Probability Bingo

From Brian Mehmed, 2011 WV APSI

Project Report

Minhaj Fahad, Eddie Ramirez, Oluwasola Ogundare

Cornell University
Operations Research and Informational Engineering Department

Cornell Engineering

Operations Research and Informational Engineering
Cornell University
https://www.orie.cornell.edu/orie

Title:
Probability Bingo

Theme:
ENGRD 2700

Project Period:
Fall Semester 2023

Project Group:
XXX

Participant(s):
Eddie Ramirez
Minhaj Fahad
Oluwasola Ogundare

Supervisor(s):
Professor Jamol Pender

Copies: 1

Page Numbers: 27

Date of Completion:
Novemeber 30, 2023

Abstract:

Each of two die has colored faces, 3 green, 2 blue and 1 red. The two dice will be rolled. The outcome will be considered to be one "bingo call." If you have this outcome on your bingo card, mark it off. The winner will be the student who gets a bingo card completely marked off (all 25 squares). We wish to find the optimal way to mark each square on the bingo card (use BG for "blue green," BB for "blue blue," etc.) so that we have the best chance of winning.

Contents

Preface

We will explore the optimal bingo strategy for nxn bingo boards for $\forall n \in [5]$

Cornell University, November 30, 2023

Minhaj Fahad
<msf257@cornell.edu>

Eddie Ramirez
<ebr66@cornell.edu>

Oluwasola Ogundare
<odo5@cornell.edu>

Chapter 1

Introduction

Here we will explore important probability that will be used in the rest of the report. 2.

1.1 Bingo

Bingo is a game of probability in which players mark off numbers on cards as the numbers are drawn randomly by a caller, the winner being the first person to mark off all their numbers. In this case, the numbers will be colored pairs based on the colors of the two dice rolled.

1.2 Colored Dice Probabilities

If we assume each dice is fair, then the probability of any color being rolled is equally likely, i.e. $\frac{1}{6}$. We also note that the distinction between dice 1 and dice 2 does not matter. That means that getting a "Green-Red" roll is analogous to saying the first roll was green and the second was red, or the first roll was red and the second roll was green.

The probabilities of each color on a single die are:

$$P(\text{Green}) = \frac{3}{6} = \frac{1}{2},$$
$$P(\text{Blue}) = \frac{2}{6} = \frac{1}{3},$$
$$P(\text{Red}) = \frac{1}{6}.$$

The total number of combinations for two dice rolls is as follows:

$$\text{Total combinations} = 6 \times 6 = 36$$

Probabilities for each pair of rolls:

$$P(\text{Green-Green}) = \left(\frac{3}{6}\right) \times \left(\frac{3}{6}\right) = \frac{9}{36}$$

$$P(\text{Green-Blue}) = 2 \times \left(\frac{3}{6}\right) \times \left(\frac{2}{6}\right) = \frac{12}{36}$$

$$P(\text{Green-Red}) = 2 \times \left(\frac{3}{6}\right) \times \left(\frac{1}{6}\right) = \frac{6}{36}$$

$$P(\text{Blue-Blue}) = \left(\frac{2}{6}\right) \times \left(\frac{2}{6}\right) = \frac{4}{36}$$

$$P(\text{Blue-Red}) = 2 \times \left(\frac{2}{6}\right) \times \left(\frac{1}{6}\right) = \frac{4}{36}$$

$$P(\text{Red-Red}) = \left(\frac{1}{6}\right) \times \left(\frac{1}{6}\right) = \frac{1}{36}$$

1.3 1x1 Bingo Board

Given a 1×1 bingo board, we win the game when the roll of two dice is equal to the color pair that we place. It is trivial to see that the most optimal choice is the two colored combination with the highest probability. From above, we can see that this choice is simply Green-Blue. A common mistake is to choose Green-Green, but the problem is that it forces the first dice to be green and if it is not, then the second roll will never matter. By choosing Green-Blue, we can get a green or a blue on the first roll and then let the second roll be the color missing. Because of this, the most optimal choice is a Green-Blue placement on a 1x1 Board.

Chapter 2

2x2 Bingo Board

2.1 Introduction

Consider now a 2×2 bingo board where we want the most optimal configuration of the bingo board. Before we consider the entire 2×2 case, consider the case of a 2x1 bingo board. That is, we wish to find the most optimal placement for a 2×1 bingo board given 6 different colored pairs to choose from.

