Toss Up! is a very simple dice game that I've always wanted to simulate but never got around to doing so (until now!). This post outlines how to simulate a Toss Up! game in R, as well as how to evaluate the effectiveness of different game strategies. All the code for this blog post is available here.

Rules

The official rules for Toss Up! are available here. Here is an abbreviated version:

- As shown in the picture above, the game uses 10 six-sided dice. Each die has 3 green sides, 2 yellow sides and one red side.
- For each turn, a player rolls all 10 dice. If any greens are rolled, set them aside.
    - At this point, the player can choose to end the turn and "bank" the points (one point for one green), or keep rolling the remaining dice.
    - After every roll, greens are set aside. If you roll enough times to make all 10 dice green, you have 10 points and you can either bank or roll all 10 dice again.
    - A player can keep going until he/she decides to stop rolling and bank all the points earned on this turn.
    - **Catch:** If, on a roll, none of the dice are green and at least one is red, the turn ends and no points can be banked.
- First player to reach 100 points wins.

Simulating Toss Up!

There are several different ways to implement Toss Up!: I describe my (two-player) version below. To allow for greater flexibility for the resulting code, I implement the game with three global constants that the programmer can change:

1. `NUM_DICE`: The number of dice used in the game (default is 10).
2. `FACE_PROBS`: A numeric vector of length 3 denoting the probability that a die comes up red, yellow or green respectively (default is `c(1/6, 2/6, 3/6)`).
3. `POINTS_TO_WIN`: The number of points a player needs to obtain to win the game (default is 100).

Step 1: How can we describe the state of a game?

The state of the game can be encapsulated with 4 pieces of information:

1. `current_player`: who's turn it is to make a decision.
2. `scores`: a vector of length 2 containing the scores for players 1 and 2 respectively.
3. `turn_points`: Number of points scored on the current turn so far (these points have not been banked yet).
4. `dice_rolls`: A vector of variable length denoting the outcome of the dice rolls (0: red, 1: yellow, 2: green).

In my implementation, the state is stored as a list with the 4 elements above.

Step 2: Updating the state

From a given state, there are 3 possibilities for what comes next:

1. The dice rolls don't have any greens and at least one red. In this case, the current players turn is over. We need to change `current_player`, reset `turn_points` to 0, and re-roll the dice (`NUM_DICE` of them).
2. The dice rolls either (i) have at least one green, or (ii) have no reds. In this case, the current player has a choice of what to do.
    1. If the player chooses to ***bank***, then we need to update `scores`, reset `turn_points` to 0, change `current_player` and re-roll the dice (`NUM_DICE` of them).
    2. If the player chooses to ***roll***, then we need to update `turn_points` and re-roll just the dice that were not green.

The function `updateState` below does all of the above. I have also added a `verbose` option which, if set to `TRUE`, prints information on the game state to the console. (The function `DiceToString` is a small helper function for printing out the dice rolls.)

```
DiceToString <- function(dice_rolls) {
  return(paste(sum(dice_rolls == 2), "Green,",
               sum(dice_rolls == 1), "Yellow,",
               sum(dice_rolls == 0), "Red"))
}

# Executes the current player's action and updates the state. Is also
used if
# no greens and at least one red is rolled.
UpdateState <- function(state, action, verbose = FALSE) {
  if (verbose) {
    cat("Current roll:", DiceToString(state$dice_rolls))
    if (action != "rolled at least 1 red and no green")
      cat(" (bank", state$turn_points + sum(state$dice_rolls == 2),
"pts?)",
          fill = TRUE)
    else
      cat("", fill = TRUE)
    cat(paste0("Player ", state$current_player, " ", action, "s"),
        fill = TRUE)
  }

  if (action == "bank") {
```

```
    # action = "bank": bank the points current player earned this turn,
then
    # re-roll the dice.
    state$scores[state$current_player] <- state$scores[state$current_
player] +
      state$turn_points + sum(state$dice_rolls == 2)
    state$turn_points <- 0
    state$dice_rolls <- sample(0:2, size = NUM_DICE, replace = TRUE,
                               prob = FACE_PROBS)
    state$current_player <- 3 - state$current_player
  } else if (action == "roll") {
    # action = "roll": add the green dice to points earned this turn,
then
    # re-roll the remaining dice.
    state$turn_points <- state$turn_points + sum(state$dice_rolls == 2)
    ndice <- sum(state$dice_rolls != 2)
    if (ndice == 0) ndice <- NUM_DICE
    state$dice_rolls <- sample(0:2, size = ndice, replace = TRUE,
                               prob = FACE_PROBS)
  } else if (action == "rolled at least 1 red and no green") {
    # action = "rolled at least 1 red and no green":
    # no points banked, turn ends, re-roll dice.
    state$turn_points <- 0
    state$dice_rolls <- sample(0:2, size = NUM_DICE, replace = TRUE,
                               prob = FACE_PROBS)
    state$current_player <- 3 - state$current_player
  } else {
    stop("action must be 'bank', 'roll', or 'rolled at least 1 red and
no green'")
  }

  if (verbose) {
    if (action != "roll") {
      cat("Current scores:", state$scores, fill = TRUE)
      cat("", fill = TRUE)
    }
  }

  return(state)
}
```

Step 3: How to express player behavior and strategy?

