBros depth parDepth
    _ -> do
      die $ "Error: AI Mode must be one of \
        \[seq_mm, seq_ab, par_then_mm, par_then_ab, young_bros]"

parseOne :: String -> IO Int
parseOne params' = do
  case readMaybe params' of
    Just arg1 -> do
      if arg1 < 0 then
        die $ "Error: search_depth_limit must be non-negative"
      else return arg1
    Nothing -> do
      die $ "Error: AI params must be specified as \"(search_depth_limit)\""

parseTwo :: String -> IO (Int, Int)
parseTwo params' = do
  case readMaybe params' of
    Just (arg1,arg2) -> do
      if arg1 < 0 || arg2 < 0 then
        die $ "Error: search_depth_limit and parallelism_depth_limit must \
              \be non-negative"
      else return (arg1,arg2)
    Nothing -> do
      die $ "Error: AI params must be specified as \"(search_depth_limit,\
        \parallelism_depth_limit)\""
```

6.2 src/Benchmark.hs

```haskell
module Benchmark
    ( runBench
    ) where

import Criterion.Main
import MinMax
import Kalah
import System.Environment

-- Runs tests on our Kalah AI implementations with search depth limit 9
runBench :: [String] -> IO ()
runBench benchmarkName = withArgs benchmarkName tests
  where genBench mode dep = [bench (show dep) $ doBench mode dep]
        doBench mode dep = nf doBench' $ MinMaxConfig mode dep (-1)
        genBenchPar mode dep range =
          [bench (show i) $ doBenchPar mode dep i | i <- range]
        doBenchPar mode dep parDep =
          nf doBench' $ MinMaxConfig mode dep parDep
        doBench' minMaxConfig = computeAIMove minMaxConfig initState
        tests =
          defaultMain [
            bgroup "seq_mm" $ genBench SeqMM 9
          , bgroup "seq_ab" $ genBench SeqAB 9
          , bgroup "par_then_mm" $ genBenchPar ParThenMM 9 [1..9]
          , bgroup "par_then_ab" $ genBenchPar ParThenAB 9 [1..9]
          , bgroup "young_bros" $ genBenchPar YoungBros 9 [1..9]
          ]
```

6.3 src/Kalah.hs

```
{-# LANGUAGE TypeOperators #-}
module Kalah
    ( State(..),
      initState,
      makeMove,
      isTerm,
      isTermPar,
      util,
      utilPar,
      getValidMoves,
      whoseTurn,
      getBoardString,
      whoWon,
      getScore,
      getIsP0Turn
    ) where

import qualified Data.Vector as V
import Control.DeepSeq
import Control.Parallel.Strategies
import Text.Format

data State = State (V.Vector Int) Bool
instance NFData State where
  rnf (State board isP0Turn) = board `deepseq` isP0Turn `seq` ()

-- Initial Kalah state
initState :: State
initState = State (V.generate 14 f) True
  where f i
          | i == 6 || i == 13 = 0
          | otherwise = 4

-- Given a State and a house to pick, outputs the resulting State
makeMove :: State -> Int -> Maybe State
makeMove s@(State board isP0Turn) a = do
  if isValidMove s a
    then do
      let boardFunc i
            | i == a = 0
```

```haskell
            | otherwise = board V.! i
      let n = board V.! a
      return $ deposit (a + 1) (-1) boardFunc n isP0Turn
  else do
    Nothing

-- Deposit seeds around the Kalah board
deposit :: Int -> Int -> (Int -> Int) -> Int -> Bool -> State
deposit i prev f n isP0Turn
  | n == 0 = State (V.generate 14 $ steal f prev isP0Turn) nxtIsP0Turn
  | (i == 6 && not isP0Turn) || (i == 13 && isP0Turn) =
      deposit iNxt prev f n isP0Turn
  | otherwise = deposit iNxt i fNew (n - 1) isP0Turn
    where fNew i'
            | i == i' = (f i') + 1
            | otherwise = f i'
          iNxt = mod (i + 1) 14
          nxtIsP0Turn = (prev == 6 && isP0Turn) ||
                        (prev /= 13 && not isP0Turn)