2.2 2x1 Dice Probabilities

We now consider the probabilities of all possible placements given 6 colored pairs. In the given 6x6 matrix below, let $B_{i,j}$ represent a multi set of two color pairs. We let each row and each column represent the a selection from the list of possible pairs: (GG, GB, GR, BB, BR, RR). This results in a symmetric matrix which denotes the probability of getting the $B_{i,j}$ pair in two attempts of two dice rolls. For example, $B_{1,3}$ represents the probability of getting the multi-set {(Green-Green), (Green-Red)} in two dice rolls, which is simply the probability of Green-Green, times the probability of Green-Red, times 2 since the order we see each pair does not matter.

$$\begin{pmatrix} \frac{81}{1296} & \frac{216}{1296} & \frac{108}{1296} & \frac{72}{1296} & \frac{72}{1296} & \frac{18}{1296} \\ \frac{216}{1296} & \frac{144}{1296} & \frac{144}{1296} & \frac{96}{1296} & \frac{96}{1296} & \frac{24}{1296} \\ \frac{108}{1296} & \frac{144}{1296} & \frac{36}{1296} & \frac{48}{1296} & \frac{48}{1296} & \frac{12}{1296} \\ \frac{72}{1296} & \frac{96}{1296} & \frac{48}{1296} & \frac{16}{1296} & \frac{32}{1296} & \frac{8}{1296} \\ \frac{72}{1296} & \frac{96}{1296} & \frac{48}{1296} & \frac{16}{1296} & \frac{32}{1296} & \frac{8}{1296} \\ \frac{18}{1296} & \frac{24}{1296} & \frac{12}{1296} & \frac{8}{1296} & \frac{8}{1296} & \frac{1}{1296} \end{pmatrix}$$

We can see that the most likely pair of dice rolls is actually Green-Blue and Green-Green. Thus, it would follow that the optimal placement for a 2×1 bingo board is Green-Green, Green-Blue.

2.3 2x2 Optimal Placement

Now we consider how we can use this optimal placement for a 2×1 board for the 2×1 board. Given that bingo can be won given a collection of wins across the horizontal, vertical, or diagonal, we wish to place another 2×1 tile such that our chances of wining are maximized given a pre-existing placement of Green-Green, Green-Blue. We now reconsider the probabilities of winning in the bottom 2×1 tile given that the top 2×1 tile is Green-Green and Green-Blue. From oberserving the previous calculations, intuition would tell us that the optimal 2×1 bottom tile has a red, since given any red roll, we could still place a tile to progress towards "BINGO". Given that only one red occurrence is needed, one would bet that the best placement is Green-Red and Blue-Red, since these are the most likely rolls which contain a red. So the optimal configuration for a 2×2 board would be:

$$\begin{pmatrix} \textit{Green-Green} & \textit{Green-Blue} \\ \textit{Green-Red} & \textit{Blue-Red} \end{pmatrix}$$

2.4 Confirmation Simulation

In order to test our assumption, we now simulate the average number of turns needed to win a 2×2 board bingo game given all possible arrangements the board can take. With 4 tiles, each with 6 choices, there are 6^4 total bingo board arrangements. For each board, we simulated the average number of dice to be rolled in order for the bingo board to get "BINGO". From there, we ranked each boards' performance by the average turns taken.

2.5 Python Code for Simulation

```
import numpy as np
import random
import itertools
import math

random.seed(48)
```

```
# Creating list for all the possible rolls. Occurences in list
   represent probabilities of being selected
possible_rolls = ["GG"] * 9 + ["GB"] * 12 + ["GR"] * 6 + ["BB"]
    * 4 + ["BR"] * 4 + ["RR"]
rolls = set(possible_rolls)

# Create tuples of all 1296 board configurations
b = list(itertools.product(itertools.product(rolls, repeat = 2)
   , repeat = 2))
boards = {str(i + 1) : [v[0][0], v[0][1], v[1][0], v[1][1]] for
    i, v in enumerate(b)}

# Calculate how many turns it takes for the board to get BINGO
def check_wins(board):
    successes = 0
    tries = 0
    while successes != 2:
        tries += 1
        pick = random.choice(possible_rolls)
        if pick in board:
            successes += 1
            board.remove(pick)
        if tries > 20:
            return tries
    return tries

avg_turns = []
for i in boards:
    # Simulate games played 100,000 times for each board and
       append it's average
    avg_wins = [check_wins(boards[i][:]) for j in range(100000)
       ]
    avg_wins = sum(avg_wins) / len(avg_wins)
    avg_turns.append((avg_wins, boards[i]))
avg_turns.sort(key = lambda x : x[0])

# Output top 5 boards
avg_turns[:5]
```

2.6 Analyzing Results

The output of the function is the following:

```
[(2.8115, ['GR', 'BR', 'GG', 'GB']),
 (2.8119, ['GG', 'BR', 'GB', 'GR']),
 (2.81343, ['GB', 'GR', 'GG', 'BR']),
```

```
 (2.81411, ['GR', 'GB', 'BB', 'GG']),
 (2.81494, ['BB', 'GB', 'GG', 'GR'])]
```

As per our guess, the most optimal board configurations contains a row of Green-Red, Blue-Red, and another row of Green-Green, Green-Blue. On average it takes approximately 2.8 turns for the board to get "BINGO". We will now use these results in an attempt to solve the optimal placement for larger boards.

Chapter 3

3x3 Bingo Board

3.1 Proof by Inducation

Before analyzing the 3×3 board, we will provide a brief proof by induction. This proof wishes to show that the optimal configuration for a 3×3 board must include the optimal configuration for a 2×2 board as well.

