Learning Strategies for Poker with a Genetic Algorithm

Raphaël Morsomme

2019-01-07

Contents

```r
library(tidyverse)
library(gdata)
```

1 Introduction

In this script, I use a traditional genetic algorithm (GA) to learn profitable strategies for a fairly simple version of poker. I chose to investigate poker because I love playing this game, and to use a GA because it offers a nice coding challenge and is a suitable method for learning good strategies for this game. A question that I want to answer at the end of the script is if the GA will learn strategies prescribing to bluff.

2 Background

2.1 A Simple Version of Poker

We consider a fairly simple version of poker opposing two opponents whom we call *player A* and *player B* for convenience. A hand starts with each player paying an *ante* (small fixed amount) and receiving a single card. Player A then effectuates a *bet* and player B decides to either *fold* or *call* it. If player B folds, then player A wins the hand and recuperates the pot (the two antes). If player B calls player A's bet, then the player with the highest card wins the hand and recuperates the pots (the two antes and the two bets)[1]. If player B calls player A's bet and both players have the same card, they share the pot.

In short, a hand follows **5 steps** where we determine:

1. **Player A's bet** based on player A's strategy and player A's card.
2. **Player B's action** based on player B's strategy, player B's card and player A's bet.
3. **The size of the pot** based on player A's bet and player B's action.
4. **The winner of the hand** given the cards of the two players and player B's action.
5. **The gain/loss** of each player at the end of the hand.

Before delving into the GA itself, we examine the nature of the strategies, and how to simulate a hand.

2.2 Strategies

Player A's strategy indicates the amount to bet given the card (s)he receives. It consists of a vector of numeric values indicating the amount to bet for each card (s)he could receive. If we consider a game where player A can bet the following amounts 0, 2, 4, 6, 8, 10 and the cards are 1, 2, 3, 4, the following is a possible strategy for player A

```
##   Strategy A
## 1          0
## 2         10
## 3          2
## 4          8
```

that prescribes to bet `0` if (s)he receives a card `1`, `10` (a bluff!) if (s)he receives a `2`, `2` if (s)he receives a `3`, and to bet `8` if (s)he receives a `4`.

[1] For instance, if the ante is `5`, player A makes a bet of `7` and player B decides to call, then the player with the highest card recuperates the pot of $2 * 5 + 2 * 7 = 24$.

Player B's strategy indicates the action to realise i.e. whether to *fold* or to *call* given two elements: player A's bet and player B's card[2]. It consists of a matrix whose entries indicate the action to realise for each possible combination of her/his card and player A's bet. The following is a possible strategy for player B

```
## , , Strategy B
##
##    1      2      3      4
## 0  "Fold" "Fold" "Fold" "Fold"
## 2  "Fold" "Fold" "Call" "Fold"
## 4  "Call" "Call" "Fold" "Call"
## 6  "Fold" "Fold" "Call" "Fold"
## 8  "Call" "Call" "Call" "Fold"
## 10 "Fold" "Call" "Fold" "Call"
```

that prescribes to `fold` if (s)he receives a card `1` and player A bets `0`, to `call` if (s)he receives a `2` and player A bets `4` and to `call` if (s)he receives a `4` and player A bets `10`. Now that we can generate a strategy for each player, let us simulate a hand.

2.3 Simulating a Hand

2.3.1 Setup

We assign the cards we play with, the possible bets for player A and the value of the ante to `cards`, `bets` and `ante` respectively.

```
cards  <- 1:4
bets   <- seq(from = 0, to = 10, by = 2)
ante   <- 2

n_card <- length(cards) # for convenience
n_bet  <- length(bets)  # for convenience
```

We use `sample()` in `array()` to create random strategies for the two players and assign them to `strategy_A` and `strategy_B`.

```
set.seed(123)
strategy_A <- array(sample(x = bets, size = n_card, replace = T),
                    dim = c(n_card, 1), dimnames = list(cards, "Strategy A"))
strategy_B <- array(sample(x = c("Call", "Fold"), n_card * n_bet, T),
                    dim=c(n_card, n_bet, 1), dimnames = list(cards, bets, "Strategy B"))
# Random strategy for player A
print(strategy_A)
```

```
##   Strategy A
## 1          2
## 2          8
## 3          4
## 4         10
```

```
# Random strategy for player B
print(strategy_B)
```

```
## , , Strategy B
##
##   0      2      4      6      8      10
```

[2]In this script, I investigate *deterministic* strategies. I will consider strategies with a stochastic element in another script.

```
## 1 "Fold" "Fold" "Fold" "Call" "Fold" "Fold"
## 2 "Call" "Call" "Fold" "Call" "Fold" "Fold"
## 3 "Fold" "Fold" "Call" "Call" "Fold" "Fold"
## 4 "Fold" "Call" "Fold" "Fold" "Fold" "Fold"
```

2.3.2 The Naive Approach

To simulate a hand, one could be tempted to imitate what would happen if two players sat down around a table to play a hand[^12]. Although such code would be very intuitive, it is painfully slow for simulating a large number of hands – whihc is exactly what the GA requires – as one needs to use loops (slow on `R`) to accomplish this. We therefore need a more efficient approach.

