Tic Tac Toe War Games: The Intelligent Minimax Algorithm

Posted on **6 April 2017**

 Written by: Peter Prevos on 6 April 2017.

In a previous post, I shared how to build a randomised Tic Tac Toe simulation. The computer plays against itself playing at random positions. In this post, I will share how to teach the computer to play the game strategically.

I love the 1983 classic movie War Games. In this film, a computer plays Tic Tac Toe against itself to learn that it cannot win the game to prevent a nuclear war.

Back in those days, I devoured the wonderful book *Writing Strategy Games on your Atari* by John White which contains an algorithm to play Tic Tac Toe War Games. This is my attempt to relive the eighties using R.

You can find the code on my GitHub page.

Drawing the Board

A previous post describes the function that draws the Tic Tac Toe board. For completeness, the code is replicated below. The game board is a vector of length nine consisting of either -1 (X), 0 (empty field) or 1 (O). The vector indices correspond with locations on the game board:

1 2 3
4 5 6
7 8 9

```
draw.board <- function(board) { # Draw the board
    xo <- c("X", " ", "O") # Symbols
    par(mar = rep(0,4))
    plot.new()
    plot.window(xlim = c(0,30), ylim = c(0,30))
    abline(h = c(10, 20), col="darkgrey", lwd = 4)
    abline(v = c(10, 20), col="darkgrey", lwd = 4)
    pieces <- xo[board + 2]
    text(rep(c(5, 15, 25), 3), c(rep(25, 3), rep(15,3), rep(5, 3)), pi
```

```
    # Identify location of any three in a row
    square <- t(matrix(board, nrow = 3))
    hor <- abs(rowSums(square))
    if (any(hor == 3))
      hor <- (4 - which(hor == 3)) * 10 - 5
    else
      hor <- 0
    ver <- abs(colSums(square))
    if (any(ver == 3))
      ver <- which(ver == 3) * 10 - 5
    else
      ver <- 0
    diag1 <- sum(diag(square))
    diag2 <- sum(diag(t(apply(square, 2, rev)))) # Draw winning lines
    if (ver > 0) lines(rep(ver, 2), c(0, 30), lwd=10, col="red")
    if (abs(diag1) == 3) lines(c(2, 28), c(28, 2), lwd=10, col="red")
    if (abs(diag2) == 3) lines(c(2, 28), c(2, 28), lwd=10, col="red")
}
```

Human Players

This second code snippet lets a human player move by clicking anywhere on the graphic display using the locator function. The click location is converted to a number to denote the position on the board. The entered field is only accepted if it has not yet been used (the *empty* variable contains the available fields).

```
# Human player enters a move
move.human <- function(game) {
    text(4, 0, "Click on screen to move", col = "grey", cex=.7)
    empty <- which(game == 0)
    move <- 0
    while (!move %in% empty) {
        coords <- locator(n = 1) # add lines
        coords$x <- floor(abs(coords$x) / 10) + 1
        coords$y <- floor(abs(coords$y) / 10) + 1
        move <- coords$x + 3 * (3 - coords$y)
    }
    return (move)
}
```

Evaluate the Game

This code snippet defines the *eval.game* function which assesses the current board and assigns a score. Zero means no outcome, -6 means that the *X* player has won and +6 implies that the *O* player has won.

```
# Evaluate board position
eval.game <- function(game, player) {
    # Determine game score
    square <- t(matrix(game, nrow = 3))
    hor <- rowSums(square)
    ver <- colSums(square)
    diag1 <- sum(diag(square))
    diag2 <- sum(diag(t(apply(square, 2, rev))))
    eval <- c(hor, ver, diag1, diag2)
    # Determine best score
    minimax <- ifelse(player == -1, "min", "max")
    best.score <- do.call(minimax, list(eval))
    if (abs(best.score) == 3) best.score <- best.score * 2
    return (best.score)
}
```

Computer Moves

The computer uses a modified Minimax Algorithm to determine its next move. This article from the Never Stop Building blog and the video below explain this method in great detail.

Artificial Intelligence in Tic-Tac-Toe (bonus class)

The next function determines the computer's move. I have not used a brute-force minimax algorithm to save running time. I struggled building a fully recursive minimax function. Perhaps somebody can help me with this. This code looks only two steps deep and contains a strategic rule to maximise the score.

