Parallel Greedy *Tetris* Solver

COMS 4995 Parallel Functional Programming: Final Project Report
Trey Gilliland (jlg2266), Derek Zhang (dhz2104)

1 Abstract

Tetris is a retro tile-matching based video game in which the objective is to place a stream of descending pieces called "Tetrominoes" to fill up the rows of a 20x10 grid such that the user can place as many pieces as possible without the highest column exceeding 20 units. Placing pieces in a manner to complete the rows as efficiently as possible is advantageous as rows are cleared as they are completed, leaving more space for future Tetrominoes. We implement a heuristic-based search algorithm to develop a parallelized *Tetris* "solver" to place the current Tetromino in the most optimal position based on searching for the most optimal placement of the future N pieces exposed to the user. Our Haskell implementation is ~360 lines and our parallelization led to a 2.89x speedup over the sequential search implementation.

2 Background

Among the many different versions of *Tetris*, we decided to develop our solver on the original version of the game. In this version, the sole objective of the user is to place as many Tetrominoes as possible on a 20x10 board without the tallest column of placed units in the board exceeding the 20 unit height limit. Each time a row is filled completely with units the row is cleared from the board, leaving more room for future Tetrominoes. These Tetrominoes descend 1 unit at a time from the top of the board and as the user places more Tetrominoes, the descending speed increases. The user is able to rotate and move the Tetromino left and right while it is descending. There are 7 different Tetrominoes made up of 4 units each in various configurations as seen in Figure 2.1. When considering their rotations as unique pieces, there are a total of 19 different piece configurations.

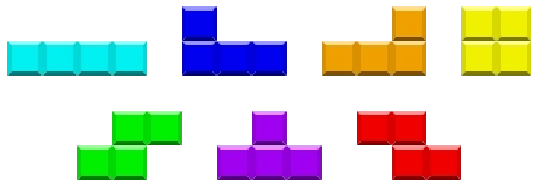


Figure 2.1: All 7 Tetrominoes used in the classic *Tetris* game

3 Search Algorithm

Our solver is based around a heuristic-based depth-breadth hybrid search on all possible Tetromino placements on a given board. Our heuristics, weights, and approach has been adapted from a sequential JavaScript implementation by Yiyuan Lee[1]. The depth of the search can be easily adjusted and performance varies as the search grows exponentially. As the entire search space for a

[1] https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

single Tetromino is excessively large and we could not search multiple depths in a reasonable amount of time, we use a “greedy” approach where we assume the optimal placement for a Tetromino can be achieved by placing all of its possible rotations in each of the board’s columns and choosing the placement that returns the highest score for the resulting board. The formula we use to score a board can be seen in Figure 3.1. The heuristics are as follows:

1. AggregateHeight is a sum over all the column heights and should be minimized as the user wants to keep their columns from exceeding the board height limit.
2. LinesCleared is a count of all the complete lines in a board prior to the rows being cleared and should be maximized as clearing a line leaves more room to place future Tetrominoes.
3. Holes is a count of all the enclosed gaps in the board and should be minimized as the row above enclosing the gap would need to be cleared before the row containing the gap could be cleared from the board.
4. Bumpiness is a sum of all absolute value differences of adjacent columns and should be minimized so that the column heights are evenly distributed allowing for a greater chance of completing a row.

$$score = \left[AggregateHeight,\ LinesCleared,\ Holes,\ Bumpiness\right] \cdot \left[-.510066,\ .760666,\ -.35663,\ -.184483\right]$$