2.3.3 The Matrix-Oriented Approach

The *matrix-oriented* approach uses matrices and matrix operations (fast on `R`) to simulate a large number of hands at once. We generate 5 matrices (one per step) whose columns and rows respectively correspond to player A's and player B's cards. This way, we can simulate all possible hands at once. This is more efficient than the naive approach and is thus prefered for the GA. The matrices' entries respectively represent (i) player A's bet (ii) player B's action (iii) the pot size (iv) the winner of the hand and (v) player A's gain/loss.

1. **Determining player A's bet**

The matrix `bet_A` represents player A's bet for each possible hand. Since player A's bet only depends on her/his card, each column contains the same values.

```
dim_mat     <- c(n_card, n_card)  # for convenience
dimname_mat <- list(cards, cards) # for convenience

bet_A <- array(rep(strategy_A, each = n_card), dim = dim_mat, dimnames = dimname_mat)
# Player A's bet
print(bet_A)
```

```
##   1 2 3  4
## 1 2 8 4 10
## 2 2 8 4 10
## 3 2 8 4 10
## 4 2 8 4 10
```

2. **Determining player B's action**

The matrix `action_B` represents player B's action. Since `bet_A`'s columns are uniform, `action_B` is simply a re-ordering of `strategy_B`'s columns.

```
action_B <- array(strategy_B[, match(strategy_A, bets), 1], dim = dim_mat, dimnames = dimname_mat)
# Player B's strategy
print(strategy_B)
```

```
## , , Strategy B
##
##   0      2      4      6      8      10
## 1 "Fold" "Fold" "Fold" "Call" "Fold" "Fold"
## 2 "Call" "Call" "Fold" "Call" "Fold" "Fold"
## 3 "Fold" "Fold" "Call" "Call" "Fold" "Fold"
## 4 "Fold" "Call" "Fold" "Fold" "Fold" "Fold"
```

```r
# Player B's action
print(action_B)
```

```
##   1      2      3      4
## 1 "Fold" "Fold" "Fold" "Fold"
## 2 "Call" "Fold" "Fold" "Fold"
## 3 "Fold" "Fold" "Call" "Fold"
## 4 "Call" "Fold" "Fold" "Fold"
```

3. **Determining the size of the pot**

`pot_size` indicates the size or the pot. If player B folds, then the pot only consists of the two antes.

```r
pot_size                     <- 2 * ante + 2 * bet_A
pot_size[action_B == "Fold"] <- 2 * ante
# Size of the pot
print(pot_size)
```

```
##   1 2  3 4
## 1 4 4  4 4
## 2 8 4  4 4
## 3 4 4 12 4
## 4 8 4  4 4
```

4. **Determining the winner of hand**

`win_game` indicates the winner of the hand (`1`: player A wins, `0`: a draw and `-1`: player B wins). If player B folds, then player A wins the hand.

```r
win_game <- array(numeric(n_card * n_card), dim = dim_mat, dimnames = dimname_mat)

upperTriangle(win_game, diag = F) <-  1
lowerTriangle(win_game, diag = F) <- -1

win_game[action_B == "Fold"] <- 1
# Winner of the hand
print(win_game)
```

```
##    1 2 3 4
## 1  1 1 1 1
## 2 -1 1 1 1
## 3  1 1 0 1
## 4 -1 1 1 1
```

5. **Determining player A's gain/loss**

`gain_A` represents the amount won by player A. A negative amount indicates that player A loses money. Since this is a zero-sum game between two players, player B's gain/loss is simply the opposite of player A's.

```r
gain_A <- pot_size * win_game / 2

# Player A's gain/loss
print(gain_A)
```

```
##    1 2 3 4
## 1  2 2 2 2
## 2 -4 2 2 2
## 3  2 2 0 2
## 4 -4 2 2 2
```

```r
# Player B's gain/loss
print(- gain_A)
```

```
##    1  2  3  4
## 1 -2 -2 -2 -2
## 2  4 -2 -2 -2
## 3 -2 -2  0 -2
## 4  4 -2 -2 -2
```

3 The Genetic Algorithm

3.1 Overview

In order to learn good strategies for our two players, we use a GA. Simply put, a GA simulates how a process of *natural selection* iteratively selects from an existing population the fittest individuals and cross over their genes to generate new individuals that replace the old ones. A *good* GA produces, after some time, a population of fit individuals. In our case, the GA learns profitable strategies for the two players. The question that we want to answer is whether the strategies that the GA learns contain contain an element of bluff.

In practice, we first consider an initial *population* of strategies for each player; these are the strategies with which we start the GA[3]. We then make each strategy of player A's population play against each strategy of player B's population. Based on the results of these confrontations, we determine how competitive (or fit) each strategy is. Finally, for each player, we combine the most competitive strategies together to generate new populations of strategies. This way, features that make strategies competitive are passed on to the next generation. These new populations replace the old ones and the genetic algorithm repeats the cycle: confrontation, evaluation, generation and replacement.