We can think of a player as a black box which takes a game state as an input and outputs a decision, “bank” or “roll”. In other words, the player is a function!

Below are two extremely simple strategies expressed as functions. The first strategy simply banks after the first roll. The second strategy banks once more than `target` points can be earned from the turn.

```
# strategy that stops after one roll
OneRoll <- function(state) {
  return("bank")
```

```
}

# strategy that stops only when the turn yields > `target` points
OverTargetPoints <- function(state, target = 10) {
  if (state$turn_points + sum(state$dice_rolls == 2) > target) {
    return("bank")
  } else {
    return("roll")
  }
}
```

(***Note:*** In my implementation, strategy functions should assume that they are `current_player`: that is how they can determine their current score and that of their opponent correctly.)

Step 4: Simulating a full game

We can now put the pieces from the previous steps together to simulate a full game of Toss Up!. The `SimulateGame` function takes in two strategy functions as input. It sets up the initial state, then updates the state repeatedly until the game ends.

```
SimulateGame <- function(Strategy1, Strategy2, verbose = FALSE) {
  # set up initial state
  state <- list(current_player = 1,
                scores = c(0, 0),
                turn_points = 0,
                dice_rolls = sample(0:2, size = NUM_DICE, replace =
TRUE,
                                    prob = FACE_PROBS))

  # check if no greens and at least one red, if so change player
  while (sum(state$dice_rolls == 2) == 0 && sum(state$dice_rolls == 0)
> 0) {
    state <- UpdateState(state, "rolled at least 1 red and no green",
verbose)
  }

  # while the game has not ended:
  # - get the next action from the current player's strategy
  # - update the state
  while (max(state$scores) < POINTS_TO_WIN) {
    if (state$current_player == 1) {
      action <- Strategy1(state)
    } else {
      action <- Strategy2(state)
    }
    state <- UpdateState(state, action, verbose)

    # check if no greens and at least one red
    while (sum(state$dice_rolls == 2) == 0 && sum(state$dice_rolls ==
0) > 0) {
      state <- UpdateState(state, "rolled at least 1 red and no green",
verbose)
```

```
    }
  }

  # game has ended: return winner
  if (verbose) {
    cat(paste("Game ends: Player", which.max(state$scores), "wins!"),
        fill = TRUE)
  }
  return(which.max(state$scores))
}
```

Two examples of simulated games

The code below shows what a simulated game of Toss Up! might look like. In the code snippet below, players need 20 points to win. The first player uses the super conservative strategy of banking immediately, while the second player tries to win the entire game in one turn.

```
NUM_DICE <- 10
FACE_PROBS <- c(1/6, 2/6, 3/6)
POINTS_TO_WIN <- 20
set.seed(1)
winner <- SimulateGame(OneRoll,
                       function(state) { OverTargetPoints(state, 19) },
                       verbose = TRUE)

# Current roll: 4 Green, 3 Yellow, 3 Red (bank 4 pts?)
# Player 1 banks
# Current scores: 4 0
#
# Current roll: 5 Green, 4 Yellow, 1 Red (bank 5 pts?)
# Player 2 rolls
# Current roll: 3 Green, 1 Yellow, 1 Red (bank 8 pts?)
# Player 2 rolls
# Current roll: 2 Green, 0 Yellow, 0 Red (bank 10 pts?)
# Player 2 rolls
# Current roll: 5 Green, 4 Yellow, 1 Red (bank 15 pts?)
# Player 2 rolls
# Current roll: 2 Green, 3 Yellow, 0 Red (bank 17 pts?)
# Player 2 rolls
# Current roll: 0 Green, 3 Yellow, 0 Red (bank 17 pts?)
# Player 2 rolls
# Current roll: 2 Green, 1 Yellow, 0 Red (bank 19 pts?)
# Player 2 rolls
# Current roll: 0 Green, 1 Yellow, 0 Red (bank 19 pts?)
# Player 2 rolls
# Current roll: 0 Green, 1 Yellow, 0 Red (bank 19 pts?)
# Player 2 rolls
# Current roll: 1 Green, 0 Yellow, 0 Red (bank 20 pts?)
# Player 2 banks
# Current scores: 4 20
#
# Game ends: Player 2 wins!
```

