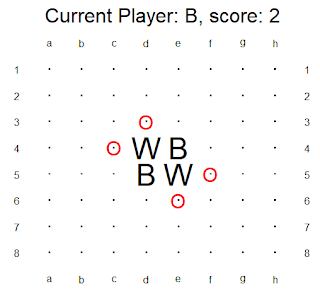
In this post, I showcase a bare-bones point-and-click implementation of the classic board Reversi (also called Othello\*) in the R programming language. R is typically used for more serious, statistical endeavors, but it works reasonably well for more playful projects. Building a classic game like this is an excellent high-school level introduction to programming, as well as a good basis for building and testing game AI.

If you want to skip ahead and just play Reversi in R right away, download this .R file:

Reversi Functions.r which is attached with the repository.

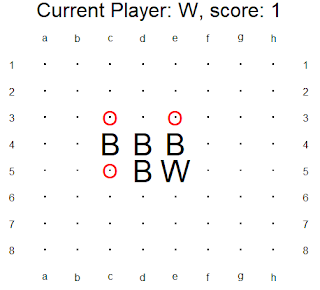
open it either base R or Rstudio, set the working directory to the location where you downloaded to, and run the file with source(“Reversi Functions.r”) . This will print some basic play instructions for you.

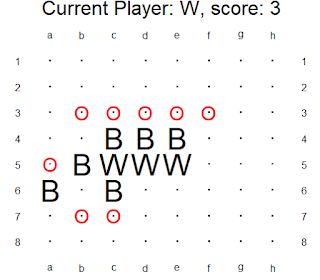
The starting configuration shown here is just one of many possible configurations that the code can handle. This program can allow for many variants in play including board dimensions, ‘walls’ that block capture, and three or more players. These will be shown in a later post.

[](https://i1.wp.com/1.bp.blogspot.com/-Tz2PPpu3oB4/XTSxGHTrMHI/AAAAAAAAApk/6QmpIER5Ruk39H8XLv9EUGlqHBIxkb1GgCLcBGAs/s1600/Reversi_Demo_1.PNG?ssl=1)

When running, the active player can click on any of the legal spaces, marked with red circles, to place a stone and capture any enemy pieces that are sandwiched between the newly placed piece and already-placed pieces.

The board after the first couple moves by “B”lack and “W”hite are shown in the next two figures.

[](https://i0.wp.com/1.bp.blogspot.com/-_KWZCv5YyhM/XTSxGFL5_VI/AAAAAAAAAp4/0hbjcvS9j3UEhIjnqtdKOQEXuft0GnmKgCEwYBhgL/s1600/Reversi_Demo_2.PNG?ssl=1)

[](https://i0.wp.com/1.bp.blogspot.com/-9XeE0XiLKSk/XTSxGBlK5qI/AAAAAAAAAp8/7XInNBEt8FwJjJclOMAUILwdnzjAkUnPACEwYBhgL/s1600/Reversi_Demo_3.PNG?ssl=1)

The game continues until neither player has a legal move, which usually, but not always, occurs when the board is filled or when there is only one player’s pieces remaining.

Before we look at the user-defined functions for this game, let’s review a few important functions in base R.

paste(x, collapse=””)  # Takes a vector of string variables and converts it into a single string with nothing between each of the original strings.

strsplit(x,””)[[1]] # Takes a single string and converts it into a vector of string variables of one character each.

plot.new(); plot.window(xlim=.. ylim=) # Clears the existing plot, if any, and creates a new one with the given x and y limits.

locator(1) # Takes the first mouse click on a plot and extracts the x and y coordinates on the plot.

The pasteand strsplit functions are useful for passing information about the stones to each other in a convenient way. String manipulation is slow, so it’s not the most computationally efficient way to handle this, but it’s also not the biggest task. If I had to run thousands of games quickly, say, for machine learning purposes, I would have used a more sophisticated method here.

The locator function is what makes it possible to play this game with a mouse instead of a keyboard, which makes the whole thing much more enjoyable to play and test.

The game instead is broken down into as many user-defined functions as is convenient. In other words, the programming is *modular*, it’s broken down into many small modules. Modularity makes it easier to modify the program and prevents errors that may occur from having to change the same code in multiple locations. Furthermore, with good function names, the whole process is easier to read by a human.

We start with setup.board, which establishes the board dimensions, starting pieces, number of players, and any walls or gaps. For this demo, let’s stick with the classic Othello setup.

setup.board = function(style = “Basic Othello”)

{

     if(style == “Basic Othello”)

     {

           board = matrix(“.”,nrow=8,ncol=8)

           board[4,4] = “W”

           board[5,5] = “W”

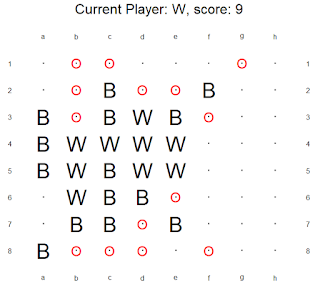
           board[4,5] = “B”

           board[5,4] = “B”

     }

}

A piece can be placed anywhere that a ‘sandwich’ can be made. To determine this, we first need a function to ‘look’ in a given direction from a given position on a board. For example, looking from b1 in the southeast direction on this board…

