Jose A. Ramos
jar2333
Parallel Functional Programming

Report: MCTS.hs

Introduction

For the final project, I implemented the Monte Carlo Tree Search[1] general game playing algorithm in Haskell, with support for parallelizing it using Haskell's *Control.Parallel.Strategies.* The library is provided as a module called *MCTS* in a stack package, and includes two executables: a demo (player vs agent) and a simulation (agent vs agent), with tunable command line parameters. For these executables, a sample usage of the library is provided: an implementation of a Connect 4 playing agent. As we will see, this choice of game severely limited the potential of big gains from the specific form of parallelism employed, which will be discussed.

Algorithm

Monte Carlo Tree Search (MCTS) is an algorithm for playing any two player game. It searches the game tree, where each node is a possible state of the game. The children of a node are those game states which can be reached by a move from the current player. A node with no children (a tree leaf) is a terminal state, where the outcome of the game is decided (who won and who lost). Unlike other tree-based algorithms which prune the search space like Minimax[2] variants, MCTS does not require any heuristics or custom evaluation function for a given game. Instead, it "samples" the outcome of a game from a given game state, by simulating an entire game starting at said state—usually by randomly sampling moves until completion. This implies that the algorithm can be applied to *any* two-player game, given that the rules of the game are known. More precisely, one has to define a game state struct/object, which has any relevant game state information, along with a few functions:

- One which takes a game state and returns all possible children.
- One which takes a terminal game state and returns the outcome of the game (or returns something indicating that the state is not terminal).
- One which takes a game state, simulates an entire game to completion, and returns the outcome of the game.

Implementation: Typeclass

The above description can be formalized in Haskell using typeclasses. The MCTS module can define a GameState typeclass (given a Player data type):

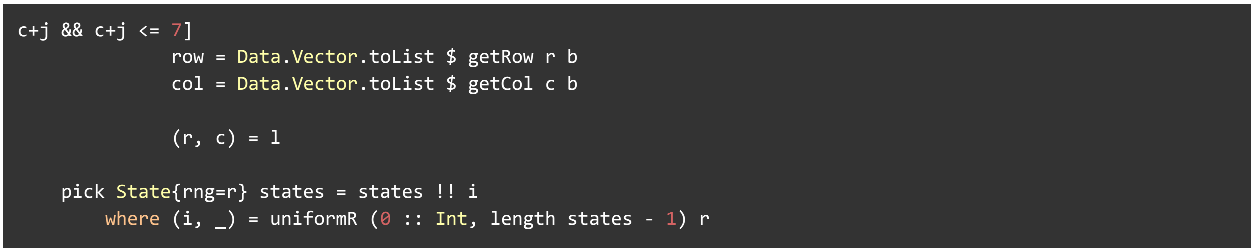
```haskell
data Player = One | Two | Tie deriving (Eq, Show)
```

[1] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
[2] https://en.wikipedia.org/wiki/Minimax

```
class GameState g where
    next :: g -> [g]
    eval :: g -> Maybe(Player)
    pick :: g -> [g] -> g

    sim :: g -> Player
    sim g = case next g of
                [] -> case eval g of
                         Just result  -> result
                         Nothing      -> Tie

                gs -> let picked = pick g gs
                      in case eval picked of
                         Just result  -> result
                         Nothing      -> sim picked
```

