Simulating Tic-Tic-Toe

Alex Stephenson

Introduction

This code sample is a simulation to have a computer play tic-tac-toc randomly against itself on different board sizes to see what, if any, advantage a player gets by moving first.

```r
set.seed(8675309)

# Libraries
library(ggplot2)
library(tidyr)
```

Functions

```r
diagSums = function(sums=NULL, diagonals){
    # A function to get all sums of each possible diagonal of a matrix
    # Important for when win condition is less than the size of a square diagonal
    # Example a 4x4 board with a win condition of 3 not having to touch
    #@params: diagonals = a list of diagonals
    #@output: sums = a vector of sums
    for(i in 1:length(diagonals)){
        sums[i] = sum(diagonals[[i]])
    }
    return(sums)
}

isWinner = function(board, win){
    # A function to determine whether a player has won the game given a board
    # position
    #@params: board = a vector representation of the board state
    #@params: win = The win condition of the board
    #@output c(0,1,2) corresponding to no win, player 1 win, player 2 win

    # Turn the vector representation of the board into a matrix
    mat = t(matrix(board, ncol = sqrt(length(board))))

    rows = rowSums(mat)
    cols = colSums(mat)

    # Create diagonal representations
    # Create an indicator for each cell's diagonal
    which_1 = row(mat) - col(mat)

    # Split the diagonals based on the indicators
    # This will get all the diagonals of a board into a  list
    diags = split(mat, which_1)

    # Store first set of diagonal sums
```

```r
    diagSumsL2R = diagSums(diagonals = diags)

    # Reverse the same procedure to get the bottom to top diagonals
    mat2 = t(apply(mat, 2, rev))
    which_2 = row(mat2) - col(mat2)
    diags2 = split(mat2, which_2)
    diagSumsR2L = diagSums(diagonals = diags2)

    # Determine if any rows, columns, or diagonals sum to win condition for
    # either player. If any do, end game and return winning player.
    # Otherwise return 0.
    if(win %in% c(rows, cols, diagSumsL2R, diagSumsR2L)){
        return(1)
    }
    if(-win %in% c(rows, cols, diagSumsL2R, diagSumsR2L)){
        return(2)
    }
    return(0)
}

# Our next function is the actual gameplay.
# According to the specifications, the function should only take the
# board size as a parameter
playGame = function(board, win){
    # A function to simulation gameplay between two computer players
    #@params: board = a vector representation of a tic-tac-toe board
    #@params: win = The number of squares necessary to win. Squares do not
        # necessarily have to be connected

    # Set winner to 0 indicating no one has won at the start
    winner = 0

    # By specification player 1 always goes first
    player = 1

    # Set the number of moves at the beginning of the game to 0
    numberOfMoves = 0

    # Play game until there is a winner or there are no free spaces left
    while (0 %in% board & winner == 0) {

        # Find all of the empty squares
        empty = which(board == 0)

        # Get a random move by sampling 1 from the list of empty squares
        move = empty[sample(length(empty), 1)]

        # Update the board and number of moves with the player's move
        board[move] = player
        numberOfMoves = numberOfMoves + 1

        # Check to see if move is winning
        winner = isWinner(board, win)
```

