Beating Humans at Rock Paper Scissors

Tom Parker - a1608020

June 13, 2014

1 Overview

My assumption going into this project is that humans have difficulty creating random sequences. With this is mind I set out to create a program that would observe what the patterns a player makes and use them to win at a game like Rock Paper Scissors. It makes sense that since the minimum result is a 50 percent draw from randomness that a player with extra information could win a higher percentage of games.

The expected result is for the numer of wins to be equal initially but as more games are played for the average number of wins to go in favour of the computer player.

2 The Algorithm

2.1 Outline

To get the best reults the program would do three runs when making the decision on which move to throw.

Initially it would pick one of the moves at random to throw. It will look at the move the opponant threw in the previous turn and count up the number of times each move that followed that move and throw the move to counter the one that has been used the most. So for example if rock has been thrown after scissors five time and paper had been thrown after scissors four times the computer will see rock as the more likely option and in turn play paper. In the case that there is a tie for the most likely move the computer will pick on of the moves at random so as to not be predictable. If none of the moves had been played before then the move from the previous step will be kept (this doesn't mean much now but will become important later).

Now the computer will do a third run over the data and look at the last two moves in conjunction with each other. It will then do exactly like the previous step but with two moves instead of one. So if the last two moves were rock and then scissors the program will look at all the times those moves have been made before and the frequency that each option has following them. Once again if the sequence of moves hasn't been made before then the program will keep the moves from the previous step.

Each person who plays against the computer will play against a fresh state so as not to corrupt the data.

2.2 limitations and possible extensions

Under testing I found that the algorithm designed has several limitations. The first is that for the most part the algorithm is not random, although that doesn't sound bad at first it means that an intelligent player can work out what's happening and beat the computer at its own game.

Another limitation in the current state of it is that it only looks at the last two moves that the opponent played. This could easily be fixed though with a reworking of the code used to implement the algorithm. When testing I would only use 50 games to gague the successfulness of the program so there weren't many occasions that a person would make the same three moves in a row.

In a future version of this algorithm there are methods that could be used to tip the odds in the favour of the computer. For instance, novice rock paper scissor players will overvalue the move rock, so paper could have a higher weighting.

The largest limitation in this version of the algorithm is that the computer does not look at if the previous moves were wins or losses. The qualitative data I gathered while testing the algorithm suggested that people base which move they're going to make now now only on their previous moves but also on their opponants and if they were successful. I suspect that accounting for that would greatly increase the program's win rate.

In the future it is also possible to expand the program out to varients of Rock Paper Scissors, such as Rock Paper Scissors Lizard Spock. This would take very little effort at all, however the algorithm would probably perform worse because there are more possible states.

3 Method

When testing I would stand in for the computer opponent, I would look at the move it tells me to do and I would play it against my human opponent.

I would sit across from my opponent so that could not see my computer screen, either in person or over videochat. I would then do a three count of "rock paper, scissors" revealing my move on scissors. I would tell my opponent if they won or lost and then enter the result into the computer.

We would play 50 rounds of rock paper scissors (including draws) and then I would tell my opponent what the total score was.

4 Results

Figure 1 shows the average number of wins the computer had over the human player as time went on. The graph actually follows basically what would be expected from a working algorithm, it's a bit random at the start but as it goes on it learns what the opponent is likely to do and moves to counter it. Note that this is the average number of games that the computer is leading by over a number of trials, not just one set of 50 rounds.

