Nash Equilibrium in a Common Schoolyard Game

Gordon Kamer
gkamer@college.harvard.edu

August 17, 2021

Abstract

A common game played amongst children called "007", "Shotgun", "Load, Shield, Shoot" and a variety of other names involves two players who simultaneously select a move from three options, similar to Rock, Paper, Scissors. In this game, players choose between reload, shield, and shoot. Unlike in Rock, Paper, Scissors, however, moves are played repeatedly, and sometimes moves influences future moves (a player cannot shoot before reloading, for example). In fact, the game exhibits a Markov process in which players transition between a finite number of states dependent only on the present game state and the pair of moves played, which are chosen probabilistically in equilibrium. When the game is in Nash equilibrium, the players have a symmetric mixed strategy. This strategy can be determined by solving a large system of non-linear equations. A numerical solution can be estimated using Python and is available in Appendix A.

1 Introduction

A common schoolyard game proceeds as follows: two players simultaneously choose one of three moves: reload, shoot, or shield. If one player chooses shoot while the other player reloads, the shooting player wins. Each reload increases a player's bullet count by one. Each play of shoot decreases a player's bullet count by one. A player may not choose shoot unless his bullet count is greater than zero. In addition to shooting while the other player reloads, a player may win the game by achieving a bullet count of five.[1] If both players achieve a bullet count of 5 in the same turn, the game resets. The game goes by a variety of names, including "Bang, Bang, Shoot", "Cowboy"[2], "007"[3], "Load, Shoot, Shield", and "Shotgun". The paper that follows demonstrates how to solve for the Nash equilibrium in this game. Appendix A includes the solution to the strategy, and Appendix B contains code that can solve for the game.

2 Playing the Game

The game begins when the players double-tap their legs with both hands simultaneously. A player signals reload by making a thumbs-up with both hands and by placing his thumbs are over his shoulders. Shoot is signaled by making both hands into finger-guns. A player signals shield by crossing both his arms over his chest, making a giant X. After each move, both players double-tap their legs in sync before displaying

[1] In one variation of the game, a special move called "bazooka" becomes available after amassing five bullets; bazooka is an automatic win unless the other player can play bazooka as well. This variation is mathematically equivalent to the variation described in the paper.

[2] At Camp Laurel South in Casco, ME, the game is called "Gash Cowboy" or "The Cowboy Game". The name sticks because the champion of the game wears a cowboy hat. The hat may only be passed to counselors in the Allagash ('gash) campus, Allagash being the name for the 5th and 6th grade boys.

[3] The name comes from the fact that some players will say "double-oh-seven" in between m oves, as one might say "Rock, Paper, Scissors, Shoot."

their next moves.

3 Game States and Notation

The entire game's context can be described by a tuple of the form (a, b) containing each player's bullet count, which is equal to the number of times a player has reloaded minus the number of times that player has shot. This number is not allowed to drop below zero. How a game arrived at a particular pairing of bullet counts does not affect the game that follows. Therefore, the game describes a Markov process.

Moreover, the number of states in this process is finite. Bullet counts may not be negative, and the game ends if either player achieves a bullet count of 5 before the other player. If both players reload in the state $(4, 4)$, the game resets, which brings the state back to $(0, 0)$. Of course, the choice of a maximum bullet count is arbitrary, and the methods described apply to any maximum bullet count. Let variations of this kind be referred to as n-bullet-bazooka games where n is the game's maximum bullet count.[4]

Let an *a-b count* refer to the game state (a, b) relative to the position of the player with the bullet count of a. Let $R_{a,b}$ refer to the probability that a player reloads in an a-b count. Let $X_{a,b}$ and $S_{a,b}$ similarly refer to the probabilities of shielding and shooting.

Finally, let the inverse of a count (a, b) refer to the count (b, a). In other words, if a player has a count (a, b), his opponent has a count (b, a), its inverse. Let a player's strategy consist of the set of values $\{R_{a,b}, X_{a,b}, S_{a,b}\} \forall a, b \in [\![0, 4]\!]$.