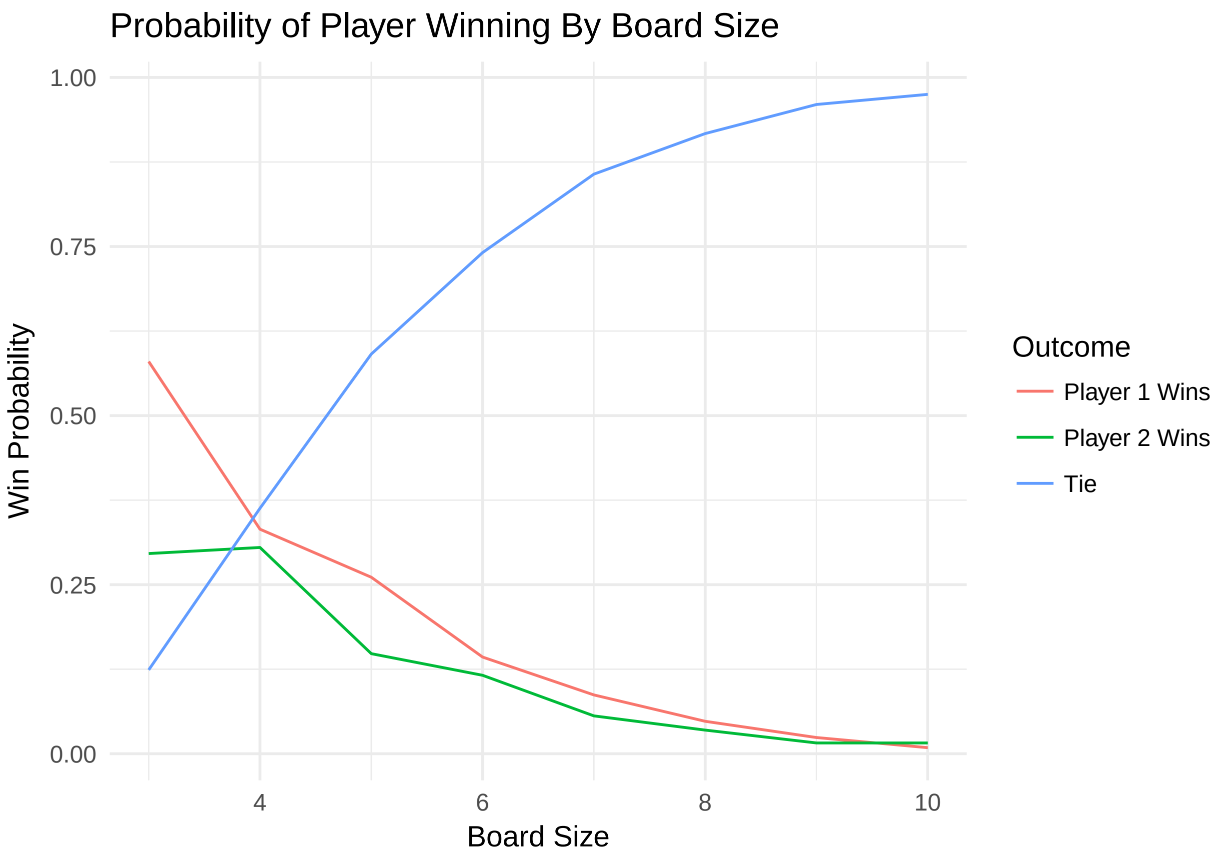
```r
        # If winner, break out of the loop
        if(winner != 0){
            break
        }
        # Otherwise update the player and continue
        player = player * -1
    }
    return(c(winner, numberOfMoves))
}

makeBoard = function(integer){
    # A function to make a tic tac toe board
    #@params: integer = a whole positive number representing the board size
    return(rep(0, integer^2))
}

# function to simulate a game of tic tac toe of arbitrary board size
# for an arbitrary number of simulations.
simulate_games = function(SIM_LENGTH, SIZE){
    # A function to simulate tic tac toe game
    #@params: SIM_LENGTH = number of games
    #@params: SIZE = board size

    WINNER = numeric(SIM_LENGTH)
    MOVES = numeric(SIM_LENGTH)
    for(i in 1:SIM_LENGTH){
        board = makeBoard(SIZE)
        game = playGame(board, win = SIZE)
        WINNER[i] = game[1]
        MOVES[i] = game[2]
    }

    # Get summary statistics for simulations
    player_1_WP = sum(WINNER == 1) / SIM_LENGTH
    player_2_WP = sum(WINNER == 2) / SIM_LENGTH
    tie_prob = sum(WINNER == 0)/ SIM_LENGTH
    avg_moves = mean(MOVES)
    return(list(player_1_WP, player_2_WP, tie_prob, avg_moves))
}
```

Simulations

The simulation plays 1000 games between two players moving at random for board sizes from 3x3 to 10x10. In all simulations, player 1 is fixed to move first. In this first simulation, a player wins when they get a connecting row, column, or diagonal of the board size. For example, it takes 3 in a row to win on a 3x3 board size.

```r
prob = matrix(NA, nrow = 8, ncol= 5)
for(i in 1:8){
    result = simulate_games(1000, i+2)
    prob[i,1] = i + 2
    prob[i,2] = result[[1]]
```