In short, the GA follows **5 steps**:

1. **Creation of initial populations** of strategies for player A and player B.
2. **Confrontation between strategies.**
3. **Evaluation of strategies' fitness** based on the results of the confrontations.
4. **Generation of new populations of strategies** from the fittest strategies.
5. **Repeat from step 2.**

The following sections explore in more depth the concepts of population of strategies, fitness, confrontataion, evaluation and generation of strategies. At the end of the section, we will be able to apply the GA.

3.2 Population of Strategies

A population of strategies is simply a collection of strategies for a player. A population of strategies for player A is a matrix where each column correspond to a strategy. For player B, it is a `3`-dimensional array where each layer (dimension `3`) corresponds to a strategy. The following code creates populations of `10` random strategies for player A and player B. We use the function `sample()` in the function `array()` to create populations of strategies for each player and assign them to `pop_A` and `pop_B`. To easily identify the strategies, we name them s1, s2, s3, s4, s5, s6, s7, s8, s9, s10.

```r
n_strategy <- 10
name_strategy <- paste("s", 1 : n_strategy, sep="")
```

[3]How we obtained these strategies does not matter for the moment. They could for instance be randomly generated (as will be the case in this script), but could also come from an external source.

```r
dim_pop_A     <- c(n_card, n_strategy)            # for convenience
dimname_pop_A <- list(cards, name_strategy)       # for convenience
dim_pop_B     <- c(n_card, n_bet, n_strategy)     # for convenience
dimname_pop_B <- list(cards, bets, name_strategy) # for convenience

pop_A <- array(sample(x = bets, size = prod(dim_pop_A), replace = T),
               dim = dim_pop_A, dimnames = dimname_pop_A)
# Population of strategies for player A
print(pop_A)
```

```
##   s1 s2 s3 s4 s5 s6 s7 s8 s9 s10
## 1  2  8  8  0  0  2  8  0  6   8
## 2  0  8  2  4  0 10  0  8  0   4
## 3 10  0  2  4  2  0  6 10  4   8
## 4 10  4  2  4  4  4  2  4  2   8
```

```r
pop_B <- array(sample(x = c("Call", "Fold"), size = prod(dim_pop_B), replace = T),
               dim = dim_pop_B, dimnames = dimname_pop_B)
# Population of strategies for player B (2 strategies)
print(pop_B[,,1:2])
```

```
## , , s1
##
##   0      2      4      6      8      10
## 1 "Fold" "Fold" "Call" "Call" "Call" "Fold"
## 2 "Call" "Call" "Fold" "Fold" "Call" "Call"
## 3 "Fold" "Call" "Call" "Call" "Fold" "Call"
## 4 "Fold" "Call" "Call" "Fold" "Fold" "Fold"
##
## , , s2
##
##   0      2      4      6      8      10
## 1 "Call" "Fold" "Fold" "Call" "Call" "Call"
## 2 "Fold" "Call" "Call" "Fold" "Call" "Fold"
## 3 "Call" "Call" "Call" "Fold" "Fold" "Fold"
## 4 "Call" "Fold" "Fold" "Fold" "Call" "Call"
```

3.2.1 Cleaning Player B's Strategies

We can slightly modify player B's strategies to make them more realistic. Following common sense, we impose that player B *calls* if her/his card is the highest or if player A bets `0`. Indeed, in both cases, player B risks nothing by calling player A's bet: those hands either end up in a draw or a win for player B.

```r
clean_pop_B <- function(pop, cards){
  pop[max(cards),    ,] <- "Call"
  pop[          , "0",] <- "Call"
  return(pop)
}

pop_B <- clean_pop_B(pop = pop_B, cards = cards)
# Clean population for player B
print(pop_B[,,1:2]) # Compare with the strategies before the cleaning (printed above).
```

```
## , , s1
```

```
##
##   0      2      4      6      8      10
## 1 "Call" "Fold" "Call" "Call" "Call" "Fold"
## 2 "Call" "Call" "Fold" "Fold" "Call" "Call"
## 3 "Call" "Call" "Call" "Call" "Fold" "Call"
## 4 "Call" "Call" "Call" "Call" "Call" "Call"
##
## , , s2
##
##   0      2      4      6      8      10
## 1 "Call" "Fold" "Fold" "Call" "Call" "Call"
## 2 "Call" "Call" "Call" "Fold" "Call" "Fold"
## 3 "Call" "Call" "Call" "Fold" "Fold" "Fold"
## 4 "Call" "Call" "Call" "Call" "Call" "Call"
```

3.3 Fitness of a Strategy

The notion of *fitness* is central to the GA: it allows the algorithm to select the best performing strategies from which to generate new ones. In our case, the goal of the GA is to generate competitive strategies. An obvious measure of fitness of a strategy is thus how much money it yields, on average, against the strategies of the other player's population.

