Project 6 – Markov Chains

CECS 381 - Sec 06

Dustin Martin - 015180085

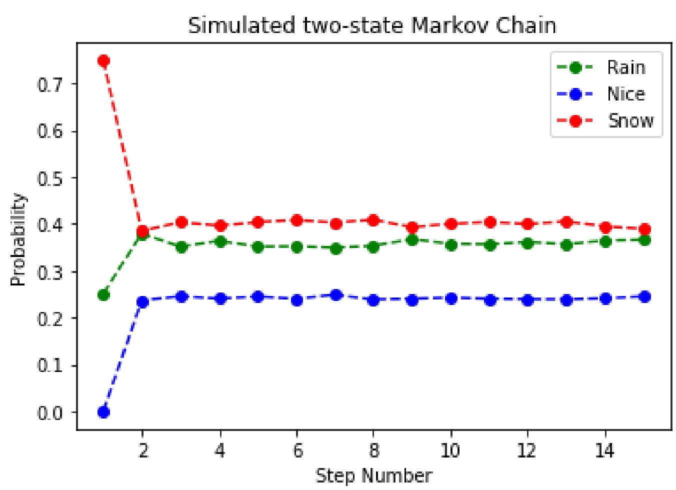
Problem 1:

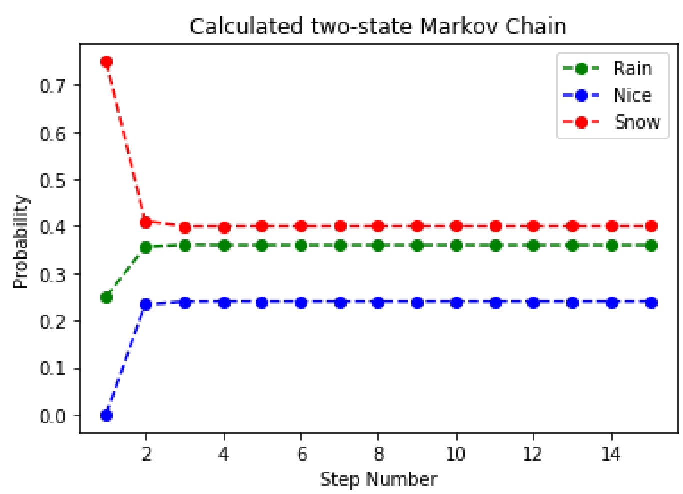
- **Introduction**
  Problem 1 demonstrates the accuracy of calculating the probabilities of Markov chains after n steps via simulation.

- **Methodology**
  Using the given State Transition matrix and initial probability vector, we can run simulations using the nSidedDie() function from previous labs. The probabilities are passed into the function and in return we get simulated transition switches. This is done for 15 steps and then repeated 10,000 times, with the results averaged out.

  We use this simulated graph to compare with the graph generated by repeatedly multiplying the matrix by itself for each step.

- **Results**





- ## Appendix/Source Code

```
def nSidedDie(p): #flips the unfair die a single time and returns the
result
    n = len(p)
    cs = np.cumsum(p)
    cp = np.append(0,cs)
    r = np.random.rand()
    for k in range(0,n):
        if r>cp[k] and r<=cp[k+1]:
            d=k+1
    return d

big_n = 10000
n = 15

R = [0] *n
N = [0] *n
S = [0] *n


initial = [1/4, 0, 3/4]

P=[ [1/3, 1/3,1/3], [1/3, 1/6, 1/2], [2/5, 1/5, 2/5]]

#1=R
#2=N
#3=S

for num in range(big_n):
    current = []
    current.append(nSidedDie(initial))
    for num in range(n-1):
        if current[-1] == 1:
            current.append(nSidedDie(P[0]))
        elif current[-1] == 2:
            current.append(nSidedDie(P[1]))
        elif current[-1] == 3:
            current.append(nSidedDie(P[2]))

    for num in range(len(current)):
        if current[num] == 1:
            R[num]+=1
        elif current[num] == 2:
            N[num]+=1
        elif current[num] == 3:
```

```
            S[num]+=1

for num in range(len(R)):
    R[num] = R[num]/big_n
    N[num] = N[num]/big_n
    S[num] = S[num]/big_n

bot = []
for num in range(n):
    bot.append(num+1)


plt.title("Simulated two-state Markov Chain")
plt.xlabel("Step Number")
plt.ylabel("Probability")
plt.plot(bot, R, 'g--', marker = "o", label='Rain')
plt.plot(bot, N, 'b--', marker = "o", label='Nice')
plt.plot(bot, S, 'r--', marker = "o", label='Snow')
plt.legend(loc="upper right")
plt.show()
```

Problem 2:

- **Introduction**
  Problem 2 uses Markov chains to rank website popularities based on traffic. Given a state transition matrix derived from a five-page web pasted in the next section, we can run simulations on it and gauge which sees the most traffic at n steps from initial.