In this particular simulation, it paid off to be a bit more risk taking. As the simulation below shows, that's not always the case.

```
NUM_DICE <- 10
FACE_PROBS <- c(0.5, 0.0, 0.5)
POINTS_TO_WIN <- 20
set.seed(1)
winner <- SimulateGame(OneRoll,
                       function(state) { OverTargetPoints(state, 19) },
                       verbose = TRUE)

# Current roll: 6 Green, 0 Yellow, 4 Red (bank 6 pts?)
# Player 1 banks
# Current scores: 6 0
#
# Current roll: 5 Green, 0 Yellow, 5 Red (bank 5 pts?)
# Player 2 rolls
# Current roll: 2 Green, 0 Yellow, 3 Red (bank 7 pts?)
# Player 2 rolls
# Current roll: 0 Green, 0 Yellow, 3 Red
# Player 2 rolled at least 1 red and no greens
# Current scores: 6 0
#
# Current roll: 5 Green, 0 Yellow, 5 Red (bank 5 pts?)
# Player 1 banks
# Current scores: 11 0
#
# Current roll: 7 Green, 0 Yellow, 3 Red (bank 7 pts?)
# Player 2 rolls
# Current roll: 2 Green, 0 Yellow, 1 Red (bank 9 pts?)
# Player 2 rolls
# Current roll: 1 Green, 0 Yellow, 0 Red (bank 10 pts?)
# Player 2 rolls
# Current roll: 3 Green, 0 Yellow, 7 Red (bank 13 pts?)
# Player 2 rolls
# Current roll: 2 Green, 0 Yellow, 5 Red (bank 15 pts?)
# Player 2 rolls
# Current roll: 2 Green, 0 Yellow, 3 Red (bank 17 pts?)
# Player 2 rolls
# Current roll: 2 Green, 0 Yellow, 1 Red (bank 19 pts?)
# Player 2 rolls
# Current roll: 0 Green, 0 Yellow, 1 Red
# Player 2 rolled at least 1 red and no greens
# Current scores: 11 0
#
# Current roll: 5 Green, 0 Yellow, 5 Red (bank 5 pts?)
# Player 1 banks
# Current scores: 16 0
#
# Current roll: 4 Green, 0 Yellow, 6 Red (bank 4 pts?)
# Player 2 rolls
# Current roll: 4 Green, 0 Yellow, 2 Red (bank 8 pts?)
```

```
# Player 2 rolls
# Current roll: 1 Green, 0 Yellow, 1 Red (bank 9 pts?)
# Player 2 rolls
# Current roll: 0 Green, 0 Yellow, 1 Red
# Player 2 rolled at least 1 red and no greens
# Current scores: 16 0
#
# Current roll: 5 Green, 0 Yellow, 5 Red (bank 5 pts?)
# Player 1 banks
# Current scores: 21 0
#
# Game ends: Player 1 wins!
```

Comparing some simple strategies

We can't tell whether one strategy is better than another by looking at a single game since there is so much randomness involved. What we should do is simulate many games and see which strategy wins out over the long run.

OneRoll vs. OneRoll

Let's do a sanity check. Here, we run 10,000 games of the strategy `OneRoll` vs. itself (this takes around 4 seconds on my machine). Since the strategy for both players is the same, we might expect player 1 to win around 50% of the time right?

```
NUM_DICE <- 10
FACE_PROBS <- c(1/6, 2/6, 3/6)
POINTS_TO_WIN <- 100

nsim <- 10000
set.seed(1)
winners <- replicate(nsim, SimulateGame(OneRoll, OneRoll))
table(winners)  # not 50-50: player who starts first has an advantage

# winners
# 1    2
# 5910 4090
```

It looks like player 1 wins around 59% of the time! On second thought, we would expect player 1 to win more because he/she has a ***first mover advantage***: if player 1 reaches 100 points, he/she wins even if player 2 might reach 100 points on the very next turn.

To make this sanity check work, we should have the players each go first half the time. The code below achieves that:

```
# for even numbered simulations, let player 2 start first
winners2 <- winners
winners2[2 * 1:(nsim/2)] <- 3 - winners2[2 * 1:(nsim/2)]
table(winners2)

# winners2
# 1    2
# 5030 4970
```

Now player 1 wins just 50.3% of the time, which is much closer to the expected 50%.

OneRoll vs. >20 points

Next, let's compare the "one roll" strategy with the strategy which stops once the player can bank more than 20 points for the turn:

```
set.seed(1)
winners <- replicate(nsim, SimulateGame(
  OneRoll,
  function(state) { OverTargetPoints(state, 20) }))
table(winners)

# winners
# 1    2
# 75 9925
```

Wow! The strategy of banking only when >20 points have been earned on a turn wins almost all the time, even though it doesn't have the first mover advantage!

>20 points vs. >50 points

Taking a bit more risk than always banking seems to pay off. What about even more risk? How does banking after >20 points compare with banking after >50 points?

```
set.seed(1)
winners <- replicate(nsim, SimulateGame(
  function(state) { OverTargetPoints(state, 20) },
  function(state) { OverTargetPoints(state, 50) }))
table(winners)

# winners
# 1    2
# 6167 3833

# switch order of players
set.seed(1)
winners <- replicate(nsim, SimulateGame(
  function(state) { OverTargetPoints(state, 50) },
  function(state) { OverTargetPoints(state, 20) }))
table(winners)

# winners
# 1    2
# 4414 5586
```

The >20 points strategy won 61.7% of the games when it went first, and won 55.9% of the games when it went second, so it's clearly a superior strategy.

Where can we go from here?

There are several future directions we could take with this code base.