3.3.1 Confrontations: one v. one

To determine the fitness of a strategy, we must thus determine its average gain/loss against each strategy of the other player's population. The function `confront` does exactly this. It follows the matrix-oriented approach to simulate all possible hands between two strategies and then simply return the `mean` of `gain_A` which corresponds to the average gain/loss of player A's strategy when opposed to that of player B[4].

```
win_game <-   win_game <- array(numeric(n_card*n_card),
dim = dim_mat)
upperTriangle(win_game, diag = F) <- 1
lowerTriangle(win_game, diag = F) <- -1

confront <- function(strategy_A, strategy_B, n_card, dim_mat, bets, ante, win_game){

  # Matrix-oriented approach
  bet_A    <- array(rep(strategy_A, each = n_card       ) , dim = dim_mat)
  action_B <- array(strategy_B[ , match(strategy_A, bets)], dim = dim_mat)
  gain_A                   <- (ante + bet_A) * win_game
  gain_A[action_B=="Fold"] <-  ante

  # Average gain for player A
  return(mean(gain_A))

}
```

Here is an example of the function `confront` in action.

```
strat_A <- pop_A[ , 3]
print(strat_A)
```

[4]To make the function `confront` run faster, we create `win_game` outside and consequently adapt how we compute `gain_A`.

```
## 1 2 3 4
## 8 2 2 2
```

```
strat_B <- pop_B[ , , 1]
print(strat_B)
```

```
##   0      2      4      6      8      10
## 1 "Call" "Fold" "Call" "Call" "Call" "Fold"
## 2 "Call" "Call" "Fold" "Fold" "Call" "Call"
## 3 "Call" "Call" "Call" "Call" "Fold" "Call"
## 4 "Call" "Call" "Call" "Call" "Call" "Call"
```

```
average_gain_A <- confront(strategy_A = strat_A, strategy_B = strat_B,
                           n_card = n_card, dim_mat = dim_mat, bets = bets,
                           ante = ante, win_game = win_game)
# Average gain of player A
print(average_gain_A)
```

```
## [1] -0.75
```

This means that, on average, `strat_A`, wins -0.75 for each hand played against `strat_B`. Since the matrix-oriented approach simulates each possible hand once, taking the `mean` of `gain_A` produces the *true* average gain of `strat_A` when opposed to `strat_B`.

3.3.2 Confrontations: all v. all

The function `confront_populations` uses two loops to make each strategy of player A play against each strategy of player B and return the result of these confrontations. In the loops, we use the function `confront` to determine the average gain of player A's strategy against that of player B and record the value in the matrix `fitness` whose columns and rows respectively correspond to player A's and player B's strategies.

```
fitness <- array(NA, dim=c(n_strategy, n_strategy),
                 dimnames = list(name_strategy, name_strategy))

confront_populations <- function(pop_A, pop_B, name_strategy, fitness, n_card,
                                 dim_mat, bets, ante, win_game){
  for(strat_a in name_strategy){
    for(strat_b in name_strategy){
      fitness[strat_b, strat_a] <- confront(strategy_A = pop_A[ , strat_a  ],
                                            strategy_B = pop_B[ , , strat_b],
                                            n_card = n_card, dim_mat = dim_mat,
                                            bets = bets, ante = ante, win_game = win_game)
    } # end-for
  } # end-for
  return(fitness)
}
```

Here is an example of the function `confront_populations` in action.

```
print(pop_A)
```

```
##   s1 s2 s3 s4 s5 s6 s7 s8 s9 s10
## 1  2  8  8  0  0  2  8  0  6   8
## 2  0  8  2  4  0 10  0  8  0   4
## 3 10  0  2  4  2  0  6 10  4   8
## 4 10  4  2  4  4  4  2  4  2   8
```

```
print(pop_B[, , 1:2])
```

```
## , , s1
##
##   0      2      4      6      8      10
## 1 "Call" "Fold" "Call" "Call" "Call" "Fold"
## 2 "Call" "Call" "Fold" "Fold" "Call" "Call"
## 3 "Call" "Call" "Call" "Call" "Fold" "Call"
## 4 "Call" "Call" "Call" "Call" "Call" "Call"
##
## , , s2
##
##   0      2      4      6      8      10
## 1 "Call" "Fold" "Fold" "Call" "Call" "Call"
## 2 "Call" "Call" "Call" "Fold" "Call" "Fold"
## 3 "Call" "Call" "Call" "Fold" "Fold" "Fold"
## 4 "Call" "Call" "Call" "Call" "Call" "Call"
```

```
fitness <- confront_populations(pop_A = pop_A, pop_B = pop_B, fitness = fitness,
                                name_strategy = name_strategy, win_game = win_game,
                                n_card = n_card, dim_mat = dim_mat, bets = bets, ante = ante)
# Average gains of player A's strategies
print(fitness)
```

```
##         s1     s2     s3    s4    s5     s6     s7     s8     s9    s10
## s1   1.000  0.000 -0.750 0.375 0.500 -1.000 -0.500  0.750 -0.250  0.750
## s2   0.500  0.000 -0.750 0.000 0.500  0.625 -0.375  0.875  0.375  0.375
## s3  -0.750  0.500  0.125 0.375 0.500  0.000  0.500 -0.125  0.375 -0.375
## s4   0.500 -2.125 -1.375 0.750 0.250 -0.375 -0.750 -0.625 -0.750  0.000
## s5   1.250 -0.875  0.125 0.000 0.250 -1.250 -0.250 -0.250  0.250  0.125
## s6   3.000 -0.625 -0.500 0.375 0.500  0.375 -0.625  0.750  0.125  0.125
## s7   1.250  0.250  0.875 0.750 0.500 -1.000  0.750  0.125 -0.750  0.125
## s8   2.625 -0.375 -0.750 0.375 0.375  0.000 -0.875  0.875 -0.250 -0.125
## s9   1.375 -2.125 -1.500 0.750 0.125 -0.875 -0.750 -0.750 -0.250  0.000
## s10  1.250 -0.625 -0.375 0.750 0.250 -1.000 -0.375  0.000 -0.625  0.375
```