- **Methodology**

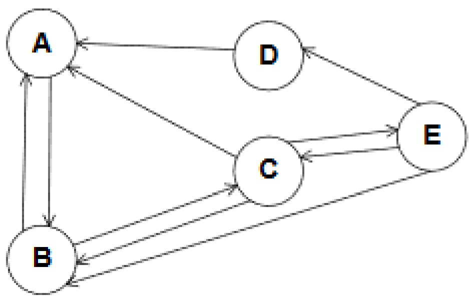


Figure 2.1: A five-page web

Given this 5 page web, we first construct a state transition matrix, P.

P = [[0, 1, 0, 0, 0],

[1/2, 0, 1/2, 0, 0],

[1/3, 1/3, 0, 0, 1/3],

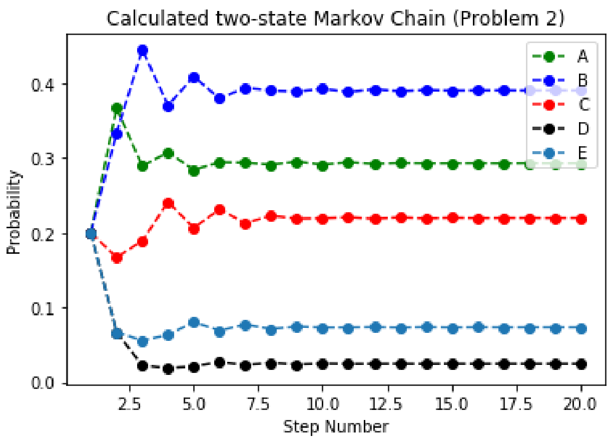
[1, 0, 0, 0, 0],

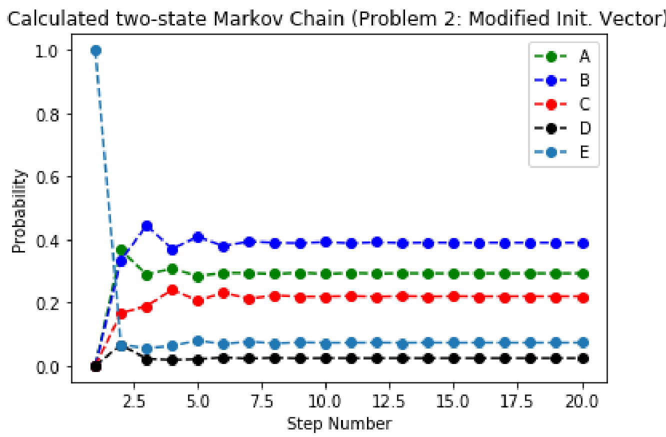
[0, 1/3, 1/3, 1/3, 0]]

Using P, we can basically just repeat the calculation made in problem 1 to estimate the rankings of each website.

We test two separate initial vectors in this experiment.
v1 = [1/5, 1/5, 1/5, 1/5, 1/5] (A, B, C, D, E)
and
v2 = [0, 0, 0, 0, 1] (E)

- **Results**





Initial probability vector: v_1		
Rank	Page	Probability
1	A	~0.2926
2	B	~0.3900
3	C	~0.2199
4	D	~0.02449
5	E	~0.07295

Initial probability vector: v_2		
Rank	Page	Probability
1	A	~0.2926
2	B	~0.3900
3	C	~0.2199
4	D	~0.02449
5	E	~0.07295

While v2 forces the matrix to start from E, the least popular page, there are enough steps that B is able to come back on top and remain the most popular page.

