PFP Final Project Report: Othello and Minimax

Bryanna Geiger: bg2603

December 24, 2020

Abstract

My final project for Parallel Functional Programming is to implement the board game, Othello, with the Minimax algorithm, focusing on incorporating different board states in Haskell. Othello is a modern day board game involving two players trying to optimize their "points" or the number of tiles of their color by the end of the game. The Minimax algorithm utilizes a tree structure and is a backtracking algorithm commonly used in games such as Othello, 2048, Chess, Checkers, and others. I largely drew from my previously implemented minimax with alpha beta pruning on a game in Python and largely started on a smaller scale tic-tac-toe game to translate that into Haskell. Additionally, due to technical difficulties, I had started the project in advance to account for the added time and included a full list of the references I used at the end of my report.

Figure 1: An example of the Othello board game

1 What is Othello?

Othello is a board game derived from the original game Reversi. The game board itself is an 8x8 square grid. Each player has 32 disk-like black and white pieces - each playing choosing a color to start the game. The initial starting position includes 4 pieces in the middle 4 squares of the board - 2 black and 2 white pieces.

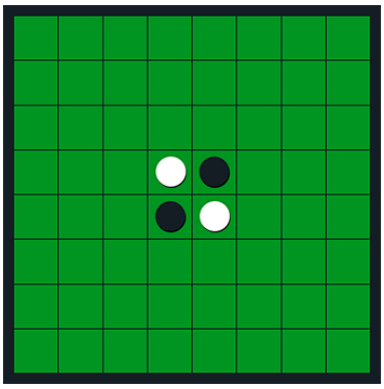

Figure 2: Standard Initial Board State

The aim of the game is to end with more pieces of your color on the board than the opposing player. Player alternate turns and when a valid move is not possible, that player's turn is skipped for the current round. A player may not voluntarily skip their turn when a valid move exists. When a single disk or a row of disks is surrounded at the ends by the opposing color, the surrounded disks will be "flipped" to the opposite color, which is how each player earns points.

The game ends when there are no valid moves remaining for either player or both players are out of disks, thus not being able to make a valid move. The player with more disks of their color wins. If there is an even amount of both color, the game will end in a tie.

2 How Does My Implementation Differ?

The standard rules for ending the game in Othello, is when there are no valid moves remaining for either player. So, if player one can not go, player one's turn would be skipped and player two would go and vice versa. This rule ensures that all pieces will be played and the board will end up being completely filled.

For the sake of my implementation, I did not account for the skipping of turns. My implementation differs in that it does not account for skipping a turn and instead, the game will end when either player does not have a valid move remaining. This alteration to the rules explains why, when running board 1, board 2, and board 3, the end state of the board might not be completely filled.

3 What is Minimax?

The general idea is that minimax is a backtracking algorithm which is a recursive algorithm. As a searching algorithm, minimax is commonly used in games such

as 2048, chess, othello, checkers, and others. It utilizes a tree structure in order to determine the optimal move for each player to make.

In this case, there are two players: "maximizer" and the "minimizer" where each player will either try to maximize the value of a move while the other will minimize the value of a move. It is important, however, to keep in mind that with the tree structure and with a game that has a large number of potential moves and game states, to limit the depth. The default depth I limited my program to was 4, although I also tested at a depth of 2.

My approach to the minimax algorithm was to 1. draw from my previously implemented minimax in Python and 2. draw from the tic-tac-toe game I wrote in Haskell as a starting point. Following a similar format from the tic-tac-toe game is how I arrived at the approach to minimax that I did. Below is an image of what the minimax algorithm might look like on a tic-tac-toe board.