`colMeans` of `fitness` gives player A's strategies's fitness and `rowMeans` of `-fitness` that of player B's.

```
# Fitness of player A's strategy
sort(colMeans( fitness), decreasing = T)
```

```
##      s1      s4      s5      s8     s10      s9      s7      s6      s3
##  1.2000  0.4500  0.3750  0.1625  0.1375 -0.1750 -0.3250 -0.4500 -0.4875
##      s2
## -0.6000
```

```
# Fitness of player B's strategy
sort(rowMeans(-fitness), decreasing = T)
```

```
##      s4      s9      s5     s10      s1      s3      s8      s2      s7
##  0.4500  0.4000  0.0625  0.0375 -0.0875 -0.1125 -0.1875 -0.2125 -0.2875
##      s6
## -0.3500
```

Let us have a closer look at player A' strategies (easier to analyze than player B's). `s5` is the best performing strategy against `pop_B`. It is a conservative strategy with small bets for small cards and large bets for large cards. With `s5`, player A wins on average 0.375 per hand against player B's strategies. On the opposite

spectrum, `s1` and `s8` have the lowest fitness. Both strategies prescribe to bet `10` (largest bet) for a `0` (lowest card) and `s8` even prescribes to bet `0` for a `4` (highest card)! Although the large bets for small cards could be bluffs and are thus not bad per se, betting `0` for the highest card is a bad idea[5]. Player A loses on average -1.2 with `s1` and -0.1625 with `s8`.

3.4 Generating New Strategies

Now that we have the strategies' fitness, we can generate a new population following **three steps**:

1. **Parent Selection**
2. **Crossover**
3. **Mutation**

For convenience, we use the term *parent strategy* to refer to a strategy from which new strategies are generated and *child strategy* to denote a newly generated strategy. We demonstrate how to generate `5` child strategies for player A. The approach is fundamentally the same for player B .

3.4.1 Parent Selection

We select the fittest strategies from `pop_A` to form the set of parent strategies `parents`. In our example, we select the `7` (arbitrarily chosen number) fittest strategies.

```
n_parents  <- 7
n_children <- 5

fitness_sorted  <- sort(colMeans(fitness), decreasing  = T)
fitness_parents <- head(fitness_sorted, n_parents)
# Name and fitness of the set of parent strategies
print(fitness_parents)
```

```
##      s1      s4      s5      s8     s10      s9      s7
##  1.2000  0.4500  0.3750  0.1625  0.1375 -0.1750 -0.3250
```

```
name_parents    <- names(fitness_parents)
parents         <- pop_A[ , name_parents]
# Set of parent strategies
print(parents)
```

```
##   s1 s4 s5 s8 s10 s9 s7
## 1  2  0  0  0   8  6  8
## 2  0  4  0  8   4  0  0
## 3 10  4  2 10   8  4  6
## 4 10  4  4  4   8  2  2
```

3.4.2 Children Generation

To generate child strategies, we cross over the parent strategies and introduce random mutations. In the crossover step, we randomly combine parent strategies, element by element, to generate the child strategies. In other words, the first element of a child strategy corresponds to the first element of a randomly selected parent strategy, the second element to the second element of a randomly selected parent, etc. In the mutation step, we randomly alter a small proportion the child strategies' elements.

[5]For instance, if the ante is `5`, player A makes a bet of `7` and player B decides to call, then the player with the highest card recuperates the pot of $2 * 5 + 2 * 7 = 24$.

3.4.2.1 Children Generation: Crossover

We first use the function `sample()` on the parent strategies' names to create `pop_child` which represents the population of child strategies. At this stage, `pop_child`'s entries indicate the parent strategy from which each of its elements is inherited. Note that in `sample`, we set `prob = exp(fitness_parents)` so that children strategies are more likely to inherit their elements from fitter parent strategies.

```
pop_child <- array(sample(name_parents, size = n_card*n_children, replace = T,
                          prob = exp(fitness_parents)), # exp() ensures positive probabilities
                   dim = c(n_card, n_children),
                   dimnames = list(cards, paste("Child", 1 : n_children)))
# Origin of the element of the child strategies
print(pop_child)
```

```
##   Child 1 Child 2 Child 3 Child 4 Child 5
## 1 "s4"    "s8"    "s10"   "s9"    "s1"
## 2 "s5"    "s1"    "s1"    "s1"    "s1"
## 3 "s4"    "s4"    "s4"    "s1"    "s7"
## 4 "s4"    "s10"   "s9"    "s8"    "s1"
```

The first two elements of `Child 1` come from the parent strategy `s7`, the third element from `s9`, etc.