Base Case: For a 2×2 bingo board, the optimal configuration is a Green-Blue tile, as it has the highest probability of winning based on the dice roll probabilities. This has been previously established.

Inductive Hypothesis: Assume that for a $k \times k$ bingo board (where k $\geq$ 1), the optimal configuration includes the optimal configuration of the $(k-1) \times (k-1)$ board. This means that the best strategy for a k×k board starts with the optimal arrangement for the $(k-1) \times (k-1)$ board and extends it in a way that maximizes the probability of getting "BINGO".

Inductive Step: Consider a $k \times k$ bingo board. We know from our hypothesis that the optimal configuration for a $(k-1) \times (k-1)$ board is the starting point for the $k \times k$ board. The optimal (k-1)×(k-1) configuration maximizes the probability of winning based on the dice probabilities. To extend this to the $k \times k$ board, we must fill the additional space in a way that further increases the chances of achieving "BINGO". This involves strategically placing the remaining tiles in the new rows, columns, and diagonals. The placement should consider the probabilities of dice rolls to maximize the chances of winning. By this logic, the optimal configuration for the $k \times k$ board would build upon the $(k-1) \times (k-1)$ arrangement and extend it to the new squares.

By induction, we can conclude that the optimal configuration of a $k \times k$ bingo board will contain the optimal configuration of the $(k-1) \times (k-1)$ board, extended in such a way as to maximize the chances of achieving "BINGO" on the larger board.

3.2 Simulation

Given the above assumption, we first construct an optimal 2×2 board and then let that be the "top-left" of a new 3×3 bingo board. That means we are looking for the optimal choice for $B_{1,3}$, $B_{2,3}$, $B_{3,1}$, $B_{3,2}$, $B_{3,3}$ where B is the 3×3 matrix corresponding to the choices on each grid on the bingo board. Given these five spots which each have 6 potential choices, we create a simulation of all 6^5 possibilities.

3.3 Python Code for Simulation

```
import itertools
import random
from copy import deepcopy

# Start with optimal tiling for a 2x2 bingo board
OPTIMAL_BOARD = [["GG", "GB", ""],
                 ["GR", "BR", ""],
                 ["", "", ""]]

# Creating list for all the possible rolls. Occurences in list
   represent probabilities of being selected
possible_rolls = ["GG"] * 9 + ["GB"] * 12 + ["GR"] * 6 + ["BB"]
    * 4 + ["BR"] * 4 + ["RR"]
rolls = set(possible_rolls)

# Get all possible combinations for the last 5 cells
b = list(itertools.product(rolls, repeat = 5))
boards = {str(i + 1) : None for i in range(len(b))}
index = 1
for vals in b:
    y = list(vals)
    board = deepcopy(OPTIMAL_BOARD)
    board[0][2] = y.pop()
    board[1][2] = y.pop()
    board[2][0] = y.pop()
    board[2][1] = y.pop()
    board[2][2] = y.pop()
    boards[str(index)] = deepcopy(board)
    index += 1

# Function to check if board has bingo, X's in the rows, cols,
   on on diagonals
def check_bingo(board):
    for row in range(len(board)):
```

```
        if board[row][0] == "X" and board[row][1] == "X" and
            board[row][2] == "X":
            return True
        if board[0][row] == "X" and board[1][row] == "X" and
            board[2][row] == "X":
            return True
    if board[0][0] == "X" and board[1][1] == "X" and board
        [2][2] == "X":
        return True
    if board[0][2] == "X" and board[1][1] == "X" and board
        [2][0] == "X":
        return True
    return False

# Checks to see if the current tile picked can be placed on
    bingo board
def check_placement(board, pick):
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == pick:
                board[i][j] = "X"
                return

# Simulation to take turns for the game
def take_turns(board):
    tries = 0
    while not check_bingo(board):
        tries += 1
        pick = random.choice(possible_rolls)
        check_placement(board, pick)
        if tries > 50:
            return tries
    return tries

# Do 100,000 simulations for each board and calculate average
    turns taken
avg_turns = []
for i in boards:
    avg_wins = [take_turns(deepcopy(boards[i])) for j in range
        (1000)]
    avg_wins = sum(avg_wins) / len(avg_wins)
    avg_turns.append((avg_wins, boards[i]))
avg_turns.sort(key = lambda x : x[0])

# Output top 10 boards
avg_turns[:10]
```