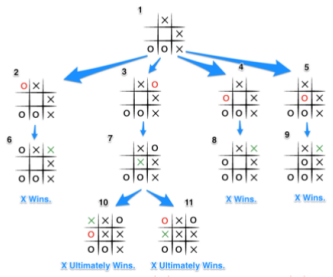

Figure 3: Minimax: Tix-Tac-Toe Example

Different sorts of heuristics can be implemented in order to adjust the values of the board. Some examples of heuristics could be in the game 2048 where it might be ideal to keep the highest valued tile in the corner even at the cost of minimizing the total value on a single tile. In Othello, for instance, a potential heuristic could be giving the corner pieces a significantly higher value. For instance, in an ideal game, the corner pieces should be prioritized over flipping over more your opponents pieces, as the corners are well protected. This heuristic is not one that I had time to implement, however, it is a potential expansion of Othello going forwards. Region 5 in the image below shows the corner pieces which are the ideal spots. There is also a further breakdown of board regions which could be accounted for with heuristics.

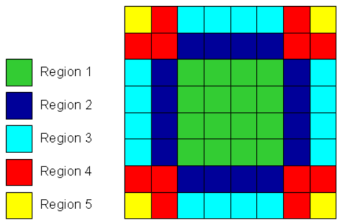

Figure 4: Othello: Region Breakdown

4 Parallelization of Minimax

The part of my final project that I parallelized was the minimax algorithm. The way that I parallelized the minimax algorithm was by using: 'using' parList rseq. The idea was derived from a Sudoku implementation I was reading about. Given that I was using lists, it made sense to implement parList. parList allows us to evaluate eacch element in a list in parallel as sparks according to a given strategy. The overall result is that it did seem to improve the runtime, however, it is just one part of the code that is parallelized. I think if I implemented and parallelized alpha beta pruning, that might decrease the overall runtime of the program, especially given that it takes anywhere from about 20 to 40 seconds to run on 4 cores.

My approach to the minimax algorithm and how to parallelize it, in addition to being derived from Sudoku, I had first applied it to my Tic-tac-toe game in Haskell. A large part of my approach to Othello was derived from 1. my minimax algorithm with alpha beta pruning I previously implemented in Python and 2. the smaller scale version of tic-tac-toe I made in Haskell which I then translated to Othello.

Minimax Psuedocode

```
minimax(in game board, in int depth, in int max_depth,
        out score chosen_score, out score chosen_move)
begin
    if (depth = max_depth) then
        chosen_score = evaluation(board);
    else
        moves_list = generate_moves(board);
        if (moves_list = NULL) then
            chosen_score = evaluation(board);
        else
            for (i = 1 to moves_list.length) do
                best_score = infinity;
                new_board = board;
                apply_move(new_board, moves_list[i]);
                minimax(new_board, depth+1, max_depth, the_score, the_move);
                if (better(the_score, best_score)) then
                    best_score = the_score;
                    best_move = the_move;
                endif
            enddo
            chosen_score = best_score;
            chosen_move = best_move;
        endif
    endif
end.
```

Figure 5: Minimax Pseudocode Example

```haskell
minimax :: Int -> Othello -> Board -> Int
minimax dpth col b
    | endGame = if (adv col b) > 0
        then 100000
        else -100000
    | dpth <= 0 = adv col b
    | otherwise = if ( moves (changeColor col) b ) /= []
        then -maxPt
        else maxPt
    where
        endGame = null (moves col b) && null (moves (changeColor col) b)
        clrUp = if ( moves (changeColor col) b ) /= []
            then changeColor col
            else col
        nm = if clrUp /= col
            then ( moves (changeColor col) b )
            else ( moves col b)
        maxPt = maximum (map (minimax (dpth - 1) clrUp . move clrUp b) nm `using` parList rseq)
```

Figure 6: My Minimax Implementation

5 Implementation of Different Boards

In terms of my board output. I used a text output for the 8x8 grid labeled with numbers 0 thought 7. The underscore indicates a blank space where a player could place a piece. The 'b' indicates where a black piece is. The 'w' indicates where a white piece is. When running this program, we can see where each piece is played and which pieces are flipped over, thus earning points for the player, by looking at the text output.