Next, we loop through the the names of the parent strategies. In the loop, we first assign to `strategy_parent` the parent strategy with the appropriate name. We then assign to `location_parent` the location of the elements of `pop_child` that are inherited from `strategy_parent`. Finally, we substitute these elements inherited from `strategy_parent` with the corresponding elements of `strategy_parent`. (Since the dimensions of `pop_child` and `strategy_parent` are different, we use `rep()` on the latter.)

```
for(name_parent in name_parents){

  strategy_parent <- parents[ , name_parent]
  location_parent <- pop_child == name_parent

  pop_child[location_parent] <- rep(strategy_parent, n_children)[location_parent]

}

pop_child <- array(as.numeric(pop_child),
                   dim = c(n_card, n_children),
                   dimnames = list(cards, paste("Child", 1:n_children)))
# Population of children strategies
print(pop_child)
```

```
##   Child 1 Child 2 Child 3 Child 4 Child 5
## 1       0       0       8       6       2
## 2       0       0       0       0       0
## 3       4       4       4      10       6
## 4       4       8       2       4      10
```

3.4.2.2 Children Generation: Mutations

Finally, we introduce random mutations to `pop_child`. We first assigne the desired mutation rate to `mutation_rate` and use the function `sample()` with `prob = c(mutation_rate, 1 - mutation_rate)` to create the matrix `mutation_location` which indicates the location of the mutations. We then use the function `sample()` again to generate the vector `mutation_outcome` which indicates the outcome of the mutations. Finally, we subsitute the elements of `pop_child` where a mutation occurs with the values of `mutation_outcome`.

```r
mutation_rate <- 0.2

mutation_location <- array(sample(c(T, F), size = n_card * n_children, replace = T,
                                  prob = c(mutation_rate, 1 - mutation_rate)),
                  dim = c(n_card, n_children),
                  dimnames = list(cards, paste("Child", 1 : n_children)))
# Location of mutations
print(mutation_location)
```

```
##   Child 1 Child 2 Child 3 Child 4 Child 5
## 1   FALSE   FALSE   FALSE   FALSE   FALSE
## 2    TRUE    TRUE   FALSE   FALSE   FALSE
## 3   FALSE   FALSE   FALSE   FALSE    TRUE
## 4   FALSE   FALSE    TRUE   FALSE   FALSE
```

```r
n_mutation       <- sum(mutation_location)
mutation_outcome <- sample(bets, size = n_mutation, T)
# Outcome of mutations
print(mutation_outcome)
```

```
## [1]  2  6  0 10
```

```r
pop_child[mutation_location] <- mutation_outcome
# Population of child strategies after mutations
print(pop_child)
```

```
##   Child 1 Child 2 Child 3 Child 4 Child 5
## 1       0       0       8       6       2
## 2       2       6       0       0       0
## 3       4       4       4      10      10
## 4       4       8       0       4      10
```

3.4.3 Code

The functions `generat_A` and `generate_B` encapsulate these three steps for player A and player B. These functions are the last one we need to be able to run the GA.

```r
generate_A <- function(fitness, pop, n_parents, dim_pop_A, dimname_pop_A,
                       n_strategy, bets, mutation_rate){

  # Parent Selection
  fitness_strategy <- colMeans(fitness)
  fitness_parents  <- head(sort(fitness_strategy, decreasing  = T), n_parents)
  name_parents     <- names(fitness_parents)
  parents          <- pop[ , name_parents]

  # Crossover
  pop <- array(sample(name_parents, size = prod(dim_pop_A), replace = T,
                      prob = exp(fitness_parents)),
               dim = dim_pop_A, dimnames = dimname_pop_A)

  for(parent in name_parents){

    strategy_parent <- parents[ , parent]
    location_parent <- pop == parent
```

```
    pop[location_parent] <- rep(strategy_parent, n_strategy)[location_parent]

  }

  pop <- array(as.numeric(pop), dim = dim_pop_A, dimnames = dimname_pop_A)

  # Mutation
  mutation_location <- array(sample(c(T, F), size = prod(dim_pop_A), T,
                                    prob = c(mutation_rate, 1-mutation_rate)),
                             dim = dim_pop_A, dimnames = dimname_pop_A)

  n_mutations      <- sum(mutation_location)
  mutation_outcome <- sample(bets, size = n_mutations, replace = T)

  pop[mutation_location] <- mutation_outcome

  return(pop)

}
```

```
generate_B <- function(fitness, pop, n_parents, dim_pop_B, dimname_pop_B,
                       n_strategy, mutation_rate, cards){

  # Parent Selection
  fitness          <- - fitness
  fitness_strategy <- rowMeans(fitness)
  fitness_parents  <- head(sort(fitness_strategy, decreasing  = T), n_parents)
  name_parents     <- names(fitness_parents)
  parents          <- pop[ , , name_parents]

  # Crossover
  pop <- array(sample(name_parents, size = prod(dim_pop_B), replace = T,
                      prob = exp(fitness_parents)),
               dim = dim_pop_B, dimnames = dimname_pop_B)

  for(parent in name_parents){

    strategy_parent <- parents[ , , parent]
    location_parent <- pop == parent

    pop[location_parent] <- rep(strategy_parent, n_strategy)[location_parent]

  }

  # Mutation
  mutation_rate <- 2 * mutation_rate
  # since player B has only two actions i.e. "Call" or "Fold"
  # half of the mutations have no effect.
  mutation_location <- array(sample(c(T, F), size = prod(dim_pop_B), replace = T,
                                    prob = c(mutation_rate, 1-mutation_rate)),
                             dim = dim_pop_B, dimnames = dimname_pop_B)

  n_mutations            <- sum(mutation_location)
  mutation_outcome       <- sample(c("Call", "Fold"), n_mutations, T)
```

```
  pop[mutation_location] <- mutation_outcome

  pop <- clean_pop_B(pop = pop, cards = cards)

  return(pop)

}
```