Any instance of this type class can be used in the MCTS algorithm, making the implementation as generic as possible. An implementation of a Connect Four state is given, which is plugged into the generic algorithm and works perfectly:

```
data (RandomGen r) => ConnectFourState r = State {  board :: Matrix Color,
                                                    currentPlayer :: Color,
                                                    lastMove :: (Int, Int),
                                                    rng :: r }

instance (RandomGen r) => GameState (ConnectFourState r) where
    next State{board=b, currentPlayer=p, rng=r} = states
        where
              states = zipWith (\childBoard move -> State childBoard nextPlayer move newRNG) boards indeces

              nextPlayer = other p
              newRNG = snd $ split r

              boards = map (\(i,j) -> setElem p (i,j) b) indeces

              indeces = zip rowIndeces colIndeces


              rowIndeces = map (place b) colIndeces
              colIndeces = filter (\j -> (getElem 1 j b) == Blank) [1..ncols b]

    eval State{board=b, currentPlayer=p, lastMove=l} = if isRun
        then Just(toPlayer $ other p)
        else Nothing
        where
              -- If a run is encountered, last move completed it, hence last player won
              isRun = maxRun sndDiag >= 4 || maxRun fstDiag >= 4 || maxRun row >= 4 || maxRun col >= 4

              sndDiag = [getElem (r+i) (c+j) b | (i, j) <- zip [-7..7] [7,6..(-7)], 1 <= r+i && r+i <= 6 && 1
<= c+j && c+j <= 7]
              fstDiag = [getElem (r+i) (c+j) b | (i, j) <- zip [-7..7] [-7..7], 1 <= r+i && r+i <= 6 && 1 <=
```

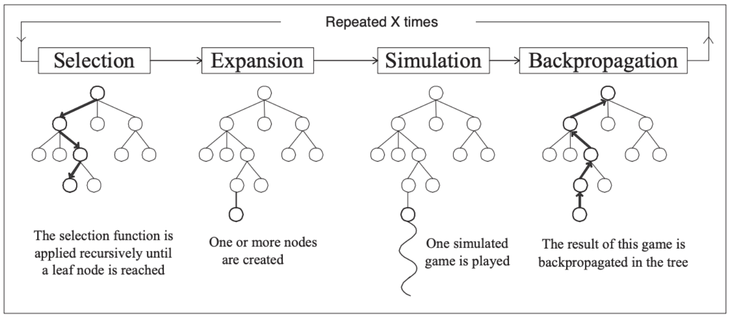


```
c+j && c+j <= 7]
              row = Data.Vector.toList $ getRow r b
              col = Data.Vector.toList $ getCol c b

              (r, c) = l

    pick State{rng=r} states = states !! i
        where (i, _) = uniformR (0 :: Int, length states - 1) r
```

Implementation: Monads

In the implementation of the algorithm as presented in the MCTS wikipedia page (or more authoritatively, this Swarthmore page[3]), the algorithm is characterized as a sequence of N transformations applied to the game tree. Each transformation can be described imperatively as 4 steps:

1. Selection: walk down the game tree and select a leaf node using UCB formula.
2. Expansion: expand the leaf node and add n of its children to the tree.
3. Simulation: simulate games starting from each of the n children.
4. Backpropagation: propagate the simulation results up the tree.


Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

After N iterations of the above process, the root node's child with the highest score is chosen as the next state to be reached. In my current implementation, the selection and backpropagation occur in the same function *walk*, which recursively walks down the tree to a leaf node, uses the State monad to store the simulation results obtained at said leaf as a state, then reconstructs or "updates" the current node's information using the state. The *step* function is simply an invocation of *evalState* on the results of *walk*. The full generic *mcts* function utilizes this *step* function as its primitive.

Library

The primary exported function besides pretty printing and testing functions from the MCTS. The function takes some parameters to tune the tree search:

- Iterations: the number of times that a full MCTS iteration is performed on the root node to expand the game tree.
- Rollout: The number of nodes that are simulated in the Simulation step from the children added in the Expansion step.
- First: The player who moves at the root of the game tree (moves first).

