Qwixx - A Fast Family Dice Game

Stat 321 Project Report

Katherine Curro and Josie Peterburs
Fall 2024

Introduction

The objective of this project is to simulate the Qwixx Dice Game, a board game involving dice rolls and strategic decision-making. The goal is to score the most points by crossing out numbers in four color-coded rows while avoiding penalty points. The game ends when a player accumulates four penalty points or when two rows have been locked. This simulation aims to model the rules of the game, including the dice rolls, player decisions, and scoring system, to better understand the mechanics and potential strategies for winning the game.

The rules of Qwixx are as follows (from https://gamewright.com/pdfs/Rules/QwixxTM-RULES.pdf):

Goal: Score the most points by crossing out as many numbers in the four color-rows as possible while avoiding penalty points.

Contents: 6 dice (2 white, 1 red, 1 green, 1 blue, 1 yellow) and a score pad

The one basic rule of Qwixx is that numbers must be crossed out from left to right in each of the four color-rows. You do not have to begin with the number farthest to the left, but if you skip any numbers, they cannot be crossed out afterward.

How to Play: The first player to roll a 6 takes on the role of "active player." The active player rolls all six dice. The following two actions are now carried out in order, always one after the other:

1. The active player adds up the **two white dice** and calculates the resulting sum. All players may then (but are not required to) cross out the number in any (but only one) of the color-rows.
2. The active player (but not the others) may then (but is not required to) add **one of the white dice** together with any one of the colored dice and cross out the number corresponding to this sum in the color-row corresponding to the color of the chosen die.

Penalties: If, after the two actions, the active player doesn't cross out at least one number, they must cross out one of the penalty boxes. Each penalty box is worth -5 points at the end of the game. (The non-active players do not take a penalty if they choose not to cross out a number.)

Locking a Row: If you wish to cross out the number at the extreme right end of a color-row (red 12, yellow 12, green 2, blue 2) you must have first crossed out at least five numbers in that row. If you cross out the number on the extreme right, then also cross off the lock symbol directly next to it. This indicates that the color-row is now locked for all players and numbers of this color cannot be crossed out in future rounds. The die of the corresponding color is immediately removed from the game.

- **Notes:**
  - If a row is locked during the first action, it is possible that other players may, at the same time, also cross out the number on the extreme right and lock the same color-row. These players must also have previously crossed out at least five numbers in that row.
  - The cross on the lock counts toward the total number of crosses marked in that color-row.

Ending the Game: The game ends immediately as soon as either someone has marked a cross in his fourth penalty box or as soon as two dice have been removed from the game (two color-rows have been locked). It may occur (during the first action) that a third row is locked simultaneously with the second row.

Scoring: Beneath the four rows is a table indicating how many points are awarded for how many crosses within each row (including any locks marked with a cross). Each crossed out penalty box scores five minus-points. Enter your points for the four color-rows and the minus-points for any penalties in the appropriate fields at the bottom of the scoresheet. The player with the highest total score is the winner.

Methods

Setup

To model the Qwixx Dice Game, we first need to simulate the dice rolls. The game uses six dice (two white dice, one red, one yellow, one green, and one blue die). The white dice are rolled to calculate the sum, which determines the possible numbers players can cross off. Players also use the colored dice to cross off numbers in the corresponding color rows.

The game setup involves initializing a board for each player, where rows correspond to the four color-coded rows (red, yellow, green, and blue). The game progresses turn by turn, with each player making a move or taking a penalty. Players' moves depend on the roll of the dice and the state of their boards, and the game ends when one player hits four penalty points or when two rows have been locked.

Simulation Design

With the help of Dr. McShane, we simulate the game step by step, with the following components:

1. Dice Rolls: The white dice are rolled to generate a sum that can be used for the players' moves. The colored dice are used by the active player to cross out numbers in the corresponding rows.
2. Game State Tracking: The game state is tracked using a matrix for each player's board, where each position corresponds to a number that can be crossed off. The state also tracks penalty points and locked rows.
3. Player Strategies: Players are assumed to use a basic strategy where they cross off numbers in the leftmost available spots in the rows, unless no valid moves are available, in which case they take a penalty. We also defined our personal strategies for this game.
4. Game End Condition: The game ends when either two color rows are locked or when a player accumulates four penalty points.