Here is an example of the function `generate_A` in action.

```
pop_A_child <- generate_A(fitness = fitness, pop = pop_A, n_parents = n_parents,
                          dim_pop_A = dim_pop_A, dimname_pop_A = dimname_pop_A,
                          n_strategy = n_strategy, bets = bets, mutation_rate = 0.05)

# Parent Strategies
print(pop_A)       # random bets for small cards
```

```
##   s1 s2 s3 s4 s5 s6 s7 s8 s9 s10
## 1  2  8  8  0  0  2  8  0  6   8
## 2  0  8  2  4  0 10  0  8  0   4
## 3 10  0  2  4  2  0  6 10  4   8
## 4 10  4  2  4  4  4  2  4  2   8
```

```
# Child Strategies
print(pop_A_child) # small bets for small cards.
```

```
##   s1 s2 s3 s4 s5 s6 s7 s8 s9 s10
## 1  0  0 10  0  0  6  2  0  0   2
## 2  0  4  0  4  0  4  0  4  0   0
## 3 10 10  4 10  4  2 10  4 10   6
## 4  2  2 10 10  4  2  2  8 10  10
```

4 The Genetic Algorithm in Action

Now that we have covered all the components of the GA, we can finally write the function `my_GA` which encapsulates them. The elements `gain_A` and `call_B` keeps track of the average gain of player A's strategies and of how often player B calls player A's bet. For the generations `gen_print`, we also generate four plots which give us an overview of the players' strategies. The two lower plots and the upper left plots are self explanatory. The upper right plot shows the average strategy of player B. The tone of the color indicates the average action prescribed by the strategies in player B's population: a darker color indicates that more strategies prescribe to call

```
my_GA <- function(cards = 1:10, bets = seq(0,20,2), ante = 5, n_strategy = 200,
                  n_generations = 200, prop_parents = 2/3, mutation_rate = 0.05){

  # Setup
  n_card        <- length(cards)
  n_bet         <- length(bets)
  n_parents     <- n_strategy * prop_parents

  name_strategy <- paste("s", 1 : n_strategy, sep = "")

  dim_pop_A     <- c(n_card, n_strategy)
  dim_pop_B     <- c(n_card, n_bet, n_strategy)
```

```r
dim_mat       <- c(n_card, n_card)
dim_fit       <- c(n_strategy, n_strategy)
dim_output_A  <- c(n_card, n_strategy, n_generations)
dim_output_B  <- c(n_card, n_bet, n_strategy, n_generations)

dimname_pop_A    <- list(cards, name_strategy)
dimname_pop_B    <- list(cards, bets, name_strategy)
dimname_mat      <- list(cards, cards)
dimname_fit      <- list(name_strategy, name_strategy)
dimname_output_A <- list(cards, name_strategy, 1:n_generations)
dimname_output_B <- list(cards, bets, name_strategy, 1:n_generations)

win_game      <- array(numeric(n_card * n_card), dim = dim_mat)
upperTriangle(win_game, diag = F) <-  1
lowerTriangle(win_game, diag = F) <- -1

fitness <- array(NA, dim = dim_fit, dimnames = dimname_fit)

output_A <- array(NA, dim = dim_output_A, dimnames = dimname_output_A)
output_B <- array(NA, dim = dim_output_B, dimnames = dimname_output_B)

# Initialization
generation <- 1

pop_A <- array(sample(bets, size = prod(dim_pop_A), replace = T),
               dim = dim_pop_A, dimnames = dimname_pop_A)
pop_B <- array(sample(c("Call", "Fold"), size = prod(dim_pop_B), replace = T),
               dim = dim_pop_B, dimnames = dimname_pop_B)
pop_B <- clean_pop_B(pop_B, cards = cards)

output_A[ , , generation  ] <- pop_A
output_B[ , , , generation] <- pop_B


# Loop
while(generation < n_generations){

  fitness <- confront_populations(pop_A = pop_A, pop_B = pop_B,
                                  name_strategy = name_strategy,
                                  fitness = fitness, n_card = n_card,
                                  dim_mat = dim_mat, bets = bets,
                                  ante = ante, win_game = win_game)

  pop_A <- generate_A(fitness = fitness, pop = pop_A, n_parents = n_parents,
                      dim_pop_A = dim_pop_A, dimname_pop_A = dimname_pop_A,
                      n_strategy = n_strategy, bets = bets,
                      mutation_rate = mutation_rate)

  pop_B <- generate_B(fitness = fitness, pop = pop_B, n_parents = n_parents,
                      dim_pop_B = dim_pop_B, dimname_pop_B = dimname_pop_B,
                      n_strategy = n_strategy, cards = cards,
                      mutation_rate = mutation_rate)
```

```
    generation <- generation + 1

    output_A[ , , generation  ] <- pop_A
    output_B[ , , , generation] <- pop_B

  } # close for-loop

  return(list(A = output_A, B = output_B))

}
```