[3] https://www.cs.swarthmore.edu/~mitchell/classes/cs63/f20/reading/mcts.html

```haskell
-- Number of iterations, number of rollouts, initial player, starting state
mcts :: GameState g => Int -> Int -> Player -> g -> g
mcts iter rollout first s = gameState choice
    where choice = getGameData $ maximumBy (compare `on` getScore . getGameData) ch
          Node _ ch = applyNtimes iter (step rollout) $ root first s
```

Implementation: Parallelism

According to the wikipedia article, there are 3 main ways to parallelize MCTS:

- Leaf parallelization, i.e. parallel execution of many playouts from one leaf of the game tree.
- Root parallelization, i.e. building independent game trees in parallel and making the move based on the root-level branches of all these trees.
- Tree parallelization, i.e. parallel building of the same game tree…

Of these, leaf parallelization was the one implemented in the project. This was accomplished by using *Control.Parallel.Strategies*. An evaluation strategy was plugged into the expression which simulates the *rollout* children nodes in the Simulation step:

```haskell
    let results = map sim toSimulate `using` parList rseq --parallelism
```

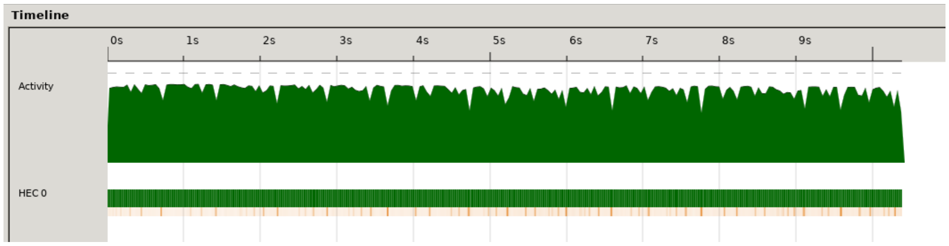
Using threadscope, we can compare the performance of the simulation executable when run on 1, 2, and 4 cores. The command run was

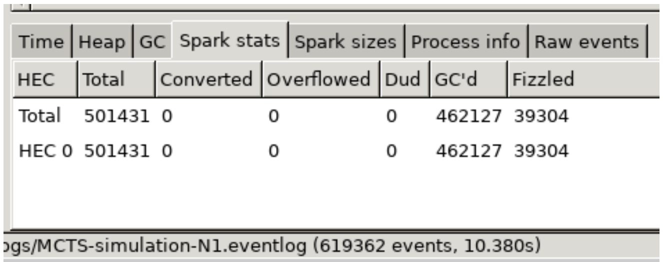
```
stack exec MCTS-simulation 5000 6 209 -- +RTS -ls -C0.01 -NX
```

Where X was the number of cores that were to be used in the execution of the simulation. Two MCTS agents played against each other until a game was completed.

From the profiling, it can be seen that sparks were consumed in the two core and four core runs. Nevertheless, the time taken only increased. This can partially be explained by the simplicity of connect four. Since only leaf parallelization was implemented, the parallelization was used to distribute the simulations across the different cores. However, due to the fact that simulation is not an intensive part of MCTS for Connect Four, the overhead of spark scheduling overtook any benefits that leaf parallelization could have offered. In the future, more complex games with more resource intensive simulations could be tried. In the case of Connect Four, ever move necessarily brings the game closer to ending. For something like chess, this is not necessarily the case, and as such a uniform random playout may take a very very long time.

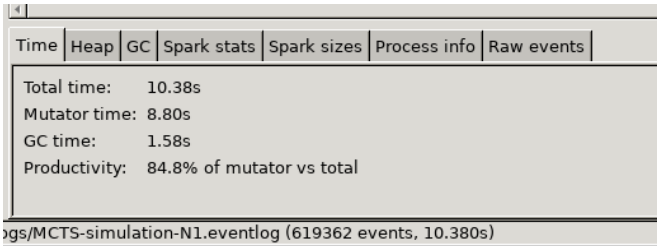
One Core (-N1):





HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	501431	0	0	0	462127	39304
HEC 0	501431	0	0	0	462127	39304