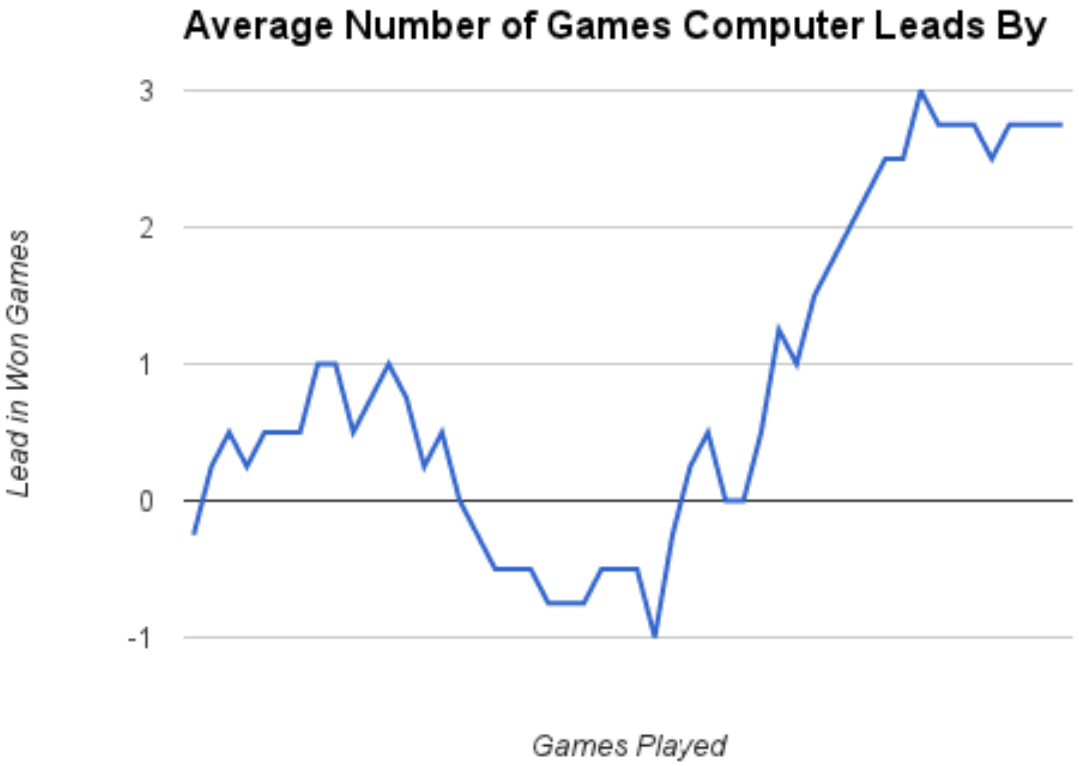

Figure 1: Average Lead Over Trial Time

5 Conclusion

I would call this a successful first version of an algorithm to beat people in rock paper scissors. I've identified flaws that could be fixed next time, as well as possible ways the algorithm could be improved. Although in some trials the human was able to achieve more wins than the program this is to be expected with the random nature of a game like this, overall the program performed better.

Appendices

Program Code

```
import random

class Agent:
    firstCheck = {}
    secondCheck = {}
    previousMoves = {}

    response = {"rock":"paper", "paper":"scissors", "scissors":"rock"}

    def __init__(self):
        #Creating random previous moves isn't perfect but the effect is negligable
        self.previousMoves[0] = random.choice(["rock", "paper", "scissors"])
        self.previousMoves[1] = random.choice(["rock", "paper", "scissors"])

        self.firstCheck["rock"] = []
        self.firstCheck["paper"] = []
        self.firstCheck["scissors"] = []

        self.secondCheck["rock"] = {"rock":[], "scissors":[], "paper":[]}
        self.secondCheck["paper"] = {"rock":[], "scissors":[], "paper":[]}
        self.secondCheck["scissors"] = {"rock":[], "scissors":[], "paper":[]}

    def addMove(self, move):
        self.firstCheck[self.previousMoves[1]].append(move)
        self.secondCheck[self.previousMoves[0]][self.previousMoves[1]].append(move)
        self.previousMoves[0] = self.previousMoves[1]
        self.previousMoves[1] = move
        return self.secondCheck

    def makeMove(self):
        #If this sequence of moves hasn't been made before a random move will be made
        highest = random.choice(["rock", "paper", "scissors"])

        #First I'll check see if there's a highest 1 gauge response
        data = self.firstCheck[self.previousMoves[1]]
        for i in {"rock", "paper", "scissors"}:
            if (data.count(i) > data.count(highest)):
                highest = i
        #Now if there isn't a second gague response at least we'll get an approximation
```

```
    #this line doesn't need to exist but it makes the code neater
    data = self.secondCheck[self.previousMoves[0]][self.previousMoves[1]]

    #check which move has been played the most in the past from this sequence
    for i in {"rock", "paper", "scissors"}:
        if (data.count(i) > data.count(highest)):
            highest = i

    return self.response[highest]


def runTest(self, n):
    wins = 0
    losses = 0
    results = [];

    for i in xrange(n):
        print "Game:", i, "Wins:", wins, "Losses:", losses

        ownMove = self.makeMove()
        print "I played:", ownMove
        otherMove = raw_input("Enter move: ")


        quick = {"p":"paper", "r":"rock", "s":"scissors"}
        if otherMove in quick:
            otherMove = quick[otherMove]

        while otherMove not in ["rock", "paper", "scissors"]:
            otherMove = raw_input("Please enter move again: ")
            if otherMove in quick:
                otherMove = quick[otherMove]

        self.addMove(otherMove)

        if (self.response[otherMove] == ownMove):
            wins += 1
            print "I win!"
        elif (self.response[ownMove] == otherMove):
              losses += 1
              print "Aww, you win!"
        else:
            print "A draw! Lets try again."
```

```
    results.append((wins, losses))
print "Wins:", wins
print "Losses:", losses
print results
```