Figure 3.1: Scoring formula with current heuristic weights

4 Implementation

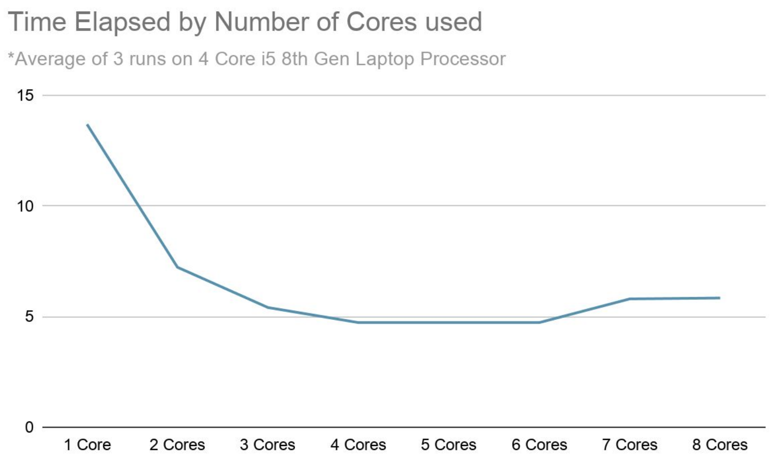
Our implementation revolves around the *Data.Matrix* package to represent a *Board*. A single rotation of a *Tetromino* can be represented as the *Piece* type: a list of 4 coordinate offsets from the bottom-left corner of a 4x4 matrix. Each coordinate represents the spot where a unit would be placed in a 4x4 matrix to represent a Tetromino rotation. The *Tetromino* newtype consists of a list of *Pieces* which represent its rotations. We use 7 *Tetromino* instances (I, O, Z, S, T, J, L) to match the 7 possible Tetrominoes used in *Tetris.* A *BoardMove* is a representation of the current state of the board in between placements and consists of a *Board, Piece* representing the last piece placed on the board, and a double representing the score of the board. A *BoardState* is an intermediate type used in the search algorithm consisting of a *Board*, the current *Piece* to be placed, and the current location of the *Piece* prior to being placed. Placing the piece in a *BoardState* converts it into a *BoardMove* and is outputted to the console at each step.

Our search is centered around a *getBestMove* method which takes in a list of the next N Tetrominoes exposed to the user and a *Board* to return a *BoardMove* representing the best placement of the first Tetromino in the list onto the board. This method does this by generating a list of all possible *BoardMoves* using *getPossibleBoards* where the score is the score of the best *BoardMove* in the deepest level of the search path extending from the placement and selects the Board with the highest score to be the most optimal placement. The search tree is searched over through the *nextTick* method which places the first Tetromino on the board in a particular rotation and column, and then searches its search subtree for the best possible score on the final depth level.

The user can supply a number N representing the number of N Tetrominoes to place onto an empty board through the *placeN* method on an interactive console or a filename containing a space separated string of numbers to run through a compiled executable.

5 Parallelization

In order to parallelize our code, we used the Par monad. The Par monad provides a simple API that enables deterministic, dataflow-based parallel programming. The command *runPar* completely evaluates Par monad's functions in parallel before continuing. Up to a point, therefore, increasing the number of cores decreases the time necessary to complete all operations within the monad. In our *getBestMove* function, we used a *parMap* to split and parallelize each row-piece rotation combination and run the *nextTick* function in parallel. As can be seen in the chart below, which was created by running our algorithm using the same input Tetromino list over a variety of cores, this method provides significant speedups when the number of available cores is increased. As the number of cores continues to increase, however, the high number of cores does more harm than good, and the time elapsed ends up increasing again. This is because there is significant overhead in managing garbage collection and synchronizing between the different threads, causing a drop in performance.



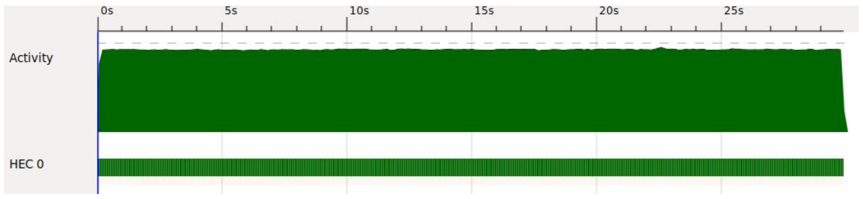
We found that the Par monad provides superior performance compared to other methods of parallelization like the Eval monad. This is likely because the Par monad forces parallel computation by manually creating the user-specified parallel tasks, whereas the Eval monad can create a large number of sparks that can often fizzle or be garbage-collected, depending on the compiler settings and flags used. Because we can manage the threads created more directly, the Par monad provides more consistent speedups and load distribution. However, both methods are a means to the same end goal of speeding up the program by distributing work across multiple processors.