5.1 Board 1: Default Board

Board 1 is the default board. If no board argument is entered in the command line, it will default to running board 1. Board 1 is the standard board that would be used in an actual Othello game. It starts with an even number of both black and white pieces centered on the board.

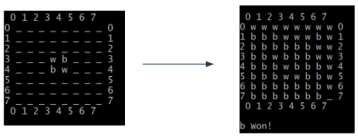

Figure 7: Board 1: Start and End States

5.2 Board 2: Mid-game

Board 2 has a start state that is midway through the game, as opposed to just starting at the initial board state. Board 2 is dominated by black pieces with 10 black pieces and 6 white pieces.

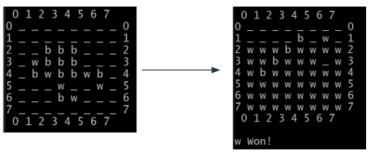

Figure 8: Board 2: Start and End States

5.3 Board 3: Mid-game

Board 3 is similar to board 2 in that it is also midway through the game. They are also at a similar point in the game with a similar number of pieces. Board 3, however, is dominated by white pieces, as opposed to board 2 which is dominated by black pieces. Board 3 starts with 7 black pieces and 11 white pieces. It is a slightly different board configuration than that of board 2.

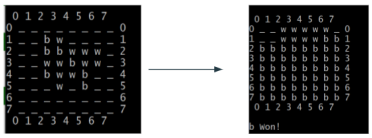

Figure 9: Board 3: Start and End States

6 Threadscope Comparison

I ran threadscope on board 1, board 2, and board 3 testing at 1 core versus 4 cores and a depth of 2 versus a depth of 4. For analysis purposes, the threadscope comparison we will be looking at is of Board 1. I chose to analyze board 1 here using the threadscope visual as it is the standard start state for an actual game of Othello.

The test that is show below was run on board 1 at a depth of 4. The comparison here is between running this program on 1 core versus running on 4 cores. The time for 1 core was given as 38.47 seconds, while the time for 4 cores was given at 18.29 seconds. This difference is about a 47.57 percent decrease in runtime which is better than 1 core, but it is not yet near the ideal 75 percent. Another interesting note is that there was also a large amount of garbage collection occurring during each run.

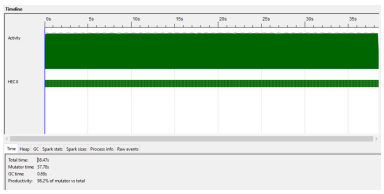

Figure 10: Board 1: 1 Core, Depth of 4

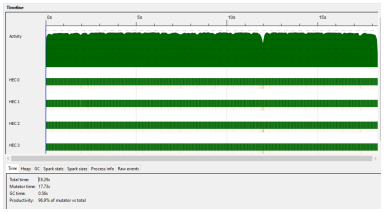

Figure 11: Board 1: 4 Cores, Depth of 4

7 Further Results and Analysis

In addition to the threadscope comparison, I analyzed each of 3 boards on 1 core versus 4 cores in addition to testing at depths of 2 and depths of 4. Please note that the default depth is 4, but this variable is labeled in my code and can be manually adjusted. I would like to incorporate the depth as a command line argument to test rather than needing to update it manually and recompile accordingly.

7.1 Testing at Different Depths

The depths I chose to focus on were 2 and 4. I tested board 1, board 2, and board 3 at a depth of 2 and at a depth of 4 on 4 cores. I also tested on 2 cores, but for the purpose of the analysis, the below results were run on 4 cores.

Total Run Time	Depth of 2	Depth of 4
Board 1	0.82 seconds	18.29 seconds
Board 2	0.578 seconds	6.84 seconds
Board 3	0.984 seconds	36.65 seconds

Figure 12: Runtime: Depth Comparison

7.2 Testing with a Different Number of Cores

I also compared the runtime of board 1, board 2, and board 3 on 1 core versus 4 cores. The first two columns (With 1 Core and With 4 Cores) give the total runtime in seconds on each board. The last column, difference, give the total difference in time in seconds. While the total time difference is important to note, I think it is more relevant to look at the percentage change which is why it is included.