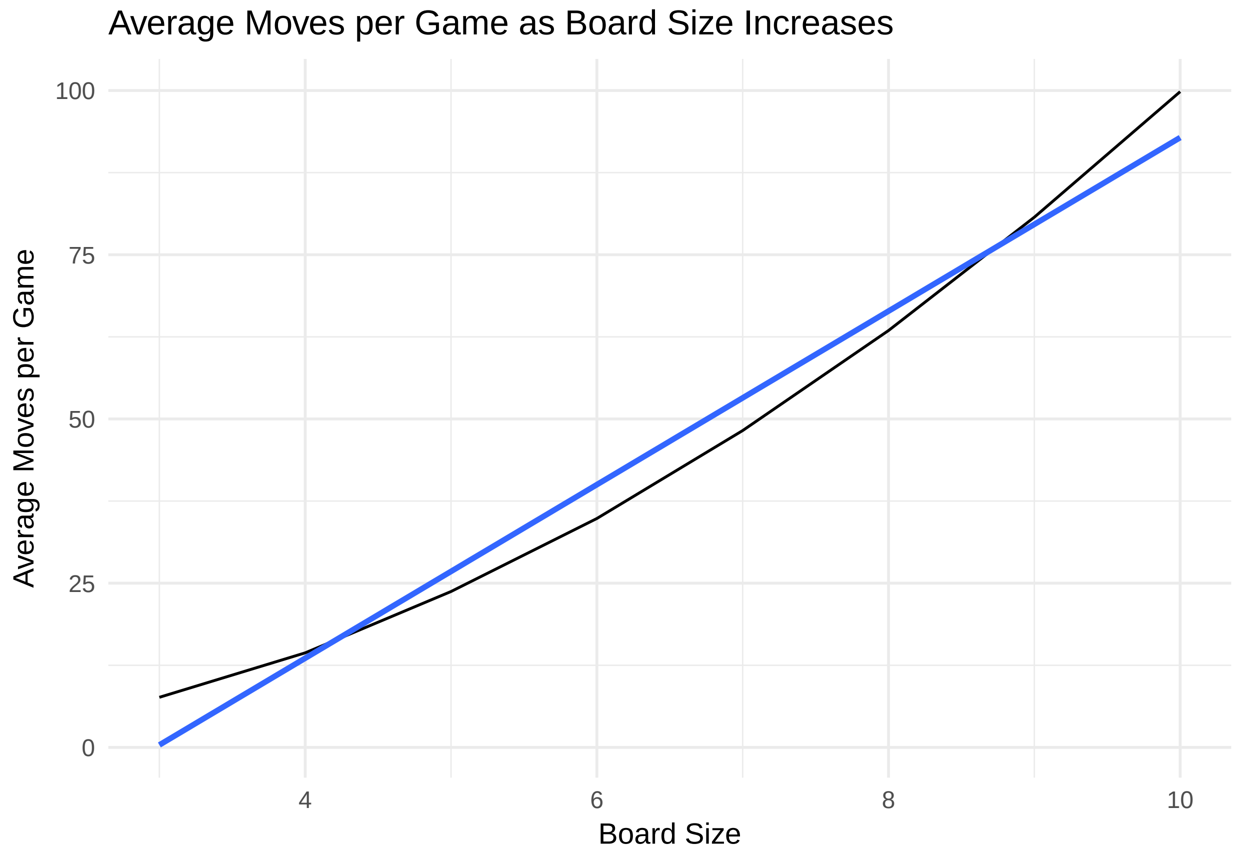
```
    prob[i,3] = result[[2]]
    prob[i,4] = result[[3]]
    prob[i,5] = result[[4]]

}

# Coerce to data frame
prob = as.data.frame(prob)
names(prob) = c("size","p1", "p2", "tie", "avg_moves")

# Plot results
prob %>%
    gather(result, prob, -size,-avg_moves) %>%
    ggplot(., aes(x = size, y = prob, group=result, colour = result))+
    geom_line()+
    xlab("Board Size")+
    ylab("Win Probability")+
    ggtitle("Probability of Player Winning By Board Size")+
    scale_color_discrete(name="Outcome",
                         labels=c("Player 1 Wins", "Player 2 Wins", "Tie"))+
    theme_minimal()
```



Clearly, as board size increases the likelihood of a win drops dramatically for long connected win conditions.

In addition, we might also be interested about whether the length of games is a constant multiplier as board size increases. We plot the results.