Appendix/Source Code:

```
#Problem 2

n=20

A = []
B = []
C = []
D = []
E = []

initial  = [1/5, 1/5, 1/5, 1/5, 1/5]

P = [[0, 1, 0, 0, 0],
     [1/2, 0, 1/2, 0, 0],
     [1/3, 1/3, 0, 0, 1/3],
     [1, 0, 0, 0, 0],
     [0, 1/3, 1/3, 1/3, 0]]


A.append(initial[0])
B.append(initial[1])
C.append(initial[2])
D.append(initial[3])
E.append(initial[4])

A.append(s.mean([P[0][0], P[1][0], P[2][0], P[3][0], P[4][0]]))
B.append(s.mean([P[0][1], P[1][1], P[2][1], P[3][1], P[4][1]]))
C.append(s.mean([P[0][2], P[1][2], P[2][2], P[3][2], P[4][2]]))
D.append(s.mean([P[0][3], P[1][3], P[2][3], P[3][3], P[4][3]]))
E.append(s.mean([P[0][4], P[1][4], P[2][4], P[3][4], P[4][4]]))

for num in range(2,n):

    y = np.linalg.matrix_power(P, num)
    A.append(s.mean([y[0][0], y[1][0], y[2][0], y[3][0], y[4][0]]))
    B.append(s.mean([y[0][1], y[1][1], y[2][1], y[3][1], y[4][1]]))
    C.append(s.mean([y[0][2], y[1][2], y[2][2], y[3][2], y[4][2]]))
    D.append(s.mean([y[0][3], y[1][3], y[2][3], y[3][3], y[4][3]]))
    E.append(s.mean([y[0][4], y[1][4], y[2][4], y[3][4], y[4][4]]))

plt.title("Calculated two-state Markov Chain (Problem 2)")
plt.xlabel("Step Number")
```

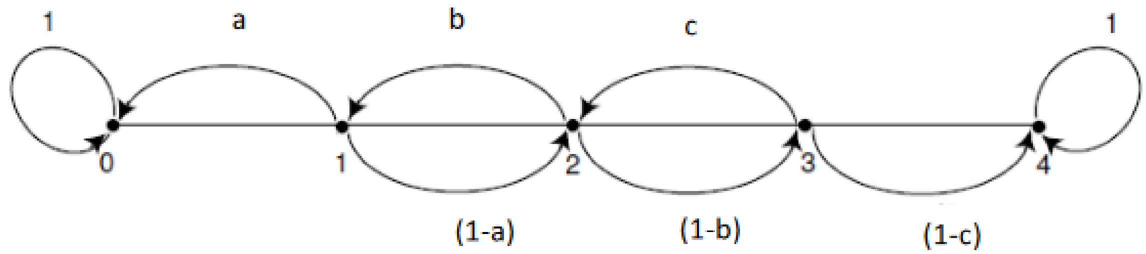
```
plt.ylabel("Probability")
plt.plot(bot, A, 'g--', marker = "o", label='A')
plt.plot(bot, B, 'b--', marker = "o", label='B')
plt.plot(bot, C, 'r--', marker = "o", label='C')
plt.plot(bot, D, 'k--', marker = "o", label='D')
plt.plot(bot, E, 'p--', marker = "o", label='E')

plt.legend(loc="upper right")
plt.show()

print(A[-1], B[-1], C[-1], D[-1], E[-1])

A = []
B = []
C = []
D = []
E = []

initial  = [0, 0, 0, 0, 1]

A.append(initial[0])
B.append(initial[1])
C.append(initial[2])
D.append(initial[3])
E.append(initial[4])

A.append(s.mean([P[0][0], P[1][0], P[2][0], P[3][0], P[4][0]]))
B.append(s.mean([P[0][1], P[1][1], P[2][1], P[3][1], P[4][1]]))
C.append(s.mean([P[0][2], P[1][2], P[2][2], P[3][2], P[4][2]]))
D.append(s.mean([P[0][3], P[1][3], P[2][3], P[3][3], P[4][3]]))
E.append(s.mean([P[0][4], P[1][4], P[2][4], P[3][4], P[4][4]]))

for num in range(2,15):

    y = np.linalg.matrix_power(P, num)
    A.append(s.mean([y[0][0], y[1][0], y[2][0], y[3][0], y[4][0]]))
    B.append(s.mean([y[0][1], y[1][1], y[2][1], y[3][1], y[4][1]]))
    C.append(s.mean([y[0][2], y[1][2], y[2][2], y[3][2], y[4][2]]))
    D.append(s.mean([y[0][3], y[1][3], y[2][3], y[3][3], y[4][3]]))
    E.append(s.mean([y[0][4], y[1][4], y[2][4], y[3][4], y[4][4]]))

plt.title("Calculated two-state Markov Chain (Problem 2: Modified
Init. Vector)")
plt.xlabel("Step Number")
plt.ylabel("Probability")
plt.plot(bot, A, 'g--', marker = "o", label='A')
```

```
plt.plot(bot, B, 'b--', marker = "o", label='B')
plt.plot(bot, C, 'r--', marker = "o", label='C')
plt.plot(bot, D, 'k--', marker = "o", label='D')
plt.plot(bot, E, 'p--', marker = "o", label