-- Steal seeds from opponent if possible
steal :: (Int -> Int) -> Int -> Bool -> (Int -> Int)
steal f lastPit isP0Turn
  | lastPitOurs && canSteal = fNew
  | otherwise = f
    where lastPitOurs = (isP0Turn && (0 <= lastPit && lastPit <= 5)) ||
                        (not isP0Turn && (7 <= lastPit && lastPit <= 12))
          seedsInLast = f lastPit
          pitAcross = 12 - lastPit
          seedsAcross = f pitAcross
          canSteal = seedsInLast == 1 && seedsAcross /= 0
          fNew i
            | i == lastPit || i == pitAcross = 0
            | (isP0Turn && i == 6) || (not isP0Turn && i == 13) =
                (f i) + seedsInLast + seedsAcross
            | otherwise = f i

-- Is the game over?
isTerm :: State -> Bool
isTerm (State board _) = (V.sum $ V.slice 0 6 board) == 0 ||
                         (V.sum $ V.slice 7 6 board) == 0
```

```haskell
-- Player 0's score - Player 1's score
util :: State -> Int
util (State board _) = (V.sum $ V.slice 0 7 board) -
                       (V.sum $ V.slice 7 7 board)

-- Given a state, what are the valid moves?
getValidMoves :: State -> [Int]
getValidMoves (State board isP0Turn)
  | isP0Turn = filter (houseNotEmpty) [0..5]
  | otherwise = filter (houseNotEmpty) [7..12]
    where houseNotEmpty = houseIsNotEmpty board

-- Given a state and a move, is the move valid?
isValidMove :: State -> Int -> Bool
isValidMove (State board isP0Turn) a
  | isP0Turn = 0 <= a && a <= 5 && houseNotEmpty
  | otherwise = 7 <= a && a <= 12 && houseNotEmpty
    where houseNotEmpty = houseIsNotEmpty board a

-- Is the house empty?
houseIsNotEmpty :: (V.Vector Int) -> Int -> Bool
houseIsNotEmpty board i = board V.! i /= 0

-- Which player's turn is it?
whoseTurn :: State -> Int
whoseTurn (State _ True) = 0
whoseTurn (State _ False) = 1

-- Function to expose isP0Turn inside State
getIsP0Turn :: State -> Bool
getIsP0Turn (State _ isP0Turn) = isP0Turn

-- Pretty prints the board
-- Trust me, it actually looks fine
getBoardString :: State -> String
getBoardString (State board _) = format template $ numsAsStrings
  where template = "        13    12    11    10     9     8\n\
          \ _____ _____ _____ _____ _____ _____ _____ _____\n\
          \|     |     |     |     |     |     |     |     |\n\
          \|     | {12}  | {11}  | {10}  | {9}  | {8}  | {7}  |     |\n\
          \| {13}  |_____|_____|_____|_____|_____|_____| {6}  |\n\
          \|     |     |     |     |     |     |     |     |\n\
          \|     | {0}  | {1}  | {2}  | {3}  | {4}  | {5}  |     |\n\
          \|_____|_____|_____|_____|_____|_____|_____|_____|\n\
          \         1     2     3     4     5     6"
        numsAsStrings = fmap (padNum . show) $ V.toList board
        padNum i
          | length i == 2 = i
          | otherwise = " " ++ i

-- Who won?
whoWon :: State -> String
whoWon state
  | utilVal > 0 = "Player 0 wins!"
  | utilVal < 0 = "Player 1 wins!"
  | otherwise = "Tie!"
    where utilVal = util state

-- Prints each player's score
getScore :: State -> String
getScore (State board _) =
  "Player 0 score: " ++ p0Score ++ "  -  Player 1 score: " ++ p1Score
  where p0Score = show $ V.sum $ V.slice 0 7 board
        p1Score = show $ V.sum $ V.slice 7 7 board