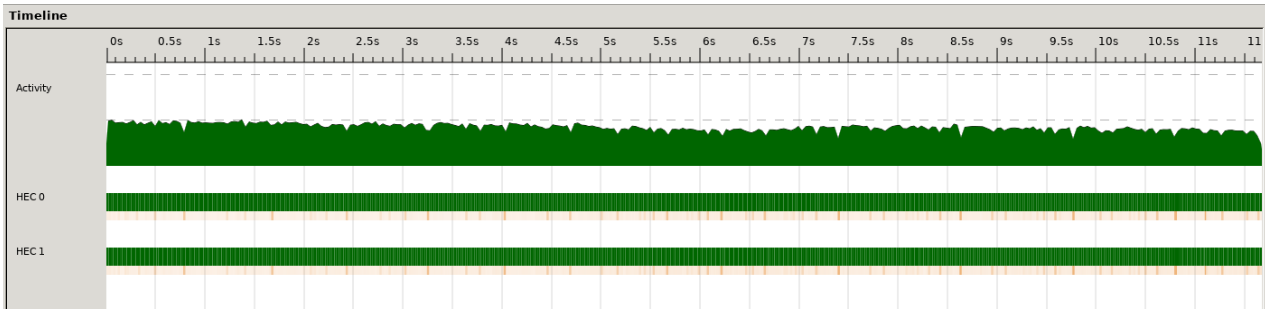
ogs/MCTS-simulation-N1.eventlog (619362 events, 10.380s)

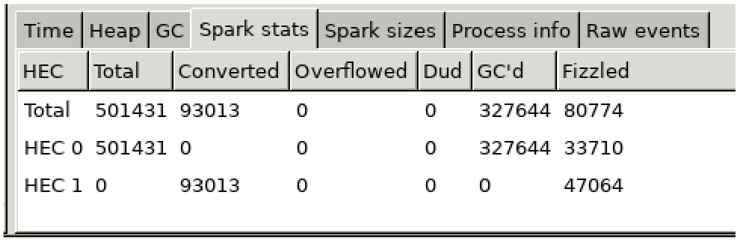


```
Total time:     10.38s
Mutator time:   8.80s
GC time:        1.58s
Productivity:   84.8% of mutator vs total
```

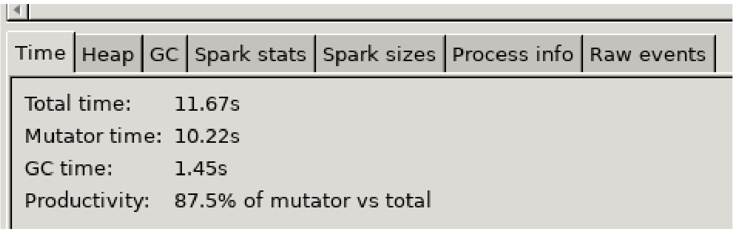
ogs/MCTS-simulation-N1.eventlog (619362 events, 10.380s)

Two Cores (-N2):





HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	501431	93013	0	0	327644	80774
HEC 0	501431	0	0	0	327644	33710
HEC 1	0	93013	0	0	0	47064