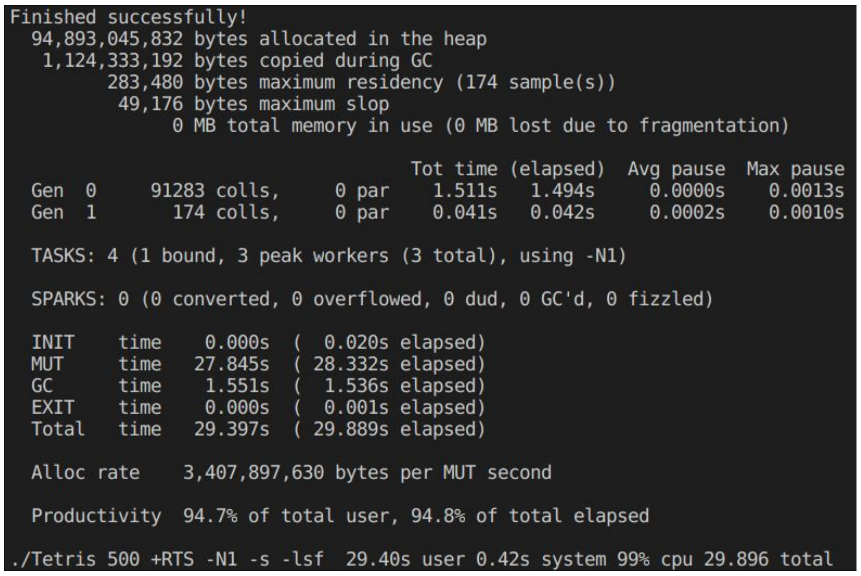
This parallelization significantly reduces the time necessary to calculate the best possible score of the current piece rotation and column, and given the next *N* pieces. However, there were other parts of the program that we did not parallelize, such as the printing of the boards at each step and the *getBestBoard* function. However, our load distribution and performance improvements were still very good because the *nextTick* calculation is by far the most expensive and time consuming operation within the program, as it traverses all possible search trees for the next *N* pieces via a DFS to find the optimal board score for the current placement.

6 Performance

Using the Par monad enabled us to get significant performance benefits over our original sequential implementation. In the following examples, we looked ahead by four pieces to determine the optimal board score for each piece rotation-column combination at each step. As can be seen in the command-line output, the total time for the multi-core runs exceeds the time elapsed, showing that the different cores were doing a large amount of work in parallel. As the number of cores increases, the GC time and the total time increase, showing that a large amount of overhead is necessary to synchronize and manage the different cores. Despite this, due to parallelization, the overall performance improves significantly. In the end benchmark, we achieved a 2.89x speedup over the sequential search implementation.

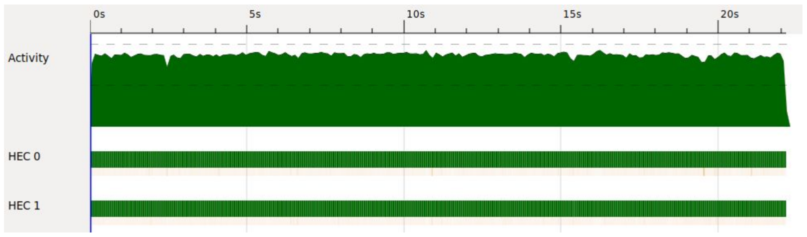
1 Core:

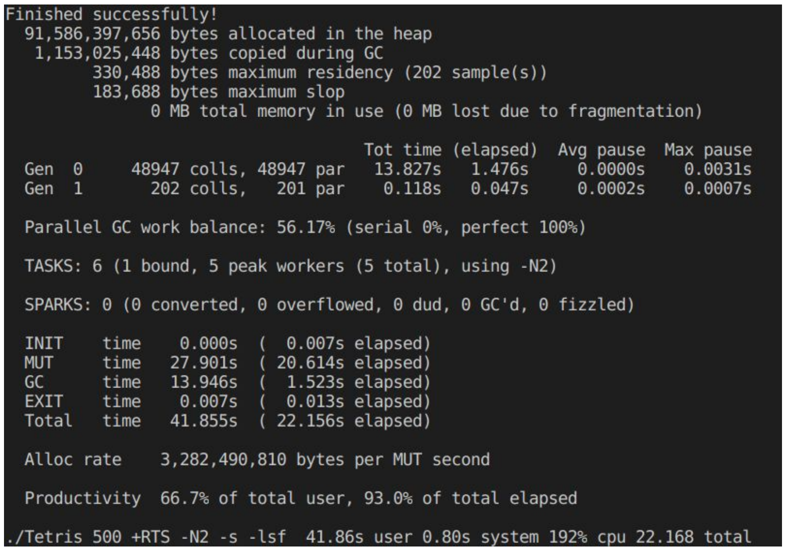




```
Finished successfully!
  94,893,045,832 bytes allocated in the heap
   1,124,333,192 bytes copied during GC
         283,480 bytes maximum residency (174 sample(s))
          49,176 bytes maximum slop
               0 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     91283 colls,     0 par    1.511s   1.494s     0.0000s    0.0013s
  Gen  1       174 colls,     0 par    0.041s   0.042s     0.0002s    0.0010s

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

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

  INIT    time    0.000s  (  0.020s elapsed)
  MUT     time   27.845s  ( 28.332s elapsed)
  GC      time    1.551s  (  1.536s elapsed)
  EXIT    time    0.000s  (  0.001s elapsed)
  Total   time   29.397s  ( 29.889s elapsed)

  Alloc rate    3,407,897,630 bytes per MUT second

  Productivity  94.7% of total user, 94.8% of total elapsed

./Tetris 500 +RTS -N1 -s -lsf  29.40s user 0.42s system 99% cpu 29.896 total
```