-- isTerm and util, but in parallel
isTermPar :: State -> Bool
isTermPar (State board _) = runEval $ do
  a <- rpar $ (V.sum $ V.slice 0 6 board) == 0
  b <- rpar $ (V.sum $ V.slice 7 6 board) == 0
  _ <- rseq $ a || b
  return $ a || b

utilPar :: State -> Int
utilPar (State board _) = runEval $ do
  a <- rpar $ V.sum $ V.slice 0 7 board
  b <- rpar $ V.sum $ V.slice 7 7 board
  _ <- rseq a
  _ <- rseq b
  return $ a - b
```

6.4 src/MinMax.hs

```haskell
module MinMax
    ( computeAIMove,
      MinMaxConfig(..),
      MinMaxMode(..)
    ) where

import Kalah
import Data.List(maximumBy, minimumBy)
import Control.Parallel.Strategies

data MinMaxConfig = MinMaxConfig { mmMode :: MinMaxMode
                                 , depthLimit :: Int
                                 , parLimit :: Int }
data MinMaxMode = SeqMM | SeqAB | ParThenMM | ParThenAB | YoungBros
  deriving (Enum)

posInfty :: Int
posInfty = 99999999

negInfty :: Int
negInfty = -posInfty

genSuccs :: State -> [(Int, State)]
genSuccs s = [(a, unwrapState $ makeMove s a) | a <- getValidMoves s]
  where unwrapState (Just s') = s'
        unwrapState Nothing = error "Something went wrong during MinMax"

-- Sequential MM
runMM :: Int -> State -> (Int, Int)
runMM k s
  | k == 0 || isTerm s = (util s, -1)
  | getIsP0Turn s = maximumBy valMoveOrdering childrenMMRes
  | otherwise = minimumBy valMoveOrdering childrenMMRes
    where succs = genSuccs s
          childrenMMRes = [(fst $ runMM (k - 1) s', a) | (a, s') <- succs]

valMoveOrdering :: (Int, Int) -> (Int, Int) -> Ordering
valMoveOrdering (v0,_) (v1,_) = compare v0 v1

-- Sequential AB
```

```haskell
runAB :: Int -> State -> (Int, Int) -> (Int, Int)
runAB k s (a, b)
  | k == 0 || isTerm s = (util s, -1)
  | getIsP0Turn s = maxAB (k - 1) (a, b) (negInfty, -1) succs
  | otherwise = minAB (k - 1) (a, b) (posInfty, -1) succs
    where succs = genSuccs s

maxAB :: Int -> (Int, Int) -> (Int, Int) -> [(Int, State)] -> (Int, Int)
maxAB _ _ (v, move) [] = (v, move)
maxAB k (a, b) (v, move) ((sMove, s):succs)
  | v' > b = (v', move')
  | otherwise = maxAB k (a', b) (v', move') succs
    where (v2, _) = runAB k s (a, b)
          (v', move', a')
            | v2 > v = (v2, sMove, max a v)
            | otherwise = (v, move, a)

minAB :: Int -> (Int, Int) -> (Int, Int) -> [(Int, State)] -> (Int, Int)
minAB _ _ (v, move) [] = (v, move)
minAB k (a, b) (v, move) ((sMove, s):succs)
  | v' <= a = (v', move')
  | otherwise = minAB k (a, b') (v', move') succs
    where (v2, _) = runAB k s (a, b)
          (v', move', b')
            | v2 < v = (v2, sMove, min b v)
            | otherwise = (v, move, b)