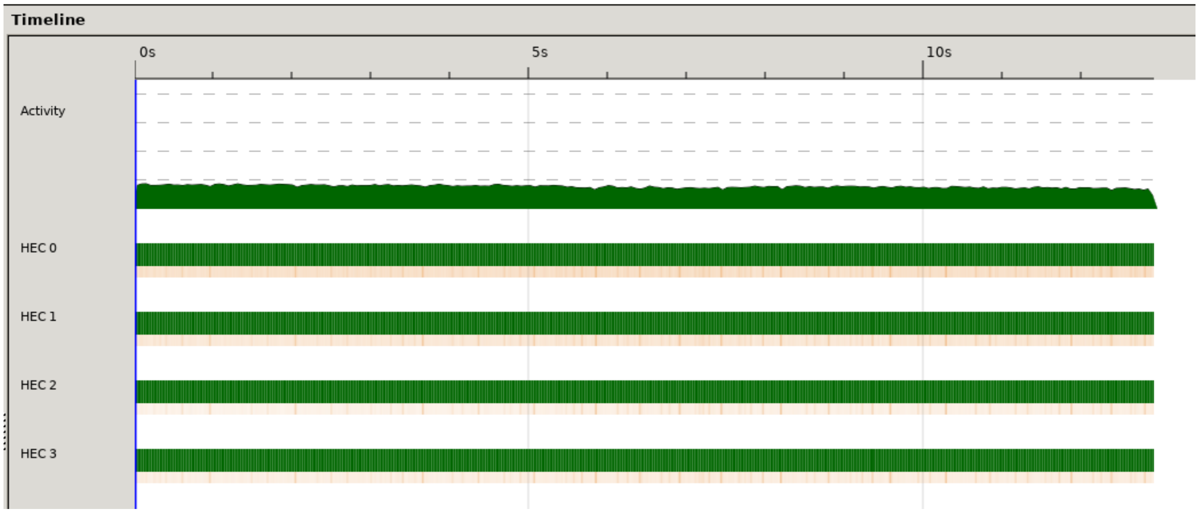
Total time: 11.67s

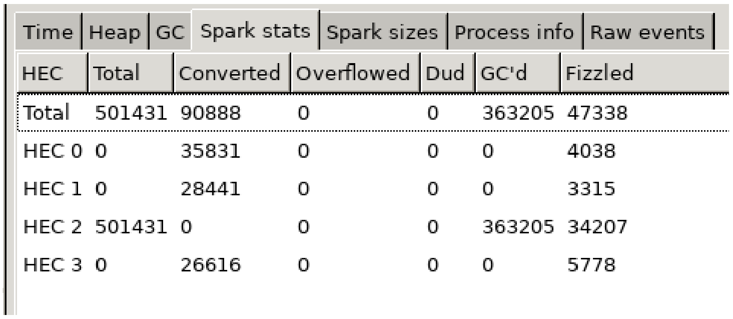
Mutator time: 10.22s

GC time: 1.45s

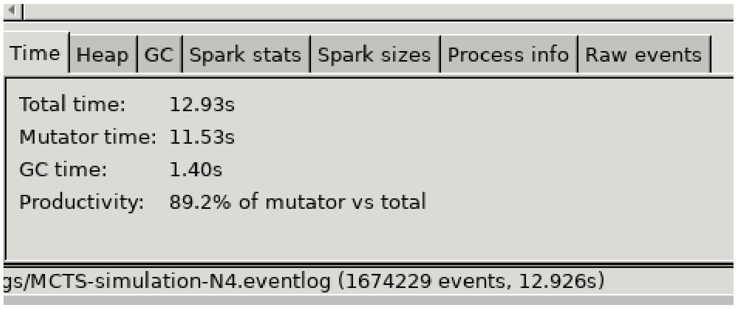
Productivity: 87.5% of mutator vs total

Four Cores (-N4):





HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	501431	90888	0	0	363205	47338
HEC 0	0	35831	0	0	0	4038
HEC 1	0	28441	0	0	0	3315
HEC 2	501431	0	0	0	363205	34207
HEC 3	0	26616	0	0	0	5778



Total time: 12.93s
Mutator time: 11.53s
GC time: 1.40s
Productivity: 89.2% of mutator vs total

gs/MCTS-simulation-N4.eventlog (1674229 events, 12.926s)

Code Listing:

src/MCTS.hs

```haskell
-- stack --resolver lts-19.23 ghci

module MCTS
    (
        Player(..),
        opposing,

        GameState,
        next,
        eval,
        pick,
        sim,

        mcts,
        drawGameTree,

        testMCTS,
        testStep
    ) where


{-# LANGUAGE InstanceSigs #-}
import Control.Monad.State
import Data.Tree
import Data.List(sortBy, maximumBy)
import Data.Function
import Control.Parallel.Strategies

data Player = One | Two | Tie deriving (Eq, Show)

class GameState g where
    next :: g -> [g]  -- gets gamestates available from given
    eval :: g -> Maybe(Player) -- determines the winner of the given gamestate
    pick :: g -> [g] -> g -- picks a gamestate from a list of available ones

    sim :: g -> Player  -- gets outcome of a game state (default is simulation)
    sim g = case next g of
                [] -> case eval g of
                        Just result  -> result
                        Nothing      -> Tie
```

```
-- if nothing new is possible, and no clear outcome, it's a Tie

                gs -> let picked = pick g gs
                      in case eval picked of
                         Just result  -> result
                         Nothing      -> sim picked

opposing :: Player -> Player
opposing One = Two
opposing Two = One
opposing Tie = Tie

-- Number of iterations, number of rollouts, initial player, starting state
mcts :: GameState g => Int -> Int -> Player -> g -> g
mcts iter rollout first s = gameState choice
    where choice = getGameData $ maximumBy (compare `on` getScore .
getGameData) ch
          Node _ ch = applyNtimes iter (step rollout) $ root first s



-----------------------------------------
-- MCTS Data Structures
-----------------------------------------

-- Holds info relevant to MCTS in each node of the tree
data GameData g = GameData {wins :: Int, total :: Int, player :: Player,
gameState :: g}

showGameData :: Show g => GameData g -> String
showGameData (GameData w t p g) = show w ++ ", " ++ show t ++ ", " ++ show p ++
", " ++ show g

getScore :: GameData g -> Double
getScore GameData{wins=w, total=t} = (fromIntegral w) / (fromIntegral t) ::
Double

type GameResult = Player

-- Takes a game result specifying the win amount and player who won to update
game data
updateWins :: GameResult -> GameData g -> GameData g
updateWins pl (GameData w t p g)
    | p == pl    = GameData (w+1) (t+1) p g
    | otherwise = GameData w (t+1) p g
```

```
-- The MCTS game tree which is incrementally created
type GameTree g = Tree (GameData g)

drawGameTree :: Show g => Tree (GameData g) -> String
drawGameTree = drawTree . fmap showGameData

getGameData :: GameTree g -> GameData g
getGameData = rootLabel

root :: Player -> g -> GameTree g
root p g = Node (GameData 0 0 p g) []

-----------------------------------------
-- MCTS Algorithm
-----------------------------------------

-- Number of rollouts, and a game tree
step :: GameState g => Int -> GameTree g -> GameTree g
step r t = evalState (walk t) ([], r)

ucb :: GameData g -> GameData g -> Double
ucb GameData{total=p_total} GameData{wins=c_wins, total=c_total} = (w / n) + c