I believe the ideal percentage decrease in time from 1 to 4 cores would be about 75 percent, my results ranged from 39.21 percent to 47.54 percent. A few interesting notes is that despite board 3 starting midway through the game,

it takes about 90 seconds on 1 core and 40 seconds on 4 cores, while board 1 only takes about 40 seconds on 1 core and 20 on 4 cores. Despite the large time difference in running board 3, the percentage change demonstrated that the runtime decreased by about 40 percent which tracks with the 40 percent decrease in runtime for board 2 and the 47 percent decrease in runtime for board 1.

Total Run Time (seconds)	With 1 Core	With 4 Cores	Difference
Board 1	38.47 seconds	18.29 seconds	20.17 s (47.54%)
Board 2	17.24 seconds	6.84 seconds	10.4 s (39.68%)
Board 3	93.46 seconds	36.65 seconds	56.81 s (39.21%)

Figure 13: Runtime: Core Comparison

8 Going Forward

Going forward, I would like to improve upon the current implementation of Othello on different boards and potentially expand it.

8.1 Alpha Beta Pruning

Similar to my implementation of minimax with alpha beta pruning in Python, it would be interesting to explore further implementing this strategy for the Othello game.

8.2 User Interaction

I would like to keep the option to run Othello on different board states. A potential expansion on that could be to allow the user to create their own initial board state rather than the given one.

Adding user interaction such that instead of selecting a board, the user would be able to make their own moves and play through the full Othello game would be interesting. From what I currently have, I do not think it is too large a leap to make and would add user further user engagement.

I currently have the depth of the search tree set to 4 and had tested it at a depth of 2 by manually changing this variable in my code, recompiling, and testing. I think adding the depth set variable to be a potential command line argument for testing purposes would work better than having to manually adjust and recompile just to experiment with the program itself.

8.3 Parallelization

My current parallelization implementation only decreases the runtime from about 40 to 50 percent total, which is not near the ideal optimization yet. I would like to explore further parts of the program that I can optimize as well as experiment with different parallelization strategies that might prove to be more effective.

8.4 Heuristics

I would also like to explore implementing more heuristics which might help the accuracy and efficiency of the game. For instance, in an actual game of Othello, if a player has the opportunity to take a corner piece even at the cost of flipping over more of their opponents' pieces, the player will make that decision to take the corner piece. The reasoning being that the corner spot is completely protected once there. Additionally, ensuring not to move in a spot if it would give your opponent the opportunity to take the corner piece would be an interesting heuristic to explore. This idea refers back to the regions of the board.

8.5 Troubleshooting

As the program is not near the optimal percentage decrease in runtime yet, I would like to explore how to further optimize the program as well as explore parallelization strategies. For instance, the starting board state of board 3 takes significantly longer than board 1 and board 2, even though board 1 starts with less pieces and board 2 is at about the same way through the game as board 3. Exploring further how to minimize the runtime in this particular case would be a solid step forward

9 References

References can also be seen here:
https://docs.google.com/document/d/1KhxJaxueMmcWSAnAHaaKYi5vADCF26gKcKw655Q4KOs/edit?usp=sharing