[](https://i0.wp.com/1.bp.blogspot.com/-Ps2dHik1r5k/XTSxG60dMOI/AAAAAAAAAqA/vtAuBn80hYA81x5Frs8cdG801F5ZxaGUACEwYBhgL/s1600/Reversi_Demo_4.PNG?ssl=1)

…should produce a ‘look’ of ”B,W,W,space,space,space”. The look.to function takes in a board, position, and direction, and returns the state of the board spaces from that location outwards until it hits the edge of the board. The direction dictates the values of xstep and ystep, which in turn dictate which spaces are the board are examined. the order The look.around function calls look.to from a given position for all eight directions, one at a time.

look.to = function(board, position, direction)

{

…

if(direction == “N”){  xstep = 0;   ystep = -1}

if(direction == “NE”){ xstep = 1;   ystep = -1}

…

if(direction == “W”){  xstep = -1;  ystep = 0}

if(direction == “NW”){ xstep = -1;  ystep = -1}

## Do this until we look to the edge of the board

      while(this\_x > 0 & this\_x <= ncol(board) &

this\_y > 0 & this\_y <= nrow(board))

      {

### Record the stone (or space) and xy coords at the observed location

            stones = c(stones, board[this\_y,this\_x])

            xlist = c(xlist, this\_x)

            ylist = c(ylist, this\_y)

### Iterate the observed location based on the selected direction

            this\_x = this\_x + xstep

            this\_y = this\_y + ystep

      }

…

}

The legal.look function takes a given sequence of stones from the look.to function, and determines if a given player (a single character of a string) can make a sandwich in that direction. It returns the *number* of sandwiched enemy stones. Note that the code considers any character other a blank space ( . ) or a wall ( # ) to be an enemy stone, allowing this code to work with 3+ players.

The legal.directions function calls legal.look for each direction and returns the list of directions in which a capture can be made by the given player by placing at the given position. The which.legal function calls legal.direction for each empty position on the board to determine which, if any, spaces a given player may place on the board.

legal.look = function(player, look)

{

…

Nenemies = 0

enemy\_chain = TRUE

while( Nenemies < length(look) & enemy\_chain)

{

### Examine a space, if it’s anything except

### the current player’s piece, a space, or a wall, it’s an enemy piece

      examined\_piece = look[Nenemies + 1]

      if(examined\_piece %in% c(player,”.”,” “,”#”))

      { ## If it’s not an enemy, stop looking

            enemy\_chain = FALSE

      }

      else

      {   ## If it is an enemy, iterate and keep looking

            Nenemies = Nenemies + 1

      }

}

### If there are enemy pieces all the way to the end of the board

### Return ‘no capture’.

if(Nenemies == length(look)){return(0)}

## There must be an allied piece at immediately after the enemy pieces

## If so, return the number of pieces that can be captured

examined\_piece = look[Nenemies + 1]

if(examined\_piece == player)

{

      return(Nenemies)

}

### Otherwise return ‘no capture’

return(0)

}

The plot.game function takes a board state, active player, and possibly the matrix of legal moves, and draws this information as a plot.

plot.game = function(board,player,legal\_board=NULL,showlegal=TRUE)

The play.move function checks if a move at the position is legal by a given player on the given board. If it is, it updates the board by placing a stone for player at position and makes all the appropriate captures. It returns the new board state. It uses look.to and legal.look to determine which stones on the board to change.

play.move = function(board, this\_player, position)

The play game is the main function that that runs the whole game.

play.game = function(board=NA)

It takes in mouse clicks with locator and converts them into positions on the board (note the inversion of the y-axis).

mouseclick = locator(1)

input\_x = round(mouseclick$x)

input\_y = round(mouseclick$y)

input\_y = nrow(board) – input\_y + 1

Before using that click, it first checks if that it maps to a space on the board, and that the space is a legal move by the given player. (If there is no legal move, it will accept any click as a ‘pass’, and move to the next player)

while(all(board == new\_board) & any(legal\_board == TRUE))

if(input\_x > 0 & input\_x <= Nx & input\_y > 0 & input\_y <= Ny)

legal\_board = which.legal(board, current\_player)

if(legal\_board[input\_y,input\_x])

If the move is legal, it calls play.move to update the board and cycles to the next player.

new\_board = play.move(board,this\_player=current\_player,position=c(input\_y,input\_x))

current\_player = player\_list[ 1 + (player\_idx %% Nplayers)]

It continues to do this until no player has a legal move, or there are no spaces left on the board. After which it returns the board state as a matrix as well as the score of each player.

while(  any(board == “.”) & players\_skipped < length(player\_list))

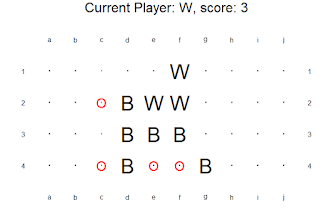
print(table(board))

return(board)

Finally, you can use setup.board to create a non-standard board and use it in play.game. For example, a 4×10 board can be created and used with the “Othello Wide” style.

board = setup.board(“Othello Wide”)

play.game(board)

[](https://i1.wp.com/1.bp.blogspot.com/-ErJeWQCMjv0/XTSxHE1PASI/AAAAAAAAAqE/MLLzYqtSlnYck9bJbqBEKPzmi7R6ofiHACEwYBhgL/s1600/Reversi_Demo_6.PNG?ssl=1)

\* There are slight differences between the commercial version Othello, and the public domain game Reversi. Also, the name Othello is trademarked by Mattel.