3.4 Analyzing Results

The output of the function is the following:

```
[(4.942, [['GG', 'GB', 'GB'], ['GR', 'BR', 'GB'], ['GG', 'BR',
    'BB']]),
 (4.956, [['GG', 'GB', 'GB'], ['GR', 'BR', 'GR'], ['GG', 'BB',
    'BB']]),
 (4.992, [['GG', 'GB', 'GB'], ['GR', 'BR', 'GR'], ['GG', 'BR',
    'BB']]),
 (5.015, [['GG', 'GB', 'GB'], ['GR', 'BR', 'BR'], ['BB', 'BR',
    'GG']]),
 (5.015, [['GG', 'GB', 'GB'], ['GR', 'BR', 'GR'], ['BB', 'BR',
    'GG']]),
 (5.015, [['GG', 'GB', 'GB'], ['GR', 'BR', 'BB'], ['GG', 'BR',
    'BB']]),
 (5.016, [['GG', 'GB', 'GB'], ['GR', 'BR', 'GR'], ['GG', 'BB',
    'RR']]),
 (5.03, [['GG', 'GB', 'GG'], ['GR', 'BR', 'BR'], ['BB', 'GB', '
    GB']]),
 (5.033, [['GG', 'GB', 'GB'], ['GR', 'BR', 'BR'], ['BB', 'RR',
    'GG']]),
 (5.042, [['GG', 'GB', 'GB'], ['GR', 'BR', 'BB'], ['GG', 'BR',
    'GR']])]
```

From our simulation, we can see that the most optimal board is as follows:

$$\begin{pmatrix} \textit{Green-Green} & \textit{Green-Blue} & \textit{Green-Blue} \\ \textit{Green-Red} & \textit{Blue-Red} & \textit{Green-Blue} \\ \textit{Green-Green} & \textit{Blue-Red} & \textit{Blue-Blue} \end{pmatrix}$$

Upon initial inspection, it would make sense that the most optimal additions to the board would be Green-Blue and Green-Green, but also Blue-Blue. Even though Blue-Blue is not the most probable roll, given that about 5 rolls are needed to get BINGO, we would add it to the board to ensure that if we observe this unlikely roll we can still add to our overall board. Given this, for the most optimal 4×4 board and 5×5 board one would expect that although very unlikely, RR will appear as a tile.

3.5 Sanity Check

The worst ten boards are the following:

```
avg_turns[-10:]
```

```
[(13.869, [['GG', 'GB', 'RR'], ['GR', 'BR', 'BR'], ['BR', 'RR',
    'RR']]),
 (13.894, [['GG', 'GB', 'RR'], ['GR', 'BR', 'RR'], ['RR', 'RR',
     'BR']]),
 (13.931, [['GG', 'GB', 'RR'], ['GR', 'BR', 'BR'], ['RR', 'BR',
     'RR']]),
 (13.993, [['GG', 'GB', 'RR'], ['GR', 'BR', 'BR'], ['BR', 'BR',
     'BR']]),
 (14.049, [['GG', 'GB', 'RR'], ['GR', 'BR', 'BR'], ['RR', 'BR',
     'BR']]),
 (14.076, [['GG', 'GB', 'RR'], ['GR', 'BR', 'RR'], ['BR', 'BR',
     'RR']]),
 (14.128, [['GG', 'GB', 'RR'], ['GR', 'BR', 'BR'], ['RR', 'RR',
     'BR']]),
 (14.16, [['GG', 'GB', 'RR'], ['GR', 'BR', 'BR'], ['RR', 'RR',
    'RR']]),
 (14.375, [['GG', 'GB', 'RR'], ['GR', 'BR', 'RR'], ['BR', 'RR',
     'RR']]),
 (28.706, [['GG', 'GB', 'RR'], ['GR', 'BR', 'RR'], ['RR', 'RR',
     'RR']])]
```

As expected, the worst boards are just numerous placements of RR and BR, which have the lowest probability of being selected. Furthermore, since they are all similar, we know this also minimizes the likelihood of the board getting bingo faster as well.

Chapter 4

4x4 Bingo Board

4.1 Board Construction

We first construct an optimal 3×3 board and then let that be the "top-left" of a new 4×4 bingo board. That means we are looking for the optimal choice for $B_{1,4}$, $B_{2,4}$, $B_{3,4}$, $B_{4,1}$, $B_{4,2}$, $B_{4,3}$, $B_{4,4}$ where B is the 4×4 matrix corresponding to the choices on each grid on the bingo board. Given these seven spots which each have 6 potential choices, we create a simulation of all 6^7 possibilities.