The first line stores the value of the players move, the second remainder of the matrix holds the evaluations of all the opponents moves. The code adds a randomised variable, based on the

strategic value of a field. The centre has the highest value because it is part of four winning lines. Corners have three winning lines and the rest only two winning lines. This means that the computer will, all things being equal, favour the centre over the corners and favour the other fields least. The randomised variables in the code ensure that the computer does not always pick the same field in a similar situation.

```
# Determine computer move
move.computer <- function(game, player) {
    empty <- which(game == 0)
    eval <- matrix(nrow = 10, ncol = 9, data = 0)
    for (i in empty) {
        game.tmp <- game
        game.tmp[i] <- player
        eval[1, i] <- eval.game(game.tmp, player)
        empty.tmp <- which(game.tmp ==0)
        for (j in empty.tmp) {
            game.tmp1 <- game.tmp
            game.tmp1[j] <- -player
            eval[(j + 1), i] <- eval.game(game.tmp1, -player)
        }
    }
    if (!any(abs(eval[1,]) == 6)) { # When winning, play move
        # Analyse opponent move
        minimax <- ifelse(player == -1, "max", "min") # Minimax
        best.opponent <- apply(eval[-1,], 1, minimax)
        eval[1,] <- eval[1,] * -player * best.opponent
    }
    # Add randomisation and strategic values
    board <- c(3, 2, 3, 2, 4, 2, 3, 2, 3) # Strategic values
    board <- sapply(board, function(x) runif(1, 0.1 * x, (0.1 * x) + 0
    eval[1, empty] <- eval[1, empty] + player * board[empty] # Randomi
    # Pick best game
    minimax <- ifelse(player == -1, "which.min", "which.max") # Minima
    move <- do.call(minimax, list(eval[1,])) # Select best move
    return(move)
}
```

This last code snippet enables computers and humans play each other or themselves. The *players* vector contains the identity of the two players so that a human can play a computer or vice versa. The human player moves by clicking on the screen.

The loop keeps running until the board is full or a winner has been identified. A previous Tic Tac Toe post explains the *draw.board* function.

```
# Main game engine
tic.tac.toe <- function(player1 = "human", player2 = "computer") {
    game <- rep(0, 9) # Empty board
```

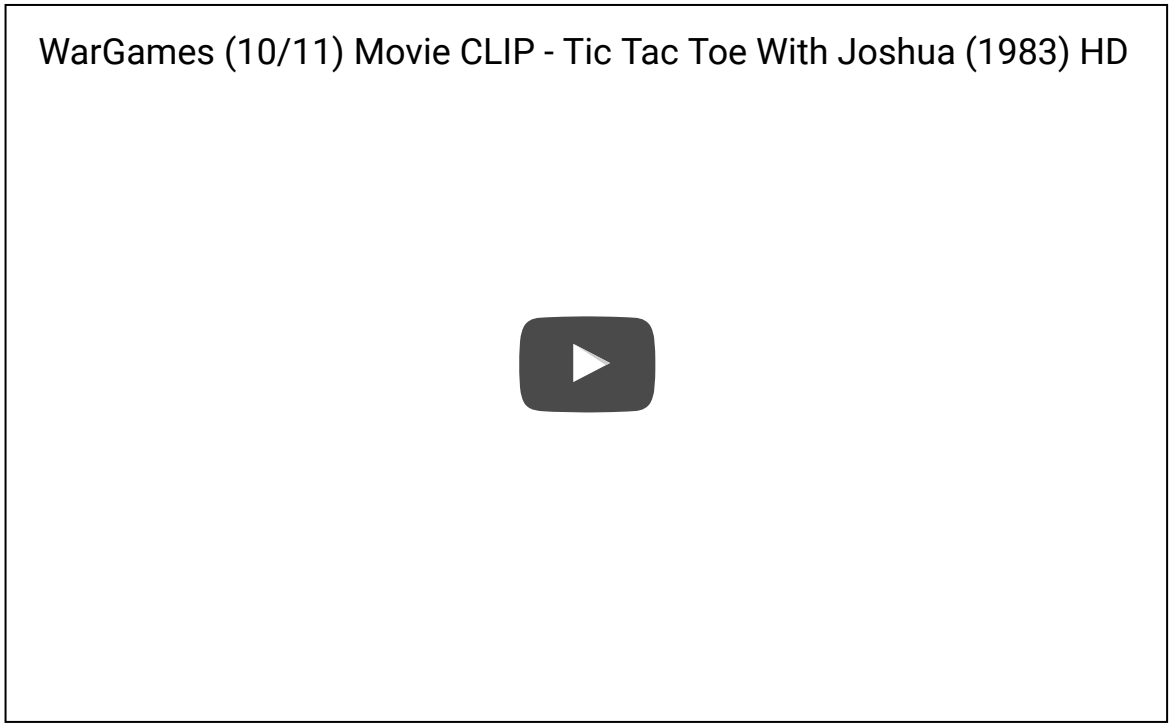
```
    winner <- FALSE # Define winner
    player <- 1 # First player
    players <- c(player1, player2)
    draw.board(game)
    while (0 %in% game & !winner) { # Keep playing until win or full b
        if (players[(player + 3) %% 3] == "human") # Human player
            move <- move.human(game)
        else # Computer player
            move <- move.computer(game, player)
        game[move] <- player # Change board
        draw.board(game)
        winner <- max(eval.game(game, 1), abs(eval.game(game, -1))) ==
        player <- -player # Change player
    }
}
```