1. https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/ch03.html
2. https://github.com/krispo/awesome-haskell
3. https://simonmar.github.io/pages/pcph.html
4. https://www.macs.hw.ac.uk/~dsg/gph/
5. https://www.reddit.com/r/haskell/comments/sdx5v/examples_of_easy_parallelism_in_haskell/
6. http://community.haskell.org/~simonmar/par-tutorial.pdf
7. https://wiki.haskell.org/Parallel
8. https://wiki.haskell.org/Parallel_GHC_Project
9. https://github.com/apoorvingle/h-reversi/blob/master/src/Main.hs
10. https://wiki.ubc.ca/Course:CPSC312-2017-Othello_Haskell
11. https://www.reddit.com/r/haskellquestions/comments/4z92r8/take_a_look_at_my_reversi_implementation/
12. https://github.com/sunjay/reversi/blob/master/src/Reversi.hs
13. https://arttuys.fi/texts/haskell/haskellversi/
14. http://giocc.com/concise-implementation-of-minimax-through-higher-order-functions.html
15. https://github.com/geon/Othello-in-Haskell/blob/master/othello.hs
16. https://github.com/12yuens2/haskell-othello-reversi/blob/master/code/src/AI.hs
17. http://www.pressibus.org/ataxx/autre/minimax/node2.html#SECTION00020000000000000000
18. http://www.pressibus.org/ataxx/autre/minimax/paper.html
19. http://kremer.cpsc.ucalgary.ca/courses/cpsc449/W2015x/assnHaskell.html
20. https://github.com/dcastro/twenty48/blob/master/src/Game/Minimax.hs
21. http://hackage.haskell.org/package/hstzaar-0.9.4/src/src/AI/Minimax.hs
22. https://www.reddit.com/r/haskell/comments/h042g0/help_creating_a_minimax_function_for_an_ai_for/
23. https://dev.to/nt591/writing-a-tictactoe-game-in-haskell-545e
24. https://gist.github.com/adwhit/db18f3fbfdb61bc56357bce25aa2d19d
25. https://woorks.co.jp/r2jyp/319e4f-othello-minimax-algorithm-python
26. https://stackoverflow.com/questions/41916409/profiling-with-threadscope-with-command-line-arguments-in-haskell
27. https://gregorias.github.io/2014/12/28/creating-ai-in-haskell.html
28. https://stackoverflow.com/questions/21966293/exception-prelude-head-empty-list-in-haskell
29. https://stackoverflow.com/questions/8852220/how-to-use-parallel-strategies-in-haskell

