Math 381: Assignment 5: Markov Chain

Ella Kim

November 5, 2021

1 Introduction

I have decided to create my own dice game that I can play by myself and still lose, win, gain points, and lose points.

Let my game start out with a score of 0, and a goal score of integer M. Each turn is a roll of a fair, 6-sided die. If my score is greater than 0 and the current turn/roll equally divides my score (i.e, roll integer d divides the score integer b if and only if there exists an integer m such that b = dm), then I subtract the current roll from my score (where the score will never be negative, only ever decreasing to 0). If it does not equally divide, then I add the current roll to my score. I continue this until I lose or win. I win if my score is greater than or equal to the goal score M. I lose if I am still under score M after adding the current roll, and the score is a perfect square greater than 4 (i.e, 9, 16, 25, 36, ...).

To analyze the probability of winning and losing based and the average turn to win on various M scores, I will use Markov chains (or in this case an absorbing Markov chain). Here I show the code I use to generate the transition matrix for a goal score of M (here I test M from 7-149 as computation time became too long after that, but I will only show the resulting M = 10 matrix):

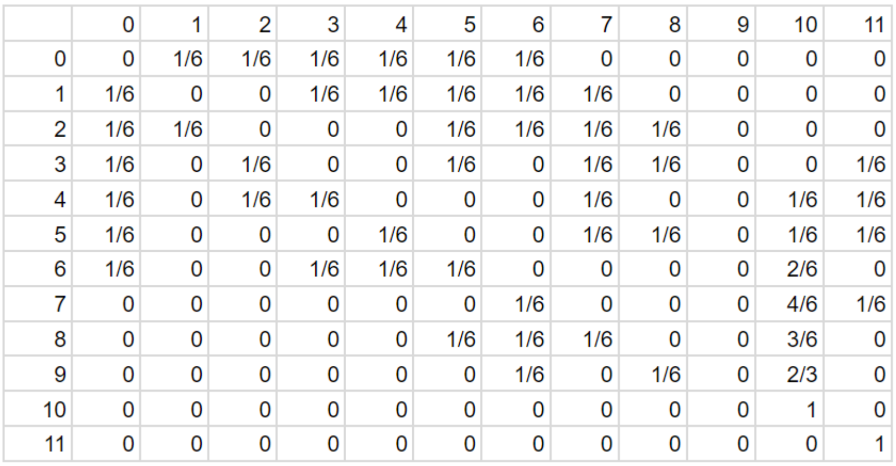
Python code:

```
import pandas as pd
import numpy as np
import math
import random
import statistics

probab = []
turns = []

# to check if the score is a perfect square
def is_square(test):
    sqroot = int(math.sqrt(test))
    return (sqroot == math.sqrt(test))

# go through goal score M 7 - 199
for a in range(7,200):
    m = a
    # age matrix to assure that values stay as fractions
    A=zero_matrix(QQ,m+2);
    # set absorbing states for winning and losing
    A[m,m] = 1
    A[m+1, m+1] = 1

    # for each score under the goal score M
    for i in range(0,m):
        # for each face of a 6-sided die
        for j in range(1,7):
            # if the starting score is 0, just add die value
            if i == 0:
                A[i,j] = 1/6
            else:
                # otherwise, if the roll divides evenly then
                # subtract from score
                score = 0
                if i%j == 0:
                    score = i-j
                # otherwise, add to score
                else:
                    score = i+j
                # check if win
                if score >= m:
                    A[i,m] += 1/6
                # check if lose
                elif is_square(score) and score >= 9:
                    A[i,m+1] += 1/6
                # add probability after checking win/loss
                else:
                    A[i,score] += 1/6
```

* * *

Let the resulting matrix A for goal score M be this transitional matrix (goal score 10 transitional matrix down below, where the row names refer to the score before a turn, the column names refer to the score after a turn, and where column name 10 (M) and 11 (M+1) correspond to winning and losing states/absorbing states):

Transition matrix for goal score 10:



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1/6	1/6	1/6	1/6	1/6	1/6	0	0	0	0	0
1	1/6	0	0	1/6	1/6	1/6	1/6	1/6	0	0	0	0
2	1/6	1/6	0	0	0	1/6	1/6	1/6	1/6	0	0	0
3	1/6	0	1/6	0	0	1/6	0	1/6	1/6	0	0	1/6
4	1/6	0	1/6	1/6	0	0	0	1/6	0	0	1/6	1/6
5	1/6	0	0	0	1/6	0	0	1/6	1/6	0	1/6	1/6
6	1/6	0	0	1/6	1/6	1/6	0	0	0	0	2/6	0
7	0	0	0	0	0	0	1/6	0	0	0	4/6	1/6
8	0	0	0	0	0	1/6	1/6	1/6	0	0	3/6	0
9	0	0	0	0	0	0	1/6	0	1/6	0	2/3	0
10	0	0	0	0	0	0	0	0	0	0	1	0
11	0	0	0	0	0	0	0	0	0	0	0	1

Throughout the next sections, I will use the corresponding transitional matrix A of dimension M+2 x M+2 for its corresponding goal score M.