You can play the computer by running all functions and then entering *tic.tac.toe()*.

I am pretty certain this simplified minimax algorithm is unbeatable—why don't you try to win and let me know when you do.

Tic Tac Toe War Games

Now that this problem is solved, I can finally recreate the epic scene from the WarGames movie. The Tic Tac Toe War Games code uses the functions explained above and the animation package. Unfortunately, there are not many opportunities to create sound in R.



```
# WAR GAMES TIC TAC TOE
source("Tic Tac Toe/Tic Tac Toe.R")

# Draw the game board
draw.board.wargames <- function(game) {
    xo <- c("X", " ", "O") # Symbols
    par(mar = rep(1,4), bg = "#050811")
    plot.new()
    plot.window(xlim = c(0,30), ylim = c(0,30))
    abline(h = c(10, 20), col = "#588fca", lwd = 20)
    abline(v = c(10, 20), col = "#588fca", lwd = 20)
    text(rep(c(5, 15, 25), 3), c(rep(25, 3), rep(15,3), rep(5, 3)), xo
    text(1,0,"r.prevos.net", col = "#588fca", cex=2)
    # Identify location of any three in a row
    square <- t(matrix(game, nrow = 3))
    hor <- abs(rowSums(square))
    if (any(hor == 3))
        hor <- (4 - which(hor == 3)) * 10 - 5
    else
        hor <- 0
    ver <- abs(colSums(square))
    if (any(ver == 3))
        ver <- which(ver == 3) * 10 - 5
    else
        ver <- 0
    diag1 <- sum(diag(square))
    diag2 <- sum(diag(t(apply(square, 2, rev)))) # Draw winning lines
    if (all(ver > 0)) for (i in ver) lines(rep(i, 2), c(0, 30), lwd =
    if (abs(diag1) == 3) lines(c(2, 28), c(28, 2), lwd = 10, col = "#5
    if (abs(diag2) == 3) lines(c(2, 28), c(2, 28), lwd = 10, col = "#5
}

library(animation)
player <- -1
games <- 100
saveGIF ({
    for (i in 1:games) {
        game <- rep(0, 9) # Empty board
        winner <- 0 # Define winner
        #draw.board.wargames(game)
        while (0 %in% game & !winner) { # Keep playing until win or fu
            empty <- which(game == 0)
            move <- move.computer(game, player)
            game[move] <- player
            if (i <= 12) draw.board.wargames(game)
            winner <- max(eval.game(game, 1), abs(eval.game(game, -1))
            player <- -player } if (i > 12) draw.board.wargames(game)
    }
},
```

```
interval = c(unlist(lapply(seq(1, 0,-.2), function (x) rep(x, 9))), re
movie.name = "wargames.gif", ani.width = 1024, ani.height = 1024)
```


This entry was posted in **Miscellaneous Debris** and tagged **games**, **simulation** by **Peter Prevos**. Bookmark the **permalink [http://r.prevos.net/tic-tac-toe-war-games/]** .

2 THOUGHTS ON "TIC TAC TOE WAR GAMES: THE INTELLIGENT MINIMAX ALGORITHM"

Pingback: Tic Tac Toe War Games: The Intelligent Minimax Algorithm | A bunch of data

Pingback: Tic Tac Toe War Games: The Intelligent Minimax Algorithm – Mubashir Qasim