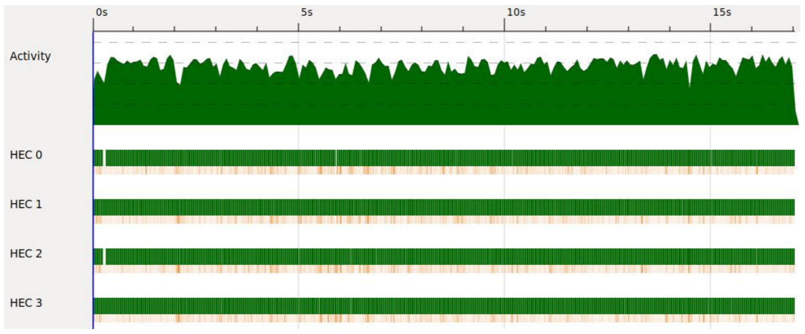
2 Cores:

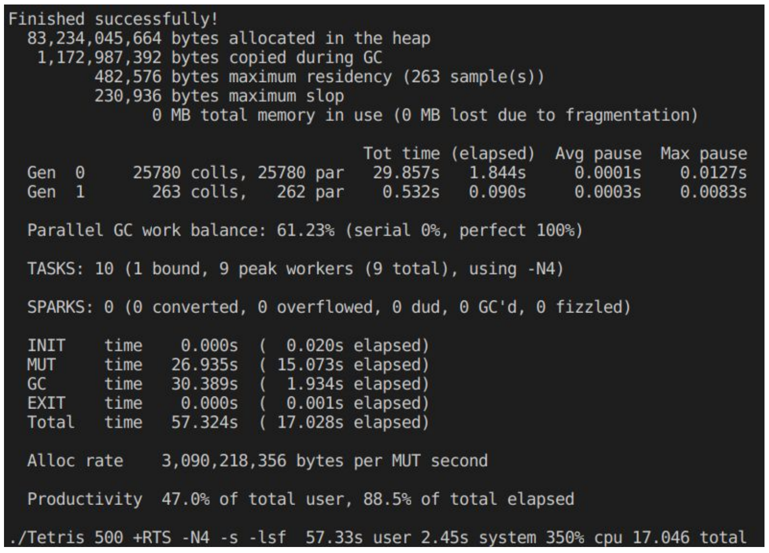




```
Finished successfully!
  91,586,397,656 bytes allocated in the heap
   1,153,025,448 bytes copied during GC
         330,488 bytes maximum residency (202 sample(s))
         183,688 bytes maximum slop
               0 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     48947 colls, 48947 par   13.827s   1.476s     0.0000s    0.0031s
  Gen  1       202 colls,   201 par    0.118s   0.047s     0.0002s    0.0007s

  Parallel GC work balance: 56.17% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N2)

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

  INIT    time    0.000s  (  0.007s elapsed)
  MUT     time   27.901s  ( 20.614s elapsed)
  GC      time   13.946s  (  1.523s elapsed)
  EXIT    time    0.007s  (  0.013s elapsed)
  Total   time   41.855s  ( 22.156s elapsed)

  Alloc rate    3,282,490,810 bytes per MUT second

  Productivity  66.7% of total user, 93.0% of total elapsed

./Tetris 500 +RTS -N2 -s -lsf  41.86s user 0.80s system 192% cpu 22.168 total
```