2 Expected # of turns

As y transition matrix is of an absorbing Markov chain, it can be rewritten in canonical form:

$$P = \left(\begin{array}{c|c} Q & R \\ \hline O & J \end{array}\right)$$

where I is the identity matrix and O is an all-zero matrix. Q & R are non-negative matrices that arise from the transition probabilities between non-absorbing states.

Let matrix N be the converging series N = I + Q + Q^2 + Q^3 + ..., which also equals $(I-Q)^{-}1$. This means that:

1. The ij-th cell of matrix N is the expected number of times that I will have score j after starting at score i
2. The sum of the i-th row of matrix N gives the mean number of steps until I lose/win when the chain is started at score i
3. The ij-th entry of the matrix B = NR is the probability that, after starting in (non-absorbing) score i, the process will end up in me winning/losing state j

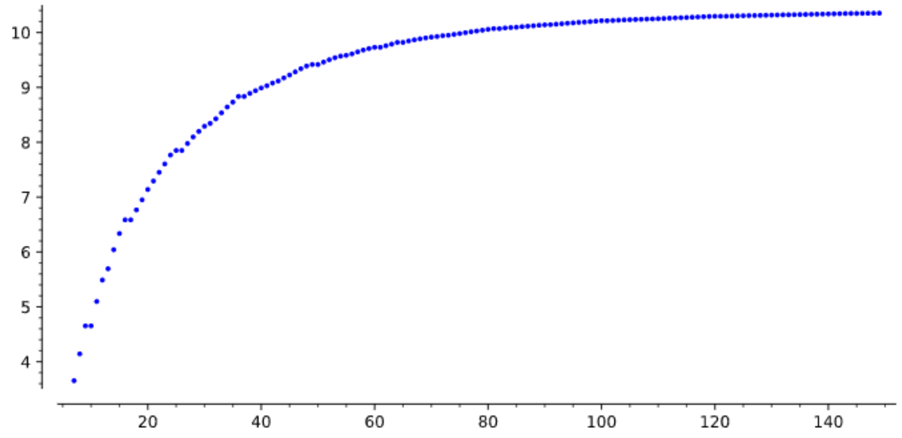
Using the second method, I calculated the number of rolls on average that it will take (i.e, the expected number of rolls) to make to win/lose when M is 10. Here, I include the Sage/Python code used to calculate this:

```
#added right after the import statements:
turns = []

    #added after the the end of the second for loop:
    I = identity_matrix(m)
    Q = A[:m,:m]
    N = (I - Q).inverse()
    turns.append(sum(N[0]))

# added to end of code
list_plot(list(zip(range(7,150), turns)))
```

* * *

Pulling the M = 10 out of the list gives the value $\frac{7515315}{1615288}$ or about 4.652616 rolls on average to end the game when the goal score is 10. Taking the rest of the list, I created a plot of the x axis being the goal score and y being the expected value of the number of rolls made before the game ends:

Plot: Expected rolls vs Goal Score M



As seen in the plot, the average increases, but decreases in magnitude of change as the goal score M increases. Another interesting behaviour is that after each perfect square, the next value is nearly the same (or the same: 9 & 10 = 7515315/1615288) as the previous, which makes the plot look like there is some steps.

3 Simulation

To test how well random simulations follow these expected values, I will now test a simulation of the game with a goal score of 10. I now show the Sage/Python code, where I run 10000 tests, find the average between those turns, and then compare 10 different runs of that:

```
simul_avg = []
simul_max = []
simul_min = []
# start with dummy values before the games start
# and to keep track of the max and min to comapare range
max_turn = 0
min_turn = 100
steps = 0
# the 10 different runs
for a in range(0, 10):
    # playing 10000 games
    for b in range(0, 10000):
        score = 0
        contin = True
        rolls = 0
        score = 0
        M = 10
        # while I have not lost/won
        while contin:
            rolls += 1
            # get random face on die
            curr_roll = random.randint(1,6)
            # make sure roll divides score evenly
            if (score > 0) and (score%rolls == 0):
                score = max(score - curr_roll , 0)
            # if not, add not subtract
            else:
                score = score + curr_roll
            # if I win
            if score >= M:
                win += 1
                contin = False
            # if I lose
            elif is_square(score) and score >= 9:
                lose += 1
                contin = False
        # find out whether max/min need to be updated
        max_turn = max(max_turn, rolls)
        min_turn = min(min_turn, rolls)
        steps += rolls
    # get average # of turns
    avg = float(steps/10000)
    simul_avg.append(avg)
    steps = 0
print("Max numer of turns until loss/win: ", max_turn)
print("Min numer of turns until loss/win: ", min_turn)
print("Average number of turns until loss/win: ",
        statistics.mean(simul_avg))
print("Range of average number of turns: ", max(simul_avg) -
min(simul_avg))
```