4.2 Python Code for Simulation

```
import itertools
import random
import numpy as np
from copy import deepcopy

random.seed(48)

# # Start with optimal tiling for a 3x3 bingo board
OPTIMAL_BOARD = [["GG", "GB", "GB", ""],
                 ["GR", "BR", "GB", ""],
                 ["GG", "BR", "BB", ""],
                 ["", "", "", ""]]

# Creating list for all the possible rolls. Occurences in list
   represent probabilities of being selected
possible_rolls = ["GG"] * 9 + ["GB"] * 12 + ["GR"] * 6 + ["BB"]
    * 4 + ["BR"] * 4 + ["RR"]
rolls = set(possible_rolls)
# Get all possible combinations for the last 7 cells
b = list(itertools.product(rolls, repeat = 7))
```

```
boards = {str(i + 1) : None for i in range(len(b))}

index = 1
# Fill in remaining 7 spots on the board
for vals in b:
    board = deepcopy(OPTIMAL_BOARD)
    board[0][3] = vals[0]
    board[1][3] = vals[1]
    board[2][3] = vals[2]
    board[3][0] = vals[3]
    board[3][1] = vals[4]
    board[3][2] = vals[5]
    board[3][3] = vals[6]

    boards[str(index)] = deepcopy(board)
    index += 1

# Function to check if board has bingo, X's in the rows, cols,
   on on diagonals
def check_bingo(board):
    for row in range(len(board)):
        if board[row][0] == "X" and board[row][1] == "X" and
           board[row][2] == "X" and board[row][3] == "X":
            return True
        if board[0][row] == "X" and board[1][row] == "X" and
           board[2][row] == "X" and board[3][row] == "X":
            return True
    if board[0][0] == "X" and board[1][1] == "X" and board
       [2][2] == "X" and board[3][3] == "X":
        return True
    if board[0][3] == "X" and board[1][2] == "X" and board
       [2][1] == "X" and board[3][0] == "X":
        return True
    return False

# Checks to see if the current tile picked can be placed on
   bingo board
def check_placement(board, pick):
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == pick:
                board[i][j] = "X"
                return

# Simulation to take turns for the game
def take_turns(board):
    tries = 0
```

```
    while not check_bingo(board):
        tries += 1
        pick = random.choice(possible_rolls)
        check_placement(board, pick)
        if tries > 20:
            return tries
    return tries

# Do 10,000 simulations for each board and calculate average
   turns taken
avg_turns = []
for i in boards:
    avg_wins = [take_turns(deepcopy(boards[i])) for j in range
       (10000)]
    avg_wins = sum(avg_wins) / len(avg_wins)
    avg_turns.append((avg_wins, boards[i]))
avg_turns.sort(key = lambda x : x[0])

# Output top 10 boards
avg_turns[:10]
```

4.3 Analyzing Results

The output of the function is the following:

```
[(5.96,
  array([['GG', 'GB', 'GB', 'GR'],
         ['GR', 'BR', 'GB', 'BB'],
         ['GG', 'BR', 'BB', 'BR'],
         ['GG', 'BB', 'GR', 'BR']], dtype='<U2')),
 (6.04,
  array([['GG', 'GB', 'GB', 'GR'],
         ['GR', 'BR', 'GB', 'GR'],
         ['GG', 'BR', 'BB', 'RR'],
         ['GG', 'BB', 'GB', 'BR']], dtype='<U2')),
 (6.04,
  array([['GG', 'GB', 'GB', 'GB'],
         ['GR', 'BR', 'GB', 'GB'],
         ['GG', 'BR', 'BB', 'BB'],
         ['RR', 'RR', 'GR', 'GG']], dtype='<U2')),
 (6.08,
  array([['GG', 'GB', 'GB', 'GR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'GG'],
         ['RR', 'RR', 'GB', 'RR']], dtype='<U2')),
 (6.12,
```

```
 array([['GG', 'GB', 'GB', 'GR'],
        ['GR', 'BR', 'GB', 'RR'],
        ['GG', 'BR', 'BB', 'RR'],
        ['BB', 'BR', 'GB', 'GG']], dtype='<U2')),
(6.2,
 array([['GG', 'GB', 'GB', 'GB'],
        ['GR', 'BR', 'GB', 'GB'],
        ['GG', 'BR', 'BB', 'BB'],
        ['GG', 'GR', 'RR', 'GG']], dtype='<U2')),
(6.2,
 array([['GG', 'GB', 'GB', 'BB'],
        ['GR', 'BR', 'GB', 'GR'],
        ['GG', 'BR', 'BB', 'GR'],
        ['GG', 'RR', 'GR', 'GG']], dtype='<U2')),
(6.24,
 array([['GG', 'GB', 'GB', 'GR'],
        ['GR', 'BR', 'GB', 'BB'],
        ['GG', 'BR', 'BB', 'RR'],
        ['GB', 'BR', 'GG', 'GG']], dtype='<U2')),
(6.24,
 array([['GG', 'GB', 'GB', 'GG'],
        ['GR', 'BR', 'GB', 'GB'],
        ['GG', 'BR', 'BB', 'RR'],
        ['GG', 'GR', 'BB', 'GR']], dtype='<U2')),
(6.28,
 array([['GG', 'GB', 'GB', 'GB'],
        ['GR', 'BR', 'GB', 'GB'],
        ['GG', 'BR', 'BB', 'GB'],
        ['GR', 'BR', 'BB', 'GB']], dtype='<U2'))]
```

From our simulation, we can see that the most optimal board is as follows:

$$\begin{pmatrix} \textit{Green-Green} & \textit{Green-Blue} & \textit{Green-Blue} & \textit{Green-Red} \\ \textit{Green-Red} & \textit{Blue-Red} & \textit{Green-Blue} & \textit{Blue-Blue} \\ \textit{Green-Green} & \textit{Blue-Red} & \textit{Blue-Blue} & \textit{Blue-Red} \\ \textit{Green-Green} & \textit{Blue-Blue} & \textit{Green-Red} & \textit{Blue-Red} \end{pmatrix}$$

From our previous observations, we see that the unlikely roll of Blue-Red is now an optimal choice for the construction of the board. Additionally, we can see that although they were not the top board, there are some in the top 10 which do have Red-Red. Given this, we expect the most optimal 5x5 board to have at least one instance of a Red-Red tile.