4 Edge Cases

On the first move, optimal play requires that both players reload; there is no need to shield because neither player is allowed to shoot without having reloaded. In fact, in any count $(a, 0)$, $X_{a,0} = 0$ because $S_{0,a} = 0$. It is also true that with optimal play, a player wins in a $(4, 0)$ count by simply reloading. Therefore, with optimal play, $R_{4,0} = 1$. On the other side, the game is lost in a count of $(0, 4)$, which means that $R_{0,4}$, $X_{0,4}$, and $S_{0,4}$ are ill-defined in Nash equilibrium.

5 Solving for Equilibrium

This game has no pure Nash equilibrium. With knowledge of the other player's moves, a player could always win by reloading or shooting while the other player reloads, by shielding when the other player shoots, and by reloading when the other player shields.

When all moves are available to each player, the game can be in Nash equilibrium only if a player is indifferent between each move given his opponent's strategy. Define $E_{a,b}$ as the expected value of a game in an a-b count where 1 is awarded for winning and -1 is awarded for losing. Consider counts in which either player can choose any move.

$$E_{a,b}[R] = E_{a+1,b+1}R_{b,a} + E_{a+1,b}X_{b,a} - S_{b,a}$$

$$E_{a,b}[S] = R_{b,a} + E_{a-1,b}X_{b,a} + E_{a-1,b-1}S_{b,a}$$

$$E_{a,b}[X] = E_{a,b+1}R_{b,a} + E_{a,b}X_{b,a} + E_{a,b-1}S_{b,a}$$

$$E_{a,b}[R] = E_{a,b}[X] = E_{a,b}[S]$$

Note that because the game is symmetric, the strategies are symmetric, and therefore the probabilities with which the the opponent reloads, shields, or shoots in an a-b count are R_{ba}, X_{ba}, and S_{ba}, respectively. In order to solve for edge cases, simply remove illegal terms, and replace the expected value of winning counts with 1 and

[4] A 2-bullet-bazooka game is equivalent to Rock, Paper, Scissors with a few extra steps. In this game, both players reload on the first move. On the second move, shoot beats reload, reload beats shield, and shield beats shoot (so long as the shielding player remembers to reload on the next move).

the expected value of losing counts with -1.

The three move probabilities in a given count must also sum to 1.

$$R_{a,b} + X_{a,b} + S_{a,b} = 1$$

Counting the three expected value variables as one variable (they must all be equal), we have four equations and four variables for each count. Therefore, we have as many equations as variables and can solve for the equilibrium strategy.

A Numerical Solution

Equilibrium Strategy ($R_{r,c}$, $X_{r,c}$, $S_{r,c}$)					
	0	1	2	3	4
0	(1.0, .00, .00)	(.47, .53, .00)	(.40, .60, .00)	(.49, .51, .00)	
1	(.70, .00, .30)	(.28, .58, .14)	(.19, .70, .11)	(.20, .69, .11)	(.29, .54, .17)
2	(.57, .00, .43)	(.24, .57, .19)	(.17, .69, .15)	(.17, .67, .67)	(.27, .48, .26)
3	(.49, .00, .51)	(.20, .58, .22)	(.14, .69, .17)	(.14, .66, .20)	(.22, .43, .35)
4	(1.0, .00, .00)	(.15, .64, .21)	(.09, .74, .17)	(.09, .70, .21)	(.13, .44, .43)