<u>4 Cores:</u>





```
Finished successfully!
  83,234,045,664 bytes allocated in the heap
   1,172,987,392 bytes copied during GC
         482,576 bytes maximum residency (263 sample(s))
         230,936 bytes maximum slop
               0 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     25780 colls, 25780 par   29.857s   1.844s     0.0001s    0.0127s
  Gen  1       263 colls,   262 par    0.532s   0.090s     0.0003s    0.0083s

  Parallel GC work balance: 61.23% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

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

  INIT    time    0.000s  (  0.020s elapsed)
  MUT     time   26.935s  ( 15.073s elapsed)
  GC      time   30.389s  (  1.934s elapsed)
  EXIT    time    0.000s  (  0.001s elapsed)
  Total   time   57.324s  ( 17.028s elapsed)

  Alloc rate    3,090,218,356 bytes per MUT second

  Productivity  47.0% of total user, 88.5% of total elapsed

./Tetris 500 +RTS -N4 -s -lsf  57.33s user 2.45s system 350% cpu 17.046 total
```

7 Possible Future Work

Tetris has hundreds of official and unofficial variants which all have their own unique twist on platforms, scoring, size of the board, tetrominoes, multiplayer, and more. Our algorithm has been designed with this in mind and could easily be modified to work using different style Tetrominos, board sizes, number of future Tetrominoes exposed to the user (representing search depth), search heuristics, and more as long as the most basic *Tetris* rules are followed. A search over the entire possible search space for a Tetromino placement could be designed and easily inserted into our code as long as it could be interfaced through the *getBestMove* method type constraints. Finally, despite looking forward to a certain number of pieces, the board sometimes still fills up completely after a very large number of pieces have been placed. With more time, we could tune the weights and add additional heuristics in order to prevent failure.

8 Code

Our code consists of a main file Tetris.hs and 3 supporting modules: AI.hs, Helpers.hs, and Types.hs In total, our code (including comments) spans ~360 lines all written in Haskell. We utilize the Data.Matrix, System.Random, and Control.Monad.Par libraries and they will need to be installed to compile our code.