4.4 Sanity Check

The worst ten boards are the following:

```
[(16.24,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'RR'],
         ['RR', 'RR', 'BR', 'BR']], dtype='<U2')),
 (16.36,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'RR'],
         ['BR', 'RR', 'BR', 'RR']], dtype='<U2')),
 (16.36,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'RR'],
         ['RR', 'RR', 'RR', 'RR']], dtype='<U2')),
 (16.4,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'BR'],
         ['RR', 'BR', 'RR', 'BR']], dtype='<U2')),
 (16.48,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'BR'],
         ['RR', 'BR', 'BR', 'BR']], dtype='<U2')),
 (16.56,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'RR'],
         ['RR', 'BR', 'BR', 'BR']], dtype='<U2')),
 (16.72,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'BB'],
         ['BR', 'BR', 'RR', 'RR']], dtype='<U2')),
 (17.04,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
         ['GG', 'BR', 'BB', 'BR'],
         ['BR', 'BR', 'BR', 'RR']], dtype='<U2')),
 (17.08,
  array([['GG', 'GB', 'GB', 'RR'],
         ['GR', 'BR', 'GB', 'RR'],
```

```
        ['GG', 'BR', 'BB', 'BR'],
        ['RR', 'BR', 'RR', 'RR']], dtype='<U2')),
(17.56,
 array([['GG', 'GB', 'GB', 'RR'],
        ['GR', 'BR', 'GB', 'RR'],
        ['GG', 'BR', 'BB', 'BR'],
        ['BR', 'BR', 'BR', 'BR']], dtype='<U2'))]
```

As expected, the worst boards are just numerous placements of RR and BR, which have the lowest probability of being selected. Furthermore, since they are all almost identical in how the dice have to be rolled, we know this also minimizes the likelihood of the board getting bingo faster as well.

Chapter 5

5x5 Bingo Board

5.1 Board Construction

We first construct an optimal 4×4 board and then let that be the "top-left" of a new 5×5 bingo board. That means we are looking for the optimal choice for $B_{1,5}$, $B_{2,5}$, $B_{3,5}$, $B_{4,5}$, $B_{5,1}$, $B_{5,2}$, $B_{5,3}$, $B_{5,4}$, and $B_{5,5}$ where B is the 5×5 matrix corresponding to the choices on each grid on the bingo board. Given these seven spots which each have 6 potential choices, there are 6^9 $(10,077,696)$ possible bingo boards.

5.2 Configuration Patterns

Given that the total board combinations is now in the millions, it is not feasible to store all combinations while keeping track of their performance in a reasonable amount of time. Before we continue, we first wish to observe how the previous optimal boards came about.
Starting from the 1×1 board of just Green-Blue, we added the following placements for each new optimal board:
2×2 : GG, GR, BR
3×3 : GG, GB, GB, BR, BB
4×4 : GG, GR, GR, BR, BR, BB, BB

From this, one observation made is that GG and BR have always been added to make an $(n+1) \times (n+1)$ new optimal bingo board. Additionally, if we let n represent the total number of new placements added, we can see that the number of unique elements of the new placements is always at least half of n. Using this, we can assume that the most optimal 5×5 board may have these restrictions:

1. GG and BR are in the added placements

2. The number of unique elements of the new placements is always at least half of the total number of placements

3. RR will be in the added placements, following our assumptions from the conclusion of the 4×4 board.

```
list(filter(lambda x : "RR" in x and "GG" in x and "BR" in x
    and len(set(x)) >= 4, b))
len(b)
4966920
```

By getting rid of all boards which don't meet this criteria, we reduced the total number of combinations which need to be attempted by more than $\frac{1}{2}$.

5.3 Simulation Optimization

Approximately 5 million potential boards still remain and would each need to be checked. Given this large computation, we provided the following optimizations to the simulation.

1. *Checking each board*. Previously we created a dictionary for each board and saved it in memory for the duration of the simulation. Instead of creating all boards at once, we apply a generator function to the *getBoards* method which allows for us to iterate through all possible boards but only use and store one at a time, significantly reducing space complexity.

2. *Efficient Bingo Checking*. Instead of manually checking each tile for all "X"'s, we apply an all() function to each row, column, and diagonal to more efficiently check if the current board has bingo.

3. *While Loop*. We change the implementation of board iterations from a for loop to a while loop. This allows us to more succinctly check each individual board and calculate the average turns the board takes.

5.4 Constraints

We will iterate through all possible bingo boards and score them each, but given the large number of total possible combinations we can no longer run each board 100,000 times. More specifically, the simulation now has only approximately 100 games done on each board. We know from the Law of Large numbers that our n is not significant enough to create a concrete conclusion about the most optimal board, but we can use this information to estimate it as good as we can. Given the additional variance each score will now have, we will take the "average" of all top five boards. That is, we let each placement for the optimal 5×5 be the mode of the values in the corresponding positions for the top performing boards in the simulation.