The simulation runs through multiple iterations to calculate and analyze the game results. The code structure ensures that the simulation can be reproduced with any set of parameters, such as the number of players or specific strategies used.

Results

The game was simulated with 1-2 players. The game proceeded in turns, with the players rolling the dice, making moves, and tracking penalties. The following is a summary of the results:

- Total Rolls: The game typically lasted around 35–50 rolls depending on the game progress and player decisions.
- Locked Rows: On average, two rows were locked within 30 turns, with the active player's strategy being key to locking rows efficiently.
- Penalty Points: The frequency of penalty points varied based on the dice rolls and player decisions, with some players reaching penalties as early as the 20th turn.

Conclusions / Future Work

In this simulation study of Qwixx, we modeled the rules and mechanics to simulate gameplay. The results demonstrate how the randomness of dice rolls and strategic choices impact the progression of the game. The default strategy, which focuses on crossing out the leftmost available numbers, is effective in most scenarios, but players who take too long to lock rows or who accumulate too many penalties are at a disadvantage.

For future work, we could explore more sophisticated player strategies and how these strategies influence the probability of winning. Additionally, expanding the simulation to include more players could provide insights into multiplayer dynamics and strategies. A deeper statistical analysis, such as running a large number of simulations to compute average scores, could further explore the game's dynamics and identify optimal play strategies.

Appendix

Code

simulate normal dice roll

```{r}
roll <- function() {
  sample(1:6, 1)
}
```

qwixx role - dice\[i\] = 1 if its locked - color order: white, white, red, yellow, green, blue

```{r}
qwixxRoll <- function(dice, longout = FALSE) {
  q = replicate(n = 6, roll()) * c(1, 1, (1-dice))
  if (longout) print(paste("Qwixx Roll was", paste(q, collapse = " ")))
  return(q)
}
```

check if game is over

- two rows locked, or
- someone has 4 penalty points

```{r}
gameOver <- function(g, dice) {
  return(sum(dice) >= 2 | sum(g[["Penalty"]] >= 4) > 0)
}
```

game set up - make boards

```{r}
gameSetup <- function(players = 2) {

  # store both player's info
  g = list()
```

```
  # set penalties to zero
  g[["Penalty"]] = rep(0, players)

  # set up each board
    # use a matrix with with 4 rows (red, yellow, blue, green) and 13 columns (2-12 and a locked
column)
  g[["Board"]] = rep(list(matrix(rep(0, 52), nrow = 4, ncol = 13)), players)

  return(g)
}
```

change whose turn it is

```{r}
nextTurn <- function(turn, players) {
  return(((turn) %% players) + 1)
}
```

default strategy

- defaults to "penalty" only if there is no possible move
- picks a single move that places an "x" in the leftmost box (if there's a tie, draw the color at random)
- should never put two "x"s down on the same roll
- should only place an "x" while they're the active roller

```{r}
defaultStrategy <- function(board, db, q, p, active, longout) {

  if (active) {

    # find legal moves given board and roll
    rightXs = findRightXs(board)
    moves = possibleMoves(board, q, active)

    if (longout) {
      printMoves(moves)
    }
```

```
    # return no moves if none
    if (is.null(moves[[1]]) & is.null(moves[[2]]) & is.null(moves[[3]]) & is.null(moves[[4]])) {
      return(list("action" = list(c()), "db" = db))
    }

    # furthest left possible option from each row
    left = leftMostOptions(moves)

    # pick a move that places an "x" in the leftmost box
    # if tie, choose the color at random
    X = defaultPick(left)

    return(list("action" = X, "db" = db))

  } else {
    return(list("action" = list(c()), "db" = db))
  }
}
```