-- Run par MM for parDepth layers, then seq MM for deeper layers
runParThenMM :: Int -> Int -> State -> (Int, Int)
runParThenMM parDepth k s
  | k == 0 || isTerm s = (util s, -1)
  | parDepth == 0 = runMM k s
  | getIsP0Turn s = maximumBy valMoveOrdering childrenMMRes
  | otherwise = minimumBy valMoveOrdering childrenMMRes
    where succs = genSuccs s
          runMMOnSucc (a, s') =
            (fst $ runParThenMM (parDepth - 1) (k - 1) s', a)
          childrenMMRes = parMap rdeepseq runMMOnSucc succs

-- Run par MM for parDepth layers, then seq AB for deeper layers
runParThenAB :: Int -> Int -> State -> (Int, Int)
runParThenAB parDepth k s
```

```haskell
  | k == 0 || isTerm s = (util s, -1)
  | parDepth == 0 = runAB k s (negInfty, posInfty)
  | getIsP0Turn s = maximumBy valMoveOrdering childrenMMRes
  | otherwise = minimumBy valMoveOrdering childrenMMRes
    where succs = genSuccs s
          runMMOnSucc (a, s') =
            (fst $ runParThenAB (parDepth - 1) (k - 1) s', a)
          childrenMMRes = parMap rdeepseq runMMOnSucc succs

-- Run seq AB on first child, prune, then recurse on rest of children
-- Once parDepth is reached, only seq AB is used
runYoungBros :: Int -> Int -> State -> (Int, Int) -> (Int, Int)
runYoungBros parDepth k s (a, b)
  | k == 0 || isTerm s = (util s, -1)
  | parDepth == 0 = runAB k s (a, b)
  | getIsP0Turn s = runYoungBrosMax (parDepth - 1) (k - 1) (a, b) succs
  | otherwise = runYoungBrosMin (parDepth - 1) (k - 1) (a, b) succs
    where succs = genSuccs s

runYoungBrosMax :: Int -> Int -> (Int, Int) -> [(Int, State)] -> (Int, Int)
runYoungBrosMax _ _ _ [] = error "Something went wrong during MinMax"
runYoungBrosMax parDepth k (a, b) ((house, s):succs)
  | v > b = (v, move)
  | otherwise = maximumBy valMoveOrdering ((v, house):childrenMMRes)
    where (v, move) = runAB k s (a, b)
          a' = max a v
          runYoungBrosOnSucc (house', s') =
            (fst $ runYoungBros parDepth k s' (a', b), house')
          childrenMMRes = parMap rdeepseq runYoungBrosOnSucc succs

runYoungBrosMin :: Int -> Int -> (Int, Int) -> [(Int, State)] -> (Int, Int)
runYoungBrosMin _ _ _ [] = error "Something went wrong during MinMax"
runYoungBrosMin parDepth k (a, b) ((house, s):succs)
  | v <= a = (v, move)
  | otherwise = minimumBy valMoveOrdering ((v, house):childrenMMRes)
    where (v, move) = runAB k s (a, b)
          b' = min b v
          runYoungBrosOnSucc (house', s') =
            (fst $ runYoungBros parDepth k s' (a, b'), house')
          childrenMMRes = parMap rdeepseq runYoungBrosOnSucc succs

computeAIMove :: MinMaxConfig -> State -> Int
computeAIMove (MinMaxConfig mode' k parDepth) s
  | k < 0 = error "Invalid search depth limit."
  | otherwise =
    case mode' of
      SeqMM -> snd $ runMM k s
      SeqAB -> snd $ runAB k s (negInfty, posInfty)
      ParThenMM -> snd $ runParThenMM parDepth k s
      ParThenAB -> snd $ runParThenAB parDepth k s
      YoungBros -> snd $ runYoungBros parDepth k s (negInfty, posInfty)
```

6.5 src/Play.hs

```haskell
module Play
    ( pVsAI,
      aIVsAI,
      runOneMove
    ) where

import Kalah
import Text.Read
import MinMax

-- PvE mode
pVsAI :: MinMaxConfig -> IO ()
pVsAI minMaxConfig = do
  putStrLn "Initializing Board..."
  let s = initState
  putStrLn $ getBoardString s
  let computeAIMove' = computeAIMove minMaxConfig
  doPVsAI computeAIMove' s