Tetris.hs

```haskell
-- Imports
import AI
import Control.Applicative
import Control.Monad.Par
import Data.List
import Helpers
import System.Environment
import System.Exit
import System.IO (hPutStrLn, stderr)
import Types

-- MAIN METHOD
-- main method version for reading space-separated integers from 0-6,
representing a predefined tetromino order
main :: IO ()
main = do
  args <- getArgs
  case args of
    [filename] -> do
      -- read in file
      -- convert to lines
      -- do this function below for all lines in file
      contents <- readFile filename
      let strings = words contents
          li = map read strings :: [Int]
          board = createBoard
          pieceList = map randomPiece li -- list of random pieces
          tupleList = take (length pieceList - 4 + 1) (map (take 4) (tails
pieceList))
      gameloop tupleList (Just (BoardMove board Nothing 0))
      return ()

    _ -> do
      pn <- getProgName -- Usage message
      hPutStrLn stderr $ "Usage: " ++ pn ++ " <file name>"
      exitFailure

-- -- ALTERNATE MAIN METHOD
-- -- gets the N input from the user and places N random pieces in their
best position starting from an empty board
-- main :: IO ()
-- main = do
--   args <- getArgs
--   case args of
--     [n] -> do
--       li <- generateNList (read n :: Int)
--       let board = createBoard
--           pieceList = map randomPiece li -- list of random pieces
--            searchDepth = 4
--            -- create a search depth length list of next pieces, currently
using a search depth of 4
--            tupleList = take (length pieceList - searchDepth + 1) (map
(take searchDepth) (tails pieceList))
--        gameloop tupleList (Just (BoardMove board Nothing 0))
--        return ()

--      _ -> do
--        pn <- getProgName -- Usage message
--        hPutStrLn stderr $ "Usage: " ++ pn ++ " <n tetrominos>"
--        exitFailure

-- interactive method for passing in an N and running the game loop for a
randomPiece list sequentially
placeN :: Int -> IO ()
placeN n = do li <- generateNList n
              let board = createBoard
                  pieceList = map randomPiece li -- list of random pieces
                  searchDepth = 4
                  -- create a search depth length list of next pieces,
currently using a search depth of 4
                  tupleList = take (length pieceList - searchDepth + 1)
(map (take searchDepth) (tails pieceList))
              gameloop tupleList (Just (BoardMove board Nothing 0))
              return ()

-- recursive helper method for main to place all pieces in a Tetromino list
gameloop :: [[Tetromino]] -> Maybe BoardMove -> IO ()
gameloop _ Nothing = putStrLn "Failed to complete board!"
gameloop [] _ = putStrLn "Finished successfully!"
gameloop (current : rest) (Just (BoardMove board piece score)) = do
  putStrLn $ show (BoardMove board piece score)
  boardMove <- getBestMove current board
  gameloop rest boardMove

-- Given a list of next N Tetrominos and a board, return the best possilbe
placement for the first Tetromino in the list
getBestMove :: [Tetromino] -> Board -> IO (Maybe BoardMove)
getBestMove [] _ = return Nothing
getBestMove (Tetromino pieces : xs) board = do
  let li = runPar $ parMap (\(piece, c) -> startNextTick xs (BoardState
board piece (5, c))) (liftA2 (,) pieces [1 .. 10])
```

```
      ans = getBestBoard li
  return ans

-- return a list of all possible placements of the first Tetromino
searching over the search space for the rest of the Tetrominoes to
calculate the best placement
getPossibleBoards :: [Tetromino] -> Board -> [Maybe BoardMove]
getPossibleBoards [] _ = [Nothing]
getPossibleBoards ((Tetromino pieces) : xs) board = map (\(piece, c) ->
nextTick xs (BoardState board piece (5, c)) GoDown) (liftA2 (,) pieces [1
.. 10])

-- Action Enum type to represent which action to do in nextTick
data Action = GoDown | Place

-- search the search path for the best board at max depth for a given piece
and location representing the column to be placed in
startNextTick :: [Tetromino] -> BoardState -> Maybe BoardMove
startNextTick tetrominos bs = nextTick tetrominos bs GoDown

nextTick :: [Tetromino] -> BoardState -> Action -> Maybe BoardMove
nextTick tetrominos (BoardState board piece loc) GoDown
  | doesNotOverlap (BoardState board piece loc) =
    getBestBoard [nextTick tetrominos (BoardState board piece (fst loc + 1,
snd loc)) GoDown, nextTick tetrominos (BoardState board piece (fst loc + 1,
snd loc)) Place]
  | otherwise = Nothing
nextTick tetrominos (BoardState board piece loc) Place
  | not $ isValidPlacement (BoardState board piece loc) = Nothing
  | otherwise = do
    newBoard <- putPiece (BoardState board piece loc)
    let newBoardCleared = clearRows newBoard
        bestBoard [] = Nothing
        bestBoard (_ : xs) = getBestBoard $ getPossibleBoards xs
newBoardCleared
    return $ BoardMove newBoardCleared (Just piece) (getScore (bestBoard
tetrominos) newBoard)
  where
    getScore (Just (BoardMove _ _ score)) _ = score
    getScore Nothing currentBoard = scoreBoard currentBoard

-- given a list of scored boards in BoardMoves, return the board with the
highest max score
```

```
getBestBoard :: [Maybe BoardMove] -> Maybe BoardMove
getBestBoard boardMoves = getBestBoardHelper Nothing boardMoves

getBestBoardHelper :: Maybe BoardMove -> [Maybe BoardMove] -> Maybe
BoardMove
getBestBoardHelper currentBest [] = currentBest
getBestBoardHelper Nothing (x : xs) = getBestBoardHelper x xs
getBestBoardHelper (Just (BoardMove board piece score)) ((Just (BoardMove
newBoard newPiece newScore)) : xs)
  | newScore > score = getBestBoardHelper (Just (BoardMove newBoard
newPiece newScore)) xs
  | otherwise = getBestBoardHelper (Just (BoardMove board piece score)) xs
getBestBoardHelper currentBest (_ : xs) = getBestBoardHelper currentBest xs
```