keep track of public rolls in db

```{r}
defaultStrategy_db <- function(board, db, q, p, active, longout) {

  # initialize p's db if needed
  # or increment the counter of whatever the public roll is
  if (is.null(db[[p]])){
    db[[p]] = rep(0,12)
  } else{
    db[[p]][q[1] + q[2]] = db[[p]][q[1] + q[2]] + 1
  }

  if (active) {

    # legal moves given board and roll
    rightXs = findRightXs(board)
    moves = possibleMoves(board, q, active)
```

```
    if (longout) {
      printMoves(moves)
    }

    # return no moves if none
    if (is.null(moves[[1]]) & is.null(moves[[2]]) & is.null(moves[[3]]) & is.null(moves[[4]])) {
      return(list("action" = list(c()), "db" = db))
    }

    # furthest left possible option from each row
    left = leftMostOptions(moves)

    # pick a move that places an "x" in the leftmost box
    # if tie, choose the color at random
    X = defaultPick(left)

    return(list("action" = X, "db" = db))

  } else {
    return(list("action" = list(c()), "db" = db))
  }
}
```

find furthest left move for each row

```{r}
leftMostOptions <- function(moves) {
  return(suppressWarnings(sapply(X = moves, FUN = min)))
}
```

find the furthest right X in each row for a certain board

```{r}
findRightXs <- function(board) {
  rows = c(0, 0, 0, 0)
  for (r in 1:4) {
    for (i in rev(2:13)) {
      if (board[r, i] == 1) {
```

```
      rows[r] = i
      break
    }
   }
  }
  return(rows)
}
```

pick a move that places an x in leftmost box

```{r}
defaultPick <- function(moves) {
  min = min(moves)
  options = which(sapply(moves, function(x) x == min))

  if (length(options) == 1) {
    return(list(c(options, min)))
  }
  else {
    return(list(c(sample(options, 1), min)))
  }
}
```

possible moves given a board and dice roll

```{r}
possibleMoves <- function(board, q, active) {
  # possible moves list
  moves = rep(list(c()), 4)

  # furthest right X for each row
  rightXs = findRightXs(board)

  # white dice options
  w = q[1] + q[2]
  for (r in 1:4) {
    if (r < 3) {
      if (allowed(w, r, q, board)) {
```

```
      moves[[r]][length(moves[[r]]) + 1] = w
    }
  } else {
    if (allowed(14 - w, r, q, board)) {
      moves[[r]][length(moves[[r]]) + 1] = 14 - w
    }
  }
 }

 # colored dice options
 if (active) {
  for (r in 1:4) {
    num1 = q[1] + q[2 + r]
    num2 = q[2] + q[2 + r]
    if (r < 3) {
      if (allowed(num1, r, q, board)) {
        moves[[r]][length(moves[[r]]) + 1] = num1
      }
    } else {
      if (allowed(14 - num1, r, q, board)) {
        moves[[r]][length(moves[[r]]) + 1] = 14 - num1
      }
    }

    if (r < 3) {
      if (allowed(num2, r, q, board)) {
        moves[[r]][length(moves[[r]]) + 1] = num2
      }
    } else {
      if (allowed(14 - num2, r, q, board)) {
        moves[[r]][length(moves[[r]]) + 1] = 14 - num2
      }
    }
  }
 }

 return(moves)
}
```

check if player is allowed to place an X in a given spot

```{r}
# check if player is allowed to place an X in a given spot
allowed <- function(index, r, q, board) {
  # right X in this row:
  rightXs = findRightXs(board)

  rowNotLocked  = q[2 + r] != 0
  spotValid     = rightXs[r] < index
  twelfthColumn = index == 12
  underFiveXs   = sum(board[r, ]) < 5

  # if row isn't locked & number is to the right of the furthest X
  spaceAvailable = rowNotLocked & spotValid

  # if placing in rightmost column, check at least 5 X's in that row. if yes, then lock
  wouldLock = !(twelfthColumn & underFiveXs)

  output = spaceAvailable & wouldLock
  return(output)
}
```

print possible moves given a vector of moves

```{R}
printMoves <- function(moves) {
  b = matrix(rep(0, 52), nrow = 4, ncol = 13)
  for (r in 1:4) {
    for (c in moves[[r]]) {
      b[r, c] = 1
    }
  }
  print("----------------Possible Moves----------------")
  print(b)
  print("----------------------------------------------")
}
```

print boards

```{r}
printBoards <- function(boards) {
  for (i in 1:length(boards)) {
    print(paste("----------------PLAYER", i, "BOARD----------------"))
    print(boards[[i]])
  }
}
```