* * *

As for the output, each time it is ran it is different as this is a random simulation, but one of the output given was:

```
Max numer of turns until loss/win:  13
Min numer of turns until loss/win:  2
Average number of turns until loss/win:  4.76798
Range of average number of turns:  0.05900000000000016
```

* * *

Considering the exact expected value for a goal score of 10 ($\frac{7515315}{1615288}$ or about 4.652616), it is about 0.1 of a difference in value from the average number of turns and the range of values the 10 runs had from one another. Therefore, as it is very close to the expected value I had calculated, I conclude that my computation and resulting expected number of rolls in a game is accurate.

4 Probability

While it is good to know the average number of rolls to win/lose a game, I am also particularly interested in the probability of winning when the goal score is M. In order to compute this, I use the 3rd method in Section 2 where I defined matrix N, and where the ij-th entry of matrix B will give the probability of winning when starting at score i.

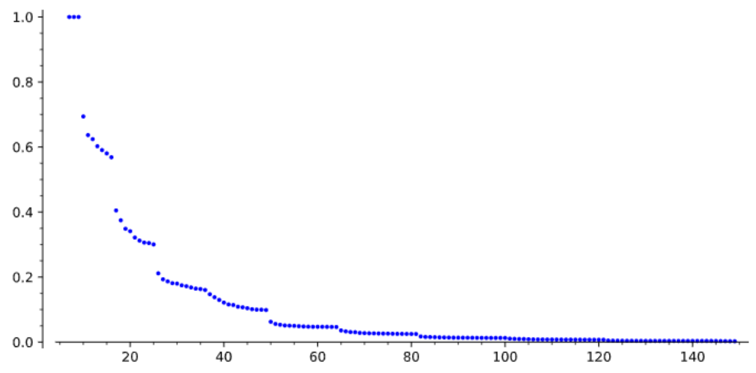
Here is the Python code added to the original for-loop (that created the transitional matrices):

```
#added before the for-loop and the is_square function
probab = []

    #added after the end of the second for-loop (from 0 to m)
    I = identity_matrix(m)
    Q = A[:m,:m]
    N = (I - Q).inverse()
    R = A[:m,m:]
    B = N*R
    # adding each probability to list
    # where [0,0] is not j = 0 but j = m
    # refer to R, where columns start at the m-th column of A
    probab.append(B[0,0])

#added at very end of code
list_plot(list(zip(range(7,150), probab)))
```

* * *

Adding this code to the original code gave a list of all the exact probabilities for winning a goal score of M. As this is a list from 7 to 149, I instead only list out the specific probability for a goal score of 10: $\frac{1120925}{1615288}$ or about 0.693947, and instead display a plot of the rest of the probabilities for each goal score m from 7 to 149:

Plot: Probability of Winning vs Goal Score M



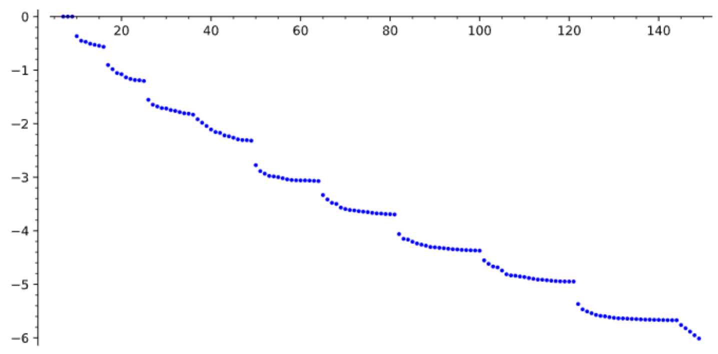
As seen in the plot, the general pattern is for the probability to decrease as the goal score M increases, and the magnitude of this decreasing trend also lessens as M increases, drawing close to 0, but again there is a pattern at the perfect squares, where, after another perfect square is considered a losing state, there is a drop in the next probability, which seems to make sense as its an increase in chance to lose the game.