-- PvE mode helper
doPVsAI :: (State -> Int) -> State -> IO ()
doPVsAI computeAI1Move s = do
  if isTerm s
    then do
      let winner = whoWon s
      putStrLn winner
      let score = getScore s
      putStrLn score
      return ()
  else do
    putStrLn $ "Player " ++ (show $ whoseTurn s) ++ "'s turn."
    if getIsP0Turn s
      then do
        s' <- makePlayerTurn s
        printAndDoPlay s'
    else do
      s' <- makeAITurn computeAI1Move s
      printAndDoPlay s'
  where printAndDoPlay s' = do
          putStrLn $ getBoardString s'
          putStrLn "\n"
          doPVsAI computeAI1Move s'

-- EvE mode
aIVsAI :: MinMaxConfig -> MinMaxConfig -> IO ()
aIVsAI minMaxConfig1 minMaxConfig2 = do
  putStrLn "Initializing Board..."
  let s = initState
  putStrLn $ getBoardString s
  let computeAIMove1 = computeAIMove minMaxConfig1
  let computeAIMove2 = computeAIMove minMaxConfig2
  doAIVsAI computeAIMove1 computeAIMove2 s

-- EvE mode helper
doAIVsAI :: (State -> Int) -> (State -> Int) -> State -> IO ()
doAIVsAI computeAIMove1 computeAIMove2 s = do
  if isTerm s
    then do
      let winner = whoWon s
      putStrLn winner
      let score = getScore s
      putStrLn score
  else do
    putStrLn $ "Player " ++ (show $ whoseTurn s) ++ "'s turn."
    if getIsP0Turn s
      then do
        s' <- makeAITurn computeAIMove1 s
        printAndDoPlay s'
    else do
      s' <- makeAITurn computeAIMove2 s
      printAndDoPlay s'
  where printAndDoPlay s' = do
          putStrLn $ getBoardString s' ++ "\n"
          doAIVsAI computeAIMove1 computeAIMove2 s'

makePlayerTurn :: State -> IO State
makePlayerTurn s = do
  let validMoves = map (1+) $ getValidMoves s
  putStrLn $ "Valid Moves are: " ++ (show validMoves)
  line <- getLine
  case readMaybe line of
    Nothing -> do
```

```
        putStrLn "Invalid move."
        makePlayerTurn s
    Just a -> do
      let maybeS' = makeMove s (a - 1)
      case maybeS' of
        Just s' -> do
          return s'
        Nothing -> do
          putStrLn "Invalid move."
          makePlayerTurn s

makeAITurn :: (State -> Int) -> State -> IO State
makeAITurn computeAIMove' s = do
  let a = computeAIMove' s
  let maybeS' = makeMove s a
  case maybeS' of
    Just s' -> do
      putStrLn $ "AI picked " ++ (show $ a + 1) ++ "."
      return s'
    Nothing -> do
      return $ error $ "AI tried to make invalid move."

-- Just run AI on initial state
runOneMove :: MinMaxConfig -> IO ()
runOneMove minMaxConfig = do
  putStrLn "Initializing Board..."
  let move = computeAIMove minMaxConfig initState
  putStrLn $ "Optimal move is " ++ show move
```

6.6 test/Spec.hs