B Python Code

The code is also available at https://github.com/gkamer8/cowboy-game.

```
from scipy.optimize import fsolve

# https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fsolve.html#scipy.optimize.fsolve

# Create lookup table to keep track of variables
lu = {}
for i in range(5):
    for k in range(5):
        # No variables in a 4, 0 or 0, 4 count
        if not(i == 0 and k == 4 or i == 4 and k == 0):
            lu['EV_' + str(i) + str(k)] = len(lu)
            lu['R_' + str(i) + str(k)] = len(lu)
            # Only include shield when opponent has > 0
            if k > 0:
                lu['X_' + str(i) + str(k)] = len(lu)
            # Only include shoot when you have > 0
            if i > 0:
                lu['S_' + str(i) + str(k)] = len(lu)

def func(x):
    # xcp[k] refers to an element of x or a constant in {0, 1, -1}, including all combos ([0-6), [0-6))
    xcp = {}  # "x copy"
    for i in range(-1, 6):
        for k in range(-1, 6):
            if 'EV_' + str(i) + str(k) in lu:
                xcp['EV_' + str(i) + str(k)] = x[lu['EV_' + str(i) + str(k)]]
```

```
        elif i == 5 and k == 5:
            xcp['EV_' + str(i) + str(k)] = x[lu['EV_00']]
        elif i == 5:
            xcp['EV_' + str(i) + str(k)] = 1
        elif k == 5:
            xcp['EV_' + str(i) + str(k)] = -1
        elif i == 4 and k == 0:
            xcp['EV_' + str(i) + str(k)] = 1
        elif i == 0 and k == 4:
            xcp['EV_' + str(i) + str(k)] = -1
        else:
            xcp['EV_' + str(i) + str(k)] = 0

        # Note: the following code creates some unused variables in counts containing 5

        for move in ['R_', 'X_', 'S_']:
            if move + str(i) + str(k) in lu:
                xcp[move + str(i) + str(k)] = x[lu[move + str(i) + str(k)]]
            else:
                xcp[move + str(i) + str(k)] = 0

m = []

for i in range(5):
    for k in range(5):
        if i == 0 and k == 4 or i == 4 and k == 0:
            continue

        # Reload
        m.append(
            # Reload
            xcp['EV_' + str(i+1) + str(k+1)] * xcp['R_' + str(k) + str(i)] +
            # Shield
            xcp['EV_' + str(i+1) + str(k)] * xcp['X_' + str(k) + str(i)] +
            # Shoot
            (-1) * xcp['S_' + str(k) + str(i)] +
            # Minus EV
            (-1) * xcp['EV_' + str(i) + str(k)]
        )

        # Shield
        if k > 0:
            m.append(
                # Reload
                xcp['EV_' + str(i) + str(k+1)] * xcp['R_' + str(k) + str(i)] +
                # Shield
                xcp['EV_' + str(i) + str(k)] * xcp['X_' + str(k) + str(i)] +
                # Shoot
                xcp['EV_' + str(i) + str(k-1)] * xcp['S_' + str(k) + str(i)] +
                # Minus EV
                (-1) * xcp['EV_' + str(i) + str(k)]
            )
        # Shoot
        if i > 0:
```

```
                m.append(
                    # Reload
                    (1) * xcp['R_' + str(k) + str(i)] +
                    # Shield
                    xcp['EV_' + str(i-1) + str(k)] * xcp['X_' + str(k) + str(i)] +
                    # Shoot
                    xcp['EV_' + str(i-1) + str(k-1)] * xcp['S_' + str(k) + str(i)] +
                    # Minus EV
                    (-1) * xcp['EV_' + str(i) + str(k)]
                )

            # All equal to one
            eq = -1
            if i != 0:
                eq += xcp['S_' + str(i) + str(k)]
            if k != 0:
                eq += xcp['X_' + str(i) + str(k)]
            eq += xcp['R_' + str(i) + str(k)]
            m.append(eq)

    return m

guesses = [1/3 for _ in range(len(lu))]
for i in range(5):
    guesses[lu['EV_' + str(i) * 2]] = 0
root = fsolve(func, guesses, factor=.01)

for key in lu:
    print(key + '\t' + ': ' + str(round(root[lu[key]], 3)))
```