calculate score for a given board

```{r}
countScore <- function(board, p) {

  # score key from bottom of board
  rubric = list("0" = 0, "1" = 1, "2" = 3, "3" = 6, "4" = 10, "5" = 15, "6" = 21,
                "7" = 28, "8" = 36, "9" = 45, "10" = 55, "11" = 66, "12" = 78)

  # calc
  score = sum(sapply(rowSums(data.frame(board)), function(x) rubric[[paste0(x)]])) - 5 * p

  return(score)
}
```

simulate a game

```{r}
simulateGame <- function(players = 2, strats = rep(0, players), longout = FALSE) {

  # keep track of number of rolls
  num_rolls <- 0

  # store each player's strategy
  db <- rep(list(c()), players)

  # randomly assign first turn
```

```
  turn = sample(c(1:players), 1)

  # set each dice to unlocked
  dice = c(FALSE, FALSE, FALSE, FALSE)

  # store all player's info
  g = gameSetup(players)

  # keep playing while game isn't over
  while (!gameOver(g, dice)) {

    if (longout) {
      print(paste("==========================PLAYER", turn,
"TURN==========================="))
      print('')
    }

    # roll dice
    q = qwixxRoll(dice, longout)
    num_rolls <- num_rolls + 1

    # coordinates of each X each player marks this round
    actions = rep(list(0), players)

    # give non-Active player(s) public roll
    for (p in 1:players) {
      if (p != turn) {
        outcome = play(p, g, q, strats, db, active = FALSE, longout)
        actions[[p]] = outcome$action
        db = outcome$db
      }
    }

    # active player goes
    outcome = play(turn, g, q, strats, db, active = TRUE, longout)
    actions[[turn]] = outcome$action
    db = outcome$db

    if (longout) {
      for (i in 1:length(actions)) {
```

```
    print(paste("Player", i, "Moves:", actions[[i]]))
  }
}

# keep track of rows that get locked this round
locks <- c(FALSE,FALSE,FALSE,FALSE)

# process actions from this turn
for (p in 1:players) {

  # pth player's moves for this turn
  Xs = actions[[p]][[1]]

  if( !(list(Xs) %in% legalAllMoves( g[["Board"]][[p]],q,  (p == turn) ) )){


    if (longout) {
      print("illegal move detected - skipping turn")
      print(list(Xs))
      print(deparse(g[["Board"]][[p]]))
      print(deparse(q))
      print(deparse(p == turn))
      print(deparse(p))
    }
    Xs =c()
  }

  # if active player had no moves, penalty
  if (is.null(Xs)) {
    if (p == turn) {
      g[["Penalty"]][p] = g[["Penalty"]][p] + 1
      if (longout) {
        print(paste("-------------------------PLAYER", p,
                    "PENALTY-------------------------"))
      }
    }
  } else {

    # mark down each X in the Xs vector (and lock if needed)
    ptr = 1
```

```
      while (ptr <= length(Xs)) {

        # coordinates of the X
        r = Xs[[ptr]]
        c = Xs[[ptr + 1]]

        # if locking a row, check its not locked and that row has 5 Xs before locking
        if (c == 12) {
          if (!dice[r] & sum(g[["Board"]][[p]][r, ]) >= 5) {
            locks[r] = TRUE
            g[["Board"]][[p]][r, c] = 1
            g[["Board"]][[p]][r, c+1] = 1
            if (longout) {
              print(paste("-------------------------PLAYER", p,
                          "LOCKING", r, "-------------------------"))
            }
          }
        } else {
          g[["Board"]][[p]][r, c] = 1
        }
        ptr = ptr + 2
      }
    }
  }

  # lock rows from this turn
  for (l in 1:4){
    if (locks[l]) dice[l] = TRUE
  }

  # next player's turn
  turn = nextTurn(turn, players)

  if (longout) {
    printBoards(g[["Board"]])
  }
}

if (longout) {
  print("GAME OVER")
```

```
  }

  # calculate final scores
  scores = 1:players
  for (i in 1:players) {
    scores[i] = countScore(g[["Board"]][i], g[["Penalty"]][i])
  }

  if (longout) {
    print("Final Scores:")
    print(scores)
  }

  # return boards and scores
  return(list(g[['Board']], scores, num_rolls))
}
```

```{r}
legalAllMoves <- function(board, q, active) {
  # list of possible moves
  moves = legalPublicMoves(board, q)
  if (active){
    move_after = function(m){
      if (length(m) == 0) {
        b2 = board
      }else{
        b2 = makeMove(board , m[1],m[2],q)
      }
      legalColoredMoves(b2, q)
    }

    moves = unlist(lapply(moves, function(m) lapply(move_after(m), function(m2) c(m, m2) ) ),
recursive =  FALSE)
  }
  return(moves)
}


legalPublicMoves <- function(board, q) {
```

```
  # list of possible moves
  moves = list(c())

  # furthest right X in each row
  rightXs = findRightXs(board)

  # options from white dice
  w = q[1] + q[2]
  for (r in 1:4) {
    if (r < 3) {
      if (allowed(w, r, q, board)) {
        moves = append(moves, list(c(r, w)))
      }
    } else {
      if (allowed(14 - w, r, q, board)) {
        moves = append(moves, list(c(r, 14-w)))
      }
    }
  }
  return(moves)
}

legalColoredMoves <- function(board, q) {
  # list of possible moves
  moves = list(c())

  # furthest Right X in each row
  rightXs = findRightXs(board)


  for (r in 1:4) {
    num1 = q[1] + q[2 + r]
    num2 = q[2] + q[2 + r]
    if (r < 3) {
      if (allowed(num1, r, q, board)) {
        moves = append(moves, list(c(r, num1)))
      }
    } else {
      if (allowed(14 - num1, r, q, board)) {
        moves = append(moves, list(c(r, 14-num1)))
```

```
    }
   }

   if (r < 3) {
    if (allowed(num2, r, q, board)) {
     moves = append(moves, list(c(r, num2)))
    }
   } else {
    if (allowed(14 - num2, r, q, board)) {
     moves = append(moves, list(c(r, 14-num2)))
    }
   }
  }

  return(moves)
}
```