5.5 Python Code for Simulation

```
import itertools
import random
from copy import deepcopy

random.seed(48)

# # Start with optimal tiling for a 5x5 bingo board
OPTIMAL_BOARD = [['GG', 'GB', 'GB', 'GR', ''],
                 ['GR', 'BR', 'GB', 'BB', ''],
                 ['GG', 'BR', 'BB', 'BR', ''],
                 ['GG', 'BB', 'GR', 'BR', ''],
                 ['', '', '', '', '']]


# Creating list for all the possible rolls. Occurences in list
   represent probabilities of being selected
possible_rolls = ["GG"] * 9 + ["GB"] * 12 + ["GR"] * 6 + ["BB"]
    * 4 + ["BR"] * 4 + ["RR"]
rolls = set(possible_rolls)
# Get all possible combinations for the last 9 cells
b = list(itertools.product(rolls, repeat = 9))

b = list(filter(lambda x : "RR" in x and "GG" in x and "BR" in
   x and len(set(x)) >= 4 , b ))
index = 1
# Fill in remaining 9 spots on the board
def get_boards(b):
    for vals in b:
        board = deepcopy(OPTIMAL_BOARD)
        board[0][4] = vals[0]
        board[1][4] = vals[1]
        board[2][4] = vals[2]
        board[3][4] = vals[3]
        board[4][0] = vals[4]
        board[4][1] = vals[5]
        board[4][2] = vals[6]
        board[4][3] = vals[7]
        board[4][4] = vals[8]

        yield board
boards = get_boards(b)
# Function to check if board has bingo, X's in the rows, cols,
   on on diagonals
def check_bingo(board):
```

```
    for row in range(len(board)):
        if all(board[row][col] == "X" for col in range(5)) or
           all(board[col][row] == "X" for col in range(5)):
            return True
    if all(board[i][i] == "X" for i in range(5)) or all(board[i
       ][4-i] == "X" for i in range(5)):
        return True
    return False

# Checks to see if the current tile picked can be placed on
   bingo board
def check_placement(board, pick):
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == pick:
                board[i][j] = "X"
                return

# Simulation to take turns for the game
def take_turns(board):
    tries = 0
    while not check_bingo(board):
        tries += 1
        pick = random.choice(possible_rolls)
        check_placement(board, pick)
        if tries > 20:
            return tries
    return tries
def check_better(bb, board, turns):
    worst = [-1000, 0] # Score : Index
    for index, tup in enumerate(bb):
        if tup[0] > worst[0]:
            worst[0] = tup[0]
            worst[1] = index
    if turns < worst[0]:
        bb[worst[1]] = (turns, board)

def check_worse(wb, board, turns):
    best_worst = [1000, 0] # Score : Index
    for index, tup in enumerate(wb):
        if tup[0] < best_worst[0]:
            best_worst[0] = tup[0]
            best_worst[1] = index
    if turns > best_worst[0]:
        wb[best_worst[1]] = (turns, board)
# Do 100 simulations for each board and calculate average turns
    taken
```

```
best_boards = [(i+100, None) for i in range(5)]
worst_boards = [(i-100, None) for i in range(5)]
while True:
    try:
        board = next(boards)
        avg_turns = [take_turns(deepcopy(board)) for _ in range
            (100)]
        avg_turns = sum(avg_turns) / len(avg_turns)
        check_better(best_boards, board, avg_turns)
        check_worse(worst_boards, board, avg_turns)
    except StopIteration:
        break
best_boards
```

5.6 Analyzing Results

The output of the function is the following:

```
[(8.92,
  [['GG', 'GB', 'GB', 'GR', 'GB'],
   ['GR', 'BR', 'GB', 'BB', 'RR'],
   ['GG', 'BR', 'BB', 'BR', 'BB'],
   ['GG', 'BB', 'GR', 'BR', 'BB'],
   ['GG', 'BR', 'BR', 'GB', 'RR']]),
 (8.98,
  [['GG', 'GB', 'GB', 'GR', 'GG'],
   ['GR', 'BR', 'GB', 'BB', 'BB'],
   ['GG', 'BR', 'BB', 'BR', 'BR'],
   ['GG', 'BB', 'GR', 'BR', 'BR'],
   ['RR', 'GR', 'BB', 'GG', 'GG']]),
 (8.99,
  [['GG', 'GB', 'GB', 'GR', 'GG'],
   ['GR', 'BR', 'GB', 'BB', 'GB'],
   ['GG', 'BR', 'BB', 'BR', 'BR'],
   ['GG', 'BB', 'GR', 'BR', 'RR'],
   ['GR', 'GB', 'GB', 'GB', 'GB']]),
 (9.02,
  [['GG', 'GB', 'GB', 'GR', 'GG'],
   ['GR', 'BR', 'GB', 'BB', 'GB'],
   ['GG', 'BR', 'BB', 'BR', 'GG'],
   ['GG', 'BB', 'GR', 'BR', 'RR'],
   ['BB', 'BR', 'GR', 'GG', 'RR']]),
 (9.02,
  [['GG', 'GB', 'GB', 'GR', 'GG'],
   ['GR', 'BR', 'GB', 'BB', 'GR'],
   ['GG', 'BR', 'BB', 'BR', 'BR'],
```

```
   ['GG', 'BB', 'GR', 'BR', 'RR'],
   ['GG', 'BR', 'BB', 'GR', 'GB']])]
```