To better see the values near the x-axis, I added another plot of the log function of the probabilities:

Python code for plot:

```
list_plot(list(zip(range(7,150), [log(float(x)) for x in probab])))
```

Plot: log(Probability) of Winning vs Goal Score M

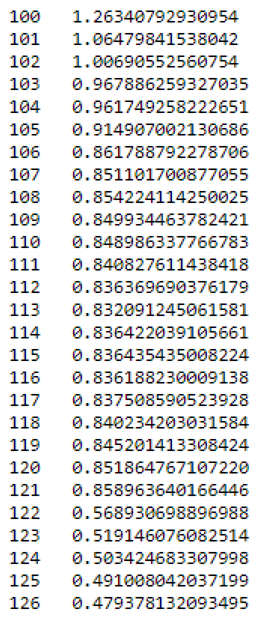


At first my purpose was to plainly see the larger M probabilities better, but just as we can see that the plot for probability of winning (without any transformations) is not a perfect/smooth equation/line, so is the ln transformation, as I would have expected the transformation to be linear if it was truly a 1/M function.

I will now try to find the function of M, or in other words, find f(M) where M is the goal score and f(M) is the probability.

My first guess is that f(M) is approximately equal to c/M for large M, for some constant c. Before I try guessing the function for c/M, I first look at the Mf(M) values:

Table of goal score M and Mf(M):



```
100   1.26340792930954
101   1.06479841538042
102   1.00690552560754
103   0.967886259327035
104   0.961749258222651
105   0.914907002130686
106   0.861788792278706
107   0.851101700877055
108   0.854224114250025
109   0.849934463782421
110   0.848986337766783
111   0.840827611438418
112   0.836369690376179
113   0.832091245061581
114   0.836422039105661
115   0.836435435008224
116   0.836188230009138
117   0.837508590523928
118   0.840234203031584
119   0.845201413308424
120   0.851864767107220
121   0.858963640166446
122   0.568930698896988
123   0.519146076082514
124   0.503424683307998
125   0.491008042037199
126   0.479378132093495
```

As seen in this table above, while M is already large, the values are still decreasing rapidly (rather than approaching a value: for reference M = 299 gives Mf(M) of 0.0204962, which is significantly smaller than Mf(M) for M = 126.

I instead focus on my ln/log plot and address how, although the points are not linear (they have intervals where they seem to be horizontally asymptotic), that I can find two linear functions $f_3(M)$ and $f_4(M)$ that, $f_3(M) <$ ln(f(M)) $< f_4(M)$, then there exists log(f(M)) between those two functions. This also only holds true for 100 <= M < 300, as I do no know the characteristics of smaller/larger f(M) in this plot, and I will be using only points on this ln(f(M)) plot to estimate upper and lower functions.

The method of finding these $f_3(M)$ and $f_4(M)$ functions may seem trivial, but I used a straight-edge and identified 1-2 points on both lower-side/upper-side of the plot (i.e, on ln(f(M))) that were the lowest/greatest, rounded up or down to prevent a function right on top of any points, and took the slope between each of those points. I then plugged one of those rounded points of ln(f(M)) and plugged it into y = mx + b were m was the slope I found to get the intercept. The resulting $f_3(M)$) and $f_4(M)$ function were -0.0242x -2.75 and -0.0238x - 1.9 respectively.

To show this visually, I now show the Python code and the corresponding output plot to show all 3 functions on the same plot:

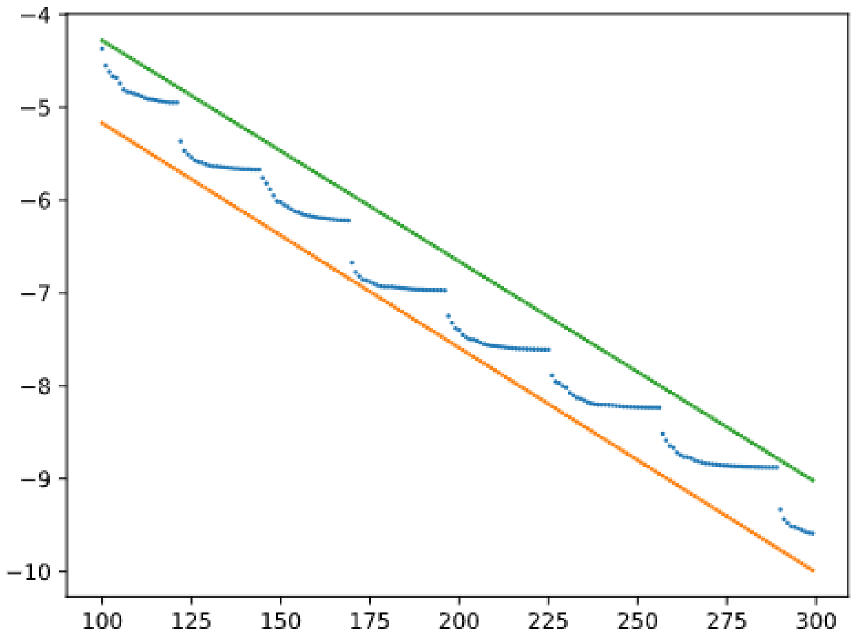
Python code for log plot of ln(f(M)), $f_3(M)$, and $f_4(M)$:

```
import matplotlib.pyplot as plt
# set x axis interval
x = list(range(100,300))
# create log version of f(M) in list in float type
logm = [math.log(float(x)) for x in probab]
f3 = []
f4 = []
# for each x-axis value
for i in range(100,300):
    # create list versions of f3 and f4
    f3.append(-0.0242*i-2.75)
    f4.append(-0.0238*i-1.9)
y = [logm,f3,f4]
# for each function
for i in y:
    # plot each function
    plt.scatter(x,i, s = 1)

plt.show()
```