Finally, the GA in action:

```
set.seed(123)
results <- my_GA(n_generations = 2)

results_A <- results$A
results_B <- results$B

# 10 strategies for player A during the 2nd generation
results_A[ , 1 : 10, 2]
```

```
##    s1 s2 s3 s4 s5 s6 s7 s8 s9 s10
## 1  20  0 10  6  4  0  4 12  2  10
## 2   4  8  4  8  2  6  6  2  4   0
## 3  14  4  8 12  6  8 12 12  4  12
## 4   2  8 18 18  4 18  8 14  6  10
## 5   6  8 12  4  6 16  4  4 16  14
## 6  18  8 12  2 10 20  0 12 14   0
## 7   6  8 18 18  2  2 16 12  8  14
## 8  12 18 16 14  2  2  8  6 16   0
## 9  12  2 20  6  6 14  4  6  0   6
## 10 20  8 10 20 20  8 20  8 20  12
```

```
# 2 strategies for player B during the 2nd generation
results_B[ , , 25 : 26, 2]
```

```
## , , s25
##
##    0      2      4      6      8      10     12     14     16     18
## 1  "Call" "Fold" "Fold" "Call" "Fold" "Fold" "Call" "Fold" "Fold" "Fold"
## 2  "Call" "Fold" "Fold" "Fold" "Call" "Call" "Fold" "Call" "Call" "Fold"
## 3  "Call" "Fold" "Fold" "Call" "Call" "Fold" "Fold" "Fold" "Call" "Fold"
## 4  "Call" "Fold" "Fold" "Fold" "Fold" "Call" "Call" "Fold" "Call" "Call"
## 5  "Call" "Fold" "Call" "Fold" "Call" "Fold" "Fold" "Call" "Call" "Call"
## 6  "Call" "Fold" "Call" "Fold" "Fold" "Call" "Fold" "Call" "Fold" "Fold"
## 7  "Call" "Call" "Fold" "Fold" "Call" "Fold" "Fold" "Fold" "Fold" "Fold"
## 8  "Call" "Fold" "Call" "Fold" "Call" "Fold" "Fold" "Call" "Call" "Call"
## 9  "Call" "Call" "Call" "Fold" "Call" "Fold" "Fold" "Call" "Call" "Call"
## 10 "Call" "Call" "Call" "Call" "Call" "Call" "Call" "Call" "Call" "Call"
##    20
## 1  "Fold"
## 2  "Call"
## 3  "Call"
## 4  "Call"
```

```
## 5  "Fold"
## 6  "Fold"
## 7  "Call"
## 8  "Fold"
## 9  "Call"
## 10 "Call"
##
## , , s26
##
##    0      2      4      6      8      10     12     14     16     18
## 1  "Call" "Call" "Fold" "Fold" "Call" "Call" "Fold" "Call" "Fold" "Fold"
## 2  "Call" "Fold" "Fold" "Fold" "Call" "Call" "Call" "Fold" "Fold" "Call"
## 3  "Call" "Fold" "Fold" "Fold" "Fold" "Fold" "Fold" "Fold" "Call" "Fold"
## 4  "Call" "Call" "Call" "Fold" "Call" "Fold" "Call" "Call" "Fold" "Call"
## 5  "Call" "Fold" "Fold" "Call" "Call" "Fold" "Fold" "Call" "Fold" "Fold"
## 6  "Call" "Call" "Call" "Call" "Call" "Call" "Call" "Call" "Fold" "Fold"
## 7  "Call" "Call" "Fold" "Fold" "Call" "Call" "Call" "Call" "Call" "Call"
## 8  "Call" "Call" "Fold" "Call" "Call" "Fold" "Call" "Call" "Fold" "Fold"
## 9  "Call" "Call" "Call" "Fold" "Call" "Call" "Call" "Call" "Fold" "Call"
## 10 "Call" "Call" "Call" "Call" "Call" "Call" "Call" "Call" "Call" "Call"
##    20
## 1  "Call"
## 2  "Fold"
## 3  "Call"
## 4  "Call"
## 5  "Fold"
## 6  "Fold"
## 7  "Call"
## 8  "Call"
## 9  "Call"
## 10 "Call"
```