Mode of each placement location (Index based on matrix representation) :
$B_{1,5}$: 4 GG | 1 GB
$B_{2,5}$: 2 GB | 1 RR | 1 BB | 1 GB
$B_{3,5}$: 2 BR | 1 BB | 1 GG | 1 GR
$B_{4,5}$: 3 RR | 1 BB | 1 BR
$B_{5,1}$: 2 GG | 1 RR | 1 GR | 1 BB
$B_{5,2}$: 3 BR | 1 GR | 1 GB
$B_{5,3}$: 2 BB | 1 BR | 1 GB | 1 GR
$B_{5,4}$: 2 GB | 2 GG | 1 GR
$B_{5,5}$: 2 RR | 2 GB | 1 GG

We can then choose the most common choice and if there is a tie, we simply pick the choice most likely to occur. Meaning if GG and RR both appeared twice, then we would choose GG since it has a higher likelihood. From our simulation and all of the calculations up until this point, we conclude that the optimal configuration for a 5×5 bingo board is:

$$\begin{pmatrix} \textit{Green-Green} & \textit{Green-Blue} & \textit{Green-Blue} & \textit{Green-Red} & \textit{Green-Green} \\ \textit{Green-Red} & \textit{Blue-Red} & \textit{Green-Blue} & \textit{Blue-Blue} & \textit{Green-BLue} \\ \textit{Green-Green} & \textit{Blue-Red} & \textit{Blue-Blue} & \textit{Blue-Red} & \textit{Blue-Red} \\ \textit{Green-Green} & \textit{Blue-Blue} & \textit{Green-Red} & \textit{Blue-Red} & \textit{Red-Red} \\ \textit{Green-Green} & \textit{Blue-Red} & \textit{Blue-Blue} & \textit{Green-Green} & \textit{Green-Blue} \end{pmatrix}$$

5.7 Final Sanity Check

The worst five boards are the following:

```
[(21.0,
  [['GG', 'GB', 'GB', 'GR', 'RR'],
   ['GR', 'BR', 'GB', 'BB', 'RR'],
   ['GG', 'BR', 'BB', 'BR', 'BB'],
   ['GG', 'BB', 'GR', 'BR', 'GG'],
   ['BB', 'GB', 'BR', 'GB', 'RR']]),
 (21.0,
  [['GG', 'GB', 'GB', 'GR', 'RR'],
   ['GR', 'BR', 'GB', 'BB', 'RR'],
   ['GG', 'BR', 'BB', 'BR', 'BR'],
   ['GG', 'BB', 'GR', 'BR', 'BB'],
   ['RR', 'GG', 'BR', 'GB', 'GB']]),
 (21.0,
  [['GG', 'GB', 'GB', 'GR', 'RR'],
   ['GR', 'BR', 'GB', 'BB', 'RR'],
```

```
   ['GG', 'BR', 'BB', 'BR', 'BR'],
   ['GG', 'BB', 'GR', 'BR', 'GB'],
   ['RR', 'RR', 'RR', 'GG', 'GR']]),
 (21.0,
  [['GG', 'GB', 'GB', 'GR', 'RR'],
   ['GR', 'BR', 'GB', 'BB', 'RR'],
   ['GG', 'BR', 'BB', 'BR', 'BR'],
   ['GG', 'BB', 'GR', 'BR', 'GG'],
   ['BB', 'BB', 'BR', 'GB', 'RR']]),
 (21.0,
  [['GG', 'GB', 'GB', 'GR', 'RR'],
   ['GR', 'BR', 'GB', 'BB', 'RR'],
   ['GG', 'BR', 'BB', 'BR', 'BR'],
   ['GG', 'BB', 'GR', 'BR', 'RR'],
   ['RR', 'GB', 'BR', 'BB', 'GG']])]
```

As expected, the worst boards are just numerous placements of RR and BR, which have the lowest probability of being selected.

Chapter 6

Conclusion

6.1 Video

We have attached our video below to provide an additional recap of the problem statement.

6.2 Summary

Overall, we were tested with exploring the probability of an optimal bingo board. We explored this by using probability as well as Python for simulation.
In case you have questions, comments, suggestions or have found a bug, please do not hesitate to contact us. You can find our contact details below.

Eddie Ramirez
ebr66@cornell.edu
Minhaj Fahad
msf257@cornell.edu
Oluwasola Ogundare
odo5@cornell.edu