AI.hs

```
module AI where

-- Imports
import Data.Matrix
import Types
import Helpers

-- weights type used to calculate the score
data Weights = Weights
 { heightWeight :: Double,
   linesWeight :: Double,
   holesWeight :: Double,
   bumpinessWeight :: Double
 }

-- method to return hardcoded weights for use in scoring function
getWeights :: Weights
getWeights =
 Weights
   { heightWeight = 0.810066,
     linesWeight = 0.760666,
     holesWeight = 0.35663,
     bumpinessWeight = 0.184483
   }
```

```haskell
-- score a board using a given set of weights
scoreBoardWithWeights :: Board -> Weights -> Double
scoreBoardWithWeights board weights = linesVal - heightVal - holesVal -
bumpinessVal
 where
   heightVal = (heightWeight weights) * (fromIntegral $ aggregateHeight
board)
   linesVal = (linesWeight weights) * (fromIntegral $ completeLines board)
   holesVal = (holesWeight weights) * (fromIntegral $ holes board)
   bumpinessVal = (bumpinessWeight weights) * (fromIntegral $ bumpiness
board)

-- score a board using the default set of weights
scoreBoard :: Board -> Double
scoreBoard board = scoreBoardWithWeights board getWeights

-- calculate the aggregateHeight of a board
aggregateHeight :: Board -> Int
aggregateHeight board = sum $ getHeights $ toLists $ transpose board

getHeights :: [[Int]] -> [Int]
getHeights lists = map getHeight lists

getHeight :: [Int] -> Int
getHeight [] = 0
getHeight (x : xs)
 | x == 1 = length xs + 1
 | otherwise = getHeight xs

-- count the completeLines in a board where the rows haven't been cleared
yet
completeLines :: Board -> Int
completeLines board = length $ getFullRowIndexes 0 [] (toLists board)

-- count the number of enclosed holes in a board
holes :: Board -> Int
holes board = foldl (\count arr -> count + getHolesInArr 0 False arr) 0
transposedBoardList
 where
```

```haskell
   transposedBoardList = toLists $ transpose board

getHolesInArr :: Int -> Bool -> [Int] -> Int
getHolesInArr count _ [] = count
getHolesInArr count started (x : xs)
 | (x == 1) && not started = getHolesInArr count True xs
 | (x == 0) && started = getHolesInArr (count + 1) True xs
 | otherwise = getHolesInArr count started xs

-- calculuate the bumpiness of a boards columns
bumpiness :: Board -> Int
bumpiness board = sum $ getHeightDiffs $ getHeights $ toLists $ transpose
board

getHeightDiffs :: [Int] -> [Int]
getHeightDiffs diffs = map abs $ zipWith (-) diffs (drop 1 diffs)
```

Helpers.hs

```haskell
module Helpers where

-- Imports
import Data.Matrix
import System.Random
import Types

-- generates n random numbers in a list, helper for creating random
Tetromino list
generateNList :: Int -> IO [Int]
generateNList n = sequence $ replicate n $ randomRIO (0, 6 :: Int)

-- maps an integer to a Tetromino, helper for creating random Tetromino
list
randomPiece :: Int -> Tetromino
randomPiece r = case r of
 0 -> tetrominoI
 1 -> tetrominoO
 2 -> tetrominoS
 3 -> tetrominoZ
 4 -> tetrominoT
 5 -> tetrominoJ
 6 -> tetrominoL
 _ -> error "invalid piece"

-- BOARD METHODS

-- creates empty board, starting point for placeN call
createBoard :: Matrix Int
createBoard = zero 25 10

-- takes top 5 rows off of matrix (buffer rows for piece placement) and
prints the board
printBoard :: Board -> IO ()
printBoard board = do
 let slice = submatrix 6 25 1 10 board
 print slice
 return ()

-- clears rows of all 1s from a board
clearRows :: Board -> Board
clearRows board = listsAsMatrix
 where
   matrixAsLists = toLists board
   clearFromRows = foldr removeIndex matrixAsLists (getFullRowIndexes 0 []
matrixAsLists)
   removeIndex index list = [replicate 10 0] ++ take index list ++ tail
(drop index list)
   listsAsMatrix = fromLists clearFromRows

-- helper for clearing rows, get indices of all full 1 rows
getFullRowIndexes :: Int -> [Int] -> [[Int]] -> [Int]
getFullRowIndexes _ currList [] = currList
getFullRowIndexes currIndex currList (x : xs) = getFullRowIndexes
(currIndex + 1) (appendedCurrList x currIndex currList) xs
 where
   appendedCurrList vals index list
     | all (== 1) vals = index : list
     | otherwise = list
```

```haskell
-- BOARDSTATE METHODS

-- converts a BoardState into a Board by placing its piece at its location
putPiece :: BoardState -> Maybe Board
putPiece (BoardState board piece loc) = foldr fillLoc (Just board) piece
 where
   fillLoc _ Nothing = Nothing
   fillLoc pieceLoc (Just myBoard) = safeSet 1 (fst loc - fst pieceLoc,
snd loc + snd pieceLoc) myBoard

-- determines if the piece in BoardState could be placed
isValidPlacement :: BoardState -> Bool
isValidPlacement (BoardState board piece loc) = doesNotOverlap (BoardState
board piece loc) && any isOnGround piece && any isBelowBuffer piece && fst
loc > 5
 where
   isBelowBuffer (r, _) = fst loc - r > 5
   isOnGround (r, c)
     | safeGet (fst loc - r + 1) (snd loc + c) board == Just 1 = True
     | (fst loc - r + 1 == nrows board + 1) && (r == 0) = True
     | otherwise = False

-- helper function to isValidPlacement to check if a piece overlaps with
already placed pieces
doesNotOverlap :: BoardState -> Bool
doesNotOverlap (BoardState board piece loc) = all (isValidLoc .
safeGetVal) piece
 where
   safeGetVal (r, c) = safeGet (fst loc - r) (snd loc + c) board
   isValidLoc (Just 0) = True
   isValidLoc _ = False
```