Make a move

- if locking a row, check its not locked and that they have 5 Xs before locking

```{r}
makeMove = function(board, r, c, dice){
  if (c == 12) {
    if (!dice[r] & sum(board[r, ]) >= 5) {
      board[r, c] = 1
      board[r, c+1] = 1
    }
  } else {
    board[r, c] = 1
  }
  return(board)
}
```

```{r}
play <- function(p, g, q, strats, db, active, longout = FALSE) {
```

```
  # player's board from game info dictionary
  board = g[["Board"]][[p]]

  # use strats key to see what strategy player is playing with
  strategy = strats[p]

  # strategy = 0 is default strategy
  if(strategy == 0) {return(defaultStrategy_db(board, db, q, p, active, longout))}
  # strategy = 1 is josie's strategy
  else if (strategy == 1) {return(josieStrat(board, db, q, p, active, longout))}
  # strategy 2 is cali's
  else if (strategy == 2) {return(caliStrat(board, db, q, p, active, longout))}
  # strategy 3 is best
  else if (strategy == 3) {return(best(board, db, q, p, active, longout))}
  # else do best strat
  else {return(defaultStrategy(board, db, q, p, active, longout))}
}
```

Trying Different Strategies for One Player Games

```{r}
# repeat 1000 1-player games with default strategy
nsim = 1000
results = 1:nsim
num_rolls_game = 1:nsim
for (i in results) {
  results[i] = simulateGame(players = 1, strats = c(0))[[2]]
  num_rolls_game[i] = simulateGame(players = 1, strats = c(0))[[3]]
}

# expected score
e <- mean(results)
print(paste("Expected Score: ", e))

# expected number of rolls
rolls <- mean(num_rolls_game)
print(paste("Expected Number of Rolls: ", rolls))
```

```
# distribution of score
hist(results, xlab = "Qwixx Score", main = "Distribution Scores: Default Strategy, 1 Player")

# distribution of rolls
hist(num_rolls_game, xlab = "# Rolls", main = "Distribution of Rolls: 1 Player")
```