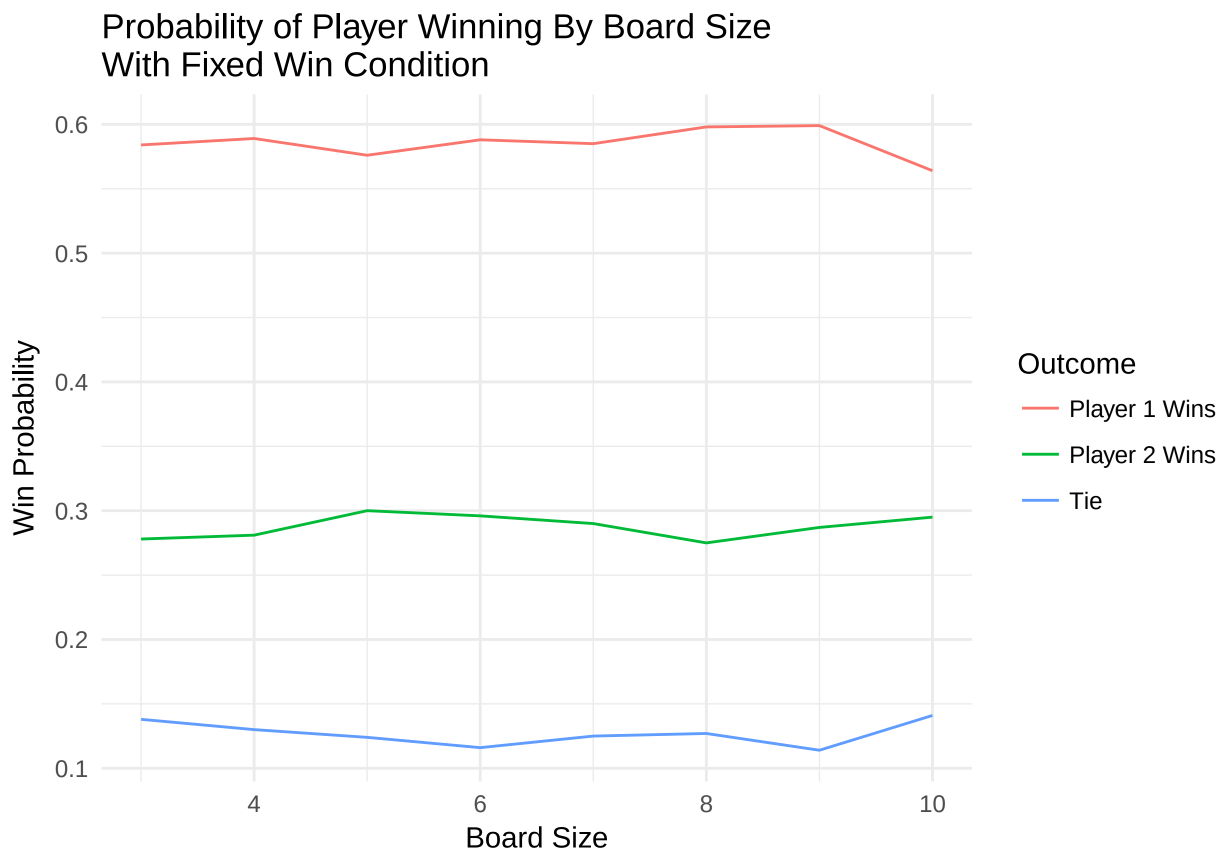
```r
prob %>%
    ggplot(., aes(x = size, y = avg_moves))+
    geom_line()+
    geom_smooth(method ="lm",se = F)+
    xlab("Board Size")+
    ylab("Average Moves per Game")+
    ggtitle("Average Moves per Game as Board Size Increases")+
    theme_minimal()
```



Now we relax the win condition so that instead of having to get a full connected row, a player wins if they can some subset of the row. In this case, what is the advantage of moving first if you only have to get three in a row, regardless of board size? We plot the results again

```r
prob2 = matrix(NA, nrow = 8, ncol= 5)
for(i in 1:8){
    result = simulate_games(1000, 3)
    prob2[i,1] = i + 2
    prob2[i,2] = result[[1]]
    prob2[i,3] = result[[2]]
    prob2[i,4] = result[[3]]
    prob2[i,5] = result[[4]]

}

# Coerce to data frame
prob2 = as.data.frame(prob2)
names(prob2) = c("size","p1", "p2", "tie", "avg_moves")
```

```r
# Plot results
prob2 %>%
    gather(result, prob, -size,-avg_moves) %>%
    ggplot(., aes(x = size, y = prob, group=result, colour = result))+
    geom_line()+
    xlab("Board Size")+
    ylab("Win Probability")+
    ggtitle("Probability of Player Winning By Board Size\nWith Fixed Win Condition")+
    scale_color_discrete(name="Outcome",
                        labels=c("Player 1 Wins", "Player 2 Wins", "Tie"))+
    theme_minimal()
```



Clearly, there is an enormous advantage to going first as the win condition requirement decreases.