* sqrt (log np / n)
    where n  = fromIntegral c_total --infinity when 0: desired behavior!
          np = fromIntegral p_total
          w  = fromIntegral c_wins
          c  = sqrt (2.0 :: Double)

-- Returns a list of child GameData from given GameData
possibleMoves :: GameState g => GameData g -> State ([GameResult], Int)
[GameData g]
possibleMoves GameData{gameState=g, player=p} = do
    (_, rollout) <- get

    let states  = next g

    let (toSimulate, other) = splitAt rollout states

    let results = map sim toSimulate `using` parList rseq --parallelism

    let simulatedChildren = zipWith updateWins results [GameData 0 0 (opposing
p) gs | gs <- toSimulate ]

    let otherChildren =  [GameData 0 0 (opposing p) gs | gs <- other ]
```

```haskell
    put (results, rollout)

    return $ simulatedChildren ++ otherChildren


-- Returns a list of children tree nodes created from given node's game state
expand :: GameState g => GameTree g -> State ([GameResult], Int) [GameTree g]
expand (Node d ch) = do
    if total d == 0 --if never visited
        then return [] --we create no children (part of MCTS)
        else do
            moves <- possibleMoves d
            let newChildren = [ Node cd [] | cd <- moves]
            return $ ch++newChildren

-- Takes a tree, traverses to a leaf using UCB, then expands it
-- Use the state monad to propagate upwards the list of winning player in simulated/evaluated nodes as a state
walk :: GameState g => GameTree g -> State ([GameResult], Int) (GameTree g)

walk n@(Node d []) = do  -- Leaf node
    newChildren <- expand n
    case newChildren of
 -- If no tree children created, simulate game from leaf and update it with results
        [] -> do
            let result = sim $ gameState d
            (_, rollout) <- get
            put ([result], rollout)
            let updatedData = updateWins result d
            return $ Node updatedData []
 -- If tree children created, some games were simulated. Update current node with sim results.
        ch -> do
            (results, _) <- get
            let updatedData = foldr updateWins d results
            return $ Node updatedData ch

walk (Node d ch) = do -- Branch node
    updatedChild    <- walk selected
    (results, _)    <- get
    let children    = updatedChild:rest
    let updatedData = foldr updateWins d results
    return $ Node updatedData children
    where
```

```haskell
          selected:rest = sortBy compareUCB ch
          compareUCB = compare `on` ((*(-1)) . ucb d . getGameData) --max instead of min hence *-1

----------------------------
-- TESTING
----------------------------

testStep :: (Show g, GameState g) => Int -> Int -> Player -> g -> IO ()
testStep n r p g = (putStrLn . drawGameTree) (applyNtimes n (step r) $ root p g)

testMCTS :: (Show g, GameState g) => Int -> Int -> Player -> g -> IO ()
testMCTS n r p g = print $ mcts n r p g


----------------------------
-- Helpers
----------------------------

-- From https://gist.github.com/thekarel/9964975
applyNtimes :: (Num n, Ord n) => n -> (a -> a) -> a -> a
applyNtimes 1 f x = f x
applyNtimes n f x = f (applyNtimes (n-1) f x)
```