Types.hs

```haskell
{-# LANGUAGE DeriveGeneric #-}

module Types where
```

```haskell
-- Imports
import Data.Matrix
import Control.DeepSeq
import GHC.Generics (Generic)

-- (row, column) representation for the matrix board representation
type Location = (Int, Int)

-- a piece represents a unique rotation of a Tetromino
type Piece = [Location]

-- a representation of the game Board
type Board = Matrix Int

-- Tetromino is a list of all possible rotations of a Tetromino
newtype Tetromino = Tetromino [Piece]

-- BoardState is an intermediate state for the search representing a
board, a piece, and its current location
data BoardState = BoardState Board Piece Location

-- Used to represent the score of a Board and the most recently placed
piece
data BoardMove = BoardMove Board (Maybe Piece) Double deriving Generic

instance NFData BoardMove

-- show method for BoardMove
instance Show BoardMove where
 show (BoardMove board piece score) = "\nScore: " ++ show score ++ "\n
Tetromino: \n" ++ tetrominoBoard ++ "\nBoard:\n" ++ show slice
   where
     tetrominoBoard = tetrominoDisplay blankBoard piece
     blankBoard = zero 4 4
     slice = submatrix 6 25 1 10 board

-- print method for displaying a tetromino onto a board
tetrominoDisplay :: Board -> Maybe Piece -> String
tetrominoDisplay _ Nothing = "No tetromino selected"
tetrominoDisplay board (Just []) = show board
tetrominoDisplay board (Just (x : xs)) = tetrominoDisplay (setElem 1
((nrows board) - (fst x), 1 + snd x) board) (Just xs)

-- all tetromino instances
tetrominoI :: Tetromino
tetrominoI = Tetromino [[(0, 0), (1, 0), (2, 0), (3, 0)], [(0, 0), (0, 1),
(0, 2), (0, 3)]]

tetrominoO :: Tetromino
tetrominoO = Tetromino [[(0, 0), (0, 1), (1, 0), (1, 1)]]

tetrominoZ :: Tetromino
tetrominoZ = Tetromino [[(1, 0), (1, 1), (0, 1), (0, 2)], [(0, 0), (1, 0),
(1, 1), (2, 1)]]

tetrominoS :: Tetromino
tetrominoS = Tetromino [[(0, 0), (0, 1), (1, 1), (1, 2)], [(0, 1), (1, 1),
(1, 0), (2, 0)]]

tetrominoT :: Tetromino
tetrominoT =
 Tetromino
   [ [(0, 0), (0, 1), (0, 2), (1, 1)],
     [(0, 0), (1, 0), (2, 0), (1, 1)],
     [(1, 0), (1, 1), (1, 2), (0, 1)],
     [(1, 0), (0, 1), (1, 1), (2, 1)]
   ]

tetrominoJ :: Tetromino
tetrominoJ =
 Tetromino
   [ [(0, 0), (0, 1), (1, 1), (2, 1)],
     [(0, 0), (1, 0), (0, 1), (0, 2)],
     [(0, 0), (1, 0), (2, 0), (2, 1)],
     [(1, 0), (1, 1), (1, 2), (0, 2)]
   ]

tetrominoL :: Tetromino
tetrominoL =
 Tetromino
   [ [(0, 0), (1, 0), (2, 0), (0, 1)],
     [(0, 0), (1, 0), (1, 1), (1, 2)],
     [(2, 0), (2, 1), (1, 1), (0, 1)],
     [(0, 0), (0, 1), (0, 2), (1, 2)]
   ]
```