* * *

And the visual output:

Plot of log(f(M)), $f_3(M)$, and $f_4(M)$ vs score M:



As this is not the original f(M) function, I now convert my plot back to f(M) by taking the $e^{ln(f(M))} = f(M)$, as well as $e^{ln(f_3(M))} = f_1(M)$ and $e^{ln(f_4(M))} = f_2(M)$, where the statement $f_3(M) <$ ln(f(M)) $< f_4(M)$ will hold true for $f_1(M)$ $<$ f(M)) $< f_2(M)$

Here again I show the Python code for this transition and plot of the f(M), $f_1(M)$, and $f_2(M)$:

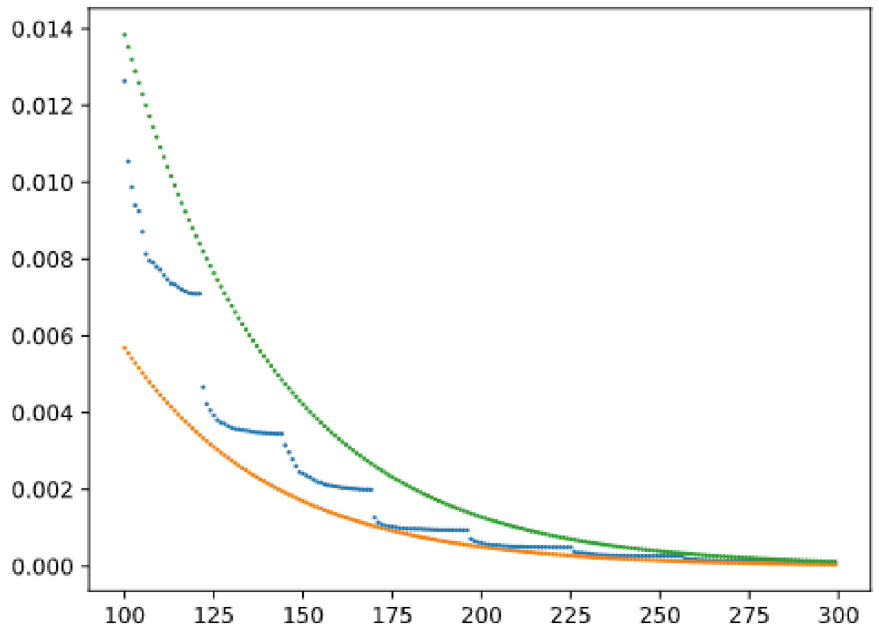
```
import matplotlib.pyplot as plt
# set x axis interval
x = list(range(100,300))
print(type(x))
f1 = []
f2 = []
# for each x-axis value
for i in range(100,300):
    # create list versions of f1 and f2
    f1.append(math.exp(-0.0242*i-2.75))
    f2.append(math.exp(-0.0238*i-1.9))
y = [probab,f1,f2]
```

```
# for each function
for i in y:
    # plot each function
    plt.scatter(x,i, s = 1)
# display all of them at once
plt.show()
```

* * *

And the visual output:

Plot of f(M), $f_1(M)$, and $f_2(M)$ vs score M:



Although the functions f1 and f2 could follow f(M) better (especially f1), it really would only be able to be a non-linear function in ln form to fit closer to the curved-like plot of f(M) on the lower boundary, but for now this is a approximation and a true statement that f(M) is between $e^{-0.0242M-2.75}$ and $e^{-0.0238M-1.9}$, and therefore fits better with a variation of e^{-M} than 1/M.