src/ConnectFour.hs

```
module ConnectFour (
    simulation,
    game,
    initial
) where

import MCTS
import Data.Matrix
import System.Random
import Data.Vector(findIndex, toList)

----------------------------
-- CONNECT 4 IMPLEMENTATION
----------------------------
data Color = Red | Yellow | Blank deriving (Eq)

instance Show Color where
    show Red = "R"
    show Yellow = "Y"
    show Blank = "."

other :: Color -> Color
other Red    = Yellow
other Yellow = Red
other Blank  = Blank

toPlayer :: Color -> Player
toPlayer Red    = One
toPlayer Yellow = Two
toPlayer Blank  = error "No corresponding player"

------------------------------------------
-- Game specific helpers
------------------------------------------
place :: Matrix Color -> Int -> Int
place brd = getRowIndex . findIndex (/= Blank) . \j -> getCol j brd
    where getRowIndex (Just i) = i
          getRowIndex Nothing = nrows brd

maxRun :: [Color] -> Int
maxRun []  = 0
maxRun lst@(h:_) = (case h of
                        Blank -> 0
                        _     -> length run) `max` maxRun rest
            where (run, rest) = span (== h) lst

------------------------------------------
-- MCTS GameState instance definitions!
------------------------------------------
data (RandomGen r) => ConnectFourState r = State {  board :: Matrix Color,
                                                    currentPlayer :: Color,
                                                    lastMove :: (Int, Int),
                                                    rng :: r }

instance (RandomGen r) => Show (ConnectFourState r) where
    show (State b p l _) = "\n" ++ show b ++ "\ncurrent player: " ++ show p ++
"\nlast move: " ++ show l

instance (RandomGen r) => GameState (ConnectFourState r) where
    next State{board=b, currentPlayer=p, rng=r} = states
        where
              states = zipWith (\childBoard move -> State childBoard nextPlayer
move newRNG) boards indeces

              nextPlayer = other p
              newRNG = snd $ split r

              boards = map (\(i,j) -> setElem p (i,j) b) indeces

              indeces = zip rowIndeces colIndeces

              -- Get colummns corresponding to indeces then
              -- find index of first row that isn't blank
              rowIndeces = map (place b) colIndeces
              -- Get column indeces where there is space
              colIndeces = filter (\j -> (getElem 1 j b) == Blank) [1..ncols b]

    eval State{board=b, currentPlayer=p, lastMove=l} = if isRun
        then Just(toPlayer $ other p)
        else Nothing
        where
              -- If a run is encountered, last move completed it, hence last
player won
              isRun = maxRun sndDiag >= 4 || maxRun fstDiag >= 4 || maxRun row
>= 4 || maxRun col >= 4

              sndDiag = [getElem (r+i) (c+j) b | (i, j) <- zip [-7..7]
```

```
[7,6..(-7)], 1 <= r+i && r+i <= 6 && 1 <= c+j && c+j <= 7]
              fstDiag = [getElem (r+i) (c+j) b | (i, j) <- zip [-7..7] [-7..7],
1 <= r+i && r+i <= 6 && 1 <= c+j && c+j <= 7]
              row = Data.Vector.toList $ getRow r b
              col = Data.Vector.toList $ getCol c b

              (r, c) = l

    pick State{rng=r} states = states !! i
        where (i, _) = uniformR (0 :: Int, length states - 1) r

initial :: Int -> ConnectFourState StdGen
initial seed = State (matrix 6 7 $ \_ -> Blank) Red (-1,-1) (mkStdGen seed)

----------------------------
-- ENTRY POINT
----------------------------
game :: Int -> Int -> ConnectFourState StdGen -> Int -> IO ()
game n rollout s@(State b _ _ r) turn = do
    putStrLn $ "Turn " ++ show turn ++ ":\nPlayer's turn (choose a column): "
++ show s ++ "\n"
    j <- readLn :: IO Int
    let i = place b j
    let playerState = State (setElem Red (i,j) b) Yellow (i,j) r
    putStrLn $ show playerState ++ "\n"

    -- win check
    case eval playerState of
        Just(One) -> putStrLn "Player Win!"
        _ -> do
                putStrLn $ "Turn " ++ show (turn+1) ++ ":\nComputer's turn: \n"
                let newState = mcts n rollout Two playerState
                putStrLn $ show newState ++ "\n"

                -- win check
                case eval newState of
                    Just(Two) -> putStrLn "Computer Win!"
                    _ -> game n rollout newState (turn+2)

simulation :: Int -> Int -> ConnectFourState StdGen -> Int -> IO ()
simulation n rollout s@(State _ current _ _) turn = do
    let player = toPlayer current
    putStrLn $ "Turn " ++ show turn ++ ":\nPlayer " ++ show player ++"'s turn
(choose a column): " ++ show s ++ "\n"
```

```
    let newState = mcts n rollout player s
    putStrLn $ show newState ++ "\n"

    -- win check
    case eval newState of
        Just(p) | p == player -> putStrLn $ "Player " ++ show player ++ "(" ++
show current ++ ")" ++ " Win!"
        _ -> simulation n rollout newState (turn+1)
```

app/demo/Main.hs

```haskell
module Main (main) where

import ConnectFour
import System.Environment(getArgs)

main :: IO ()
main = do
    args <- getArgs
    let arg1:arg2:arg3:_ = args
    let n = (read arg1) :: Int
    let rollout = (read arg2) :: Int

    let seed = (read arg3) :: Int
    let initialState = initial seed

    putStrLn $ "MCTS DEMO: CONNECT 4"
    putStrLn $ "Warning: no bounds checking on user input yet please be kind to me."
    game n rollout initialState 1
```

app/simulation/Main.hs

```haskell
module Main (main) where

import ConnectFour
import System.Environment(getArgs)


main :: IO ()
main = do
    args <- getArgs
    let arg1:arg2:arg3:_ = args

    let n = (read arg1) :: Int
    let rollout = (read arg2) :: Int
    let seed = (read arg3) :: Int

    putStrLn $ "MCTS SIMULATION: CONNECT 4"
    simulation n rollout (initial seed) 1
```