```haskell
import Kalah
import MinMax
import qualified Data.Vector as V

testMakeMove1 :: Bool
testMakeMove1 = case maybeS of
  Nothing -> False
  Just (State _ turn) ->
    case turn of
      True -> False
      False -> True
  where maybeS = makeMove initState 0

testMakeMove2 :: Bool
testMakeMove2 = case maybeS of
  Nothing -> False
  Just (State _ turn) ->
    case turn of
      True -> True
      False -> False
  where maybeS = makeMove initState 2

testIsTerm1 :: Bool
testIsTerm1 = not res
  where res = isTerm initState

testIsTerm2 :: Bool
testIsTerm2 = res
  where res = isTerm s
        s = State (V.generate 14 f) True
        f i
          | i == 5 = 4
          | otherwise = 0

testUtil1 :: Bool
testUtil1 = res == 0
  where res = util initState

testUtil2 :: Bool
testUtil2 = res == -18
```

```
  where res = util s
        s = State (V.generate 14 f) True
        f i
          | i == 8 = 20
          | i == 2 = 2
          | otherwise = 0

computeAIMove1 :: Bool
computeAIMove1 = move == 5
  where move = computeAIMove minMaxConfig initState
        minMaxConfig = MinMaxConfig SeqMM 9 (-1)

computeAIMove2 :: Bool
computeAIMove2 = move == 5
  where move = computeAIMove minMaxConfig initState
        minMaxConfig = MinMaxConfig SeqAB 9 (-1)

computeAIMove3 :: Bool
computeAIMove3 = move == 5
  where move = computeAIMove minMaxConfig initState
        minMaxConfig = MinMaxConfig ParThenMM 9 2

computeAIMove4 :: Bool
computeAIMove4 = move == 5
  where move = computeAIMove minMaxConfig initState
        minMaxConfig = MinMaxConfig ParThenAB 9 2

computeAIMove5 :: Bool
computeAIMove5 = move == 5
  where move = computeAIMove minMaxConfig initState
        minMaxConfig = MinMaxConfig YoungBros 9 2

aiMoveTestState :: State
aiMoveTestState = State (V.generate 14 f) False
  where f i
          | i < 3 = 4
          | 3 <= i && i <= 6 = 3
          | 7 <= i && i <= 10 = 1
          | otherwise = 6

computeAIMove6 :: Bool
computeAIMove6 = move == 9
```

```
  where move = computeAIMove minMaxConfig aiMoveTestState
        minMaxConfig = MinMaxConfig SeqMM 9 (-1)

computeAIMove7 :: Bool
computeAIMove7 = move == 9
  where move = computeAIMove minMaxConfig aiMoveTestState
        minMaxConfig = MinMaxConfig SeqAB 9 (-1)

computeAIMove8 :: Bool
computeAIMove8 = move == 9
  where move = computeAIMove minMaxConfig aiMoveTestState
        minMaxConfig = MinMaxConfig ParThenMM 9 (-1)

computeAIMove9 :: Bool
computeAIMove9 = move == 9
  where move = computeAIMove minMaxConfig aiMoveTestState
        minMaxConfig = MinMaxConfig ParThenAB 9 (-1)

computeAIMove10 :: Bool
computeAIMove10 = move == 9
  where move = computeAIMove minMaxConfig aiMoveTestState
        minMaxConfig = MinMaxConfig YoungBros 9 (-1)

main :: IO ()
main = do
  let res = and [testMakeMove1, testMakeMove2,
                 testIsTerm1, testIsTerm2,
                 testUtil1, testUtil2,
                 computeAIMove1, computeAIMove2,
                 computeAIMove3, computeAIMove4,
                 computeAIMove5, computeAIMove6,
                 computeAIMove6, computeAIMove7,
                 computeAIMove8, computeAIMove9,
                 computeAIMove10]
  if res then
    putStrLn "Tests Passed!"
  else
    error "Some Tests Failed."
```

References

[1] Gonai, H. (2022, November 26). *kalah-min-max*. GitHub. Retrieved November 28, 2022, from https://github.com/harukigonai/kalah-min-max

[2] Wikimedia Foundation. (2022, November 6). *Kalah*. Wikipedia. Retrieved November 28, 2022, from https://en.wikipedia.org/wiki/Kalah

[3] *Young Brothers Wait Concept*. Young Brothers Wait Concept - Chessprogramming wiki. (n.d.). Retrieved December 21, 2022, from https://www.chessprogramming.org/Young_Brothers_Wait_Concept