10 Full Code Listing

```
import Data.List
import Data.Maybe
import qualified Data.Map as Map
import Control.Parallel.Strategies(using, parList, rseq)
import System.Environment as System

data Othello = White | Black | Empty deriving (Eq, Show)
type P = (Int, Int)
type Board = Map.Map P Othello

--tested at depths of 2 and 4
--can adjust the depth used here manually:
depth_set = 4

colorToString :: Othello -> String
colorToString piece =
    case piece of
        Empty -> "_"
        White -> "w"
        Black -> "b"

setBoard :: Board -> String
setBoard b = "\n 0 1 2 3 4 5 6 7 \n" ++ (intercalate "\n"
(map (setRow b)[0..7])) ++ "\n 0 1 2 3 4 5 6 7 \n \n"
    where
        setRow b r = show r ++ " " ++ (intercalate " " (map (\p ->
            colorToString (fromMaybe Empty (Map.lookup p b)))
            ([ (r,x) | x <- [0..7]]))) ++ " " ++ show r

changeColor :: Othello -> Othello
changeColor White = Black
changeColor Black = White
changeColor _ = Empty

moves :: Othello -> Board -> [P]
moves col b = filter (isValid col b) possibilities
        where
            possibilities = [(x, y) | x <- [0..7], y <- [0..7]]

isValid :: Othello -> Board -> P -> Bool
isValid col b p = flipCell col b p /= [] && isNothing (Map.lookup p b)

move :: Othello -> Board -> P -> Board
```

```
move col b p = Map.union helper b
  where
      helper = (Map.fromList (zip (flipCell col b p) (repeat col)))

adv :: Othello -> Board -> Int
adv col b = sum ( map (\(_, x) -> pt x)  (Map.toList b))
  where
    pt x
      | x == col = 1
      | x == Empty = 0
      | otherwise = -1

dir = [(0, 1), (1, 0), (1, 1), (0, -1), (-1, 0),
(-1, -1), (-1, 1), (1, -1)]

flipCell :: Othello -> Board -> P -> [P]
flipCell col b p
    | null flip = []
    | otherwise = p : flip
    where
        flip = concat(map(flipCellHelper True col b p) dir)

flipCellHelper :: Bool -> Othello -> Board -> P -> P -> [P]
flipCellHelper i col b p d
    | nxtCol == changeColor col = if r /= [] then
        if not i then
            p : r
        else r
    else []
    | nxtCol == col = if not i then [p]
    else []
    | otherwise = []
    where
        nxtP = ( \(x1,y1)(x2,y2) -> (x1+x2, y1+y2) ) p d
        nxtCol = fromMaybe Empty (Map.lookup nxtP b)
        r = flipCellHelper False col b nxtP d

endGame :: Othello -> Int -> IO ()
endGame col adv
    | adv == 0 = putStr "Tie!\n"
    | adv > 0 = putStr (colorToString col ++ " Won!\n")
    | otherwise = putStr (colorToString (changeColor col) ++ " Won!\n")


minimax :: Int -> Othello -> Board -> Int
minimax dpth col b
```

```
    | endGame = if (adv col b) > 0
        then 100000
        else -100000
    | dpth <= 0 = adv col b
    | otherwise = if ( moves (changeColor col) b ) /= []
        then -maxPt
        else maxPt
    where
        endGame = null (moves col b) && null (moves (changeColor col) b)
        clrUp = if ( moves (changeColor col) b ) /= []
            then changeColor col
            else col
        nm = if clrUp /= col
            then ( moves (changeColor col) b )
            else ( moves col b)
        maxPt = maximum (map (minimax (dpth - 1) clrUp . move clrUp b)
        nm `using` parList rseq)

--computerMove is where depth is set: default is 4
play :: Othello -> Board -> IO ()
play col b =
    if null (moves col b) && null (moves (changeColor col) b)
        then endGame col $ adv col b
        else do
            let
                brd = move col b (computerMove col b)
                clrUp = if moves (changeColor col) brd /= []
                    then (changeColor col)
                    else col
            putStr (setBoard b)
            play clrUp brd
            where computerMove col b = fst $ maximumBy (\(_, x) (_, y) ->
            compare x y)(map (\p -> (p, minimax depth_set col
            (move col b p)))(moves col b))

--board1 is the default start state: 4 pieces in the center, 2 white, 2 black
board1 = Map.fromList [((3, 3), White), ((4, 4), White),
((3, 4), Black), ((4, 3), Black)]

--board2 is a potential mid-game scenario which is black-piece dominated
--having 12 black pieces and 6 white pieces
board2 = Map.fromList [((2,2), Black), ((2,3), Black), ((2,4), Black),
((3,2), Black), ((3,3), Black), ((3,4), Black), ((4,1), Black),
((4,3), Black), ((4,4), Black), ((4,6), Black), ((6,3), Black),
((3,1), White), ((4,2), White), ((4,5), White), ((5,3), White),
((5,6), White),((6,4), White) ]
```

```
--board3 is a potential mid-game scenario, largely white-piece dominant
--totaling 11 white pieces, and 7 black pieces
board3 = Map.fromList [((1,2), Black), ((2,2), Black), ((2,3), Black),
((3,4), Black), ((4,2), Black), ((4,5), Black), ((5,5), Black),
((1,3), White), ((2,4), White), ((2,5), White), ((2,6), White),
((3,2), White), ((3,3), White), ((3,5), White), ((3,6), White),
((4,3), White), ((4,4), White), ((5,3), White)]

main :: IO ()
main = do
    args <- System.getArgs
    if length args == 0
        then play White board1
        else do
            if head args == "board1"
                then play White board1
                else do
                    if head args == "board2"
                        then play White board2
                        else do
                            if head args == "board3"
                                then play White board3
                                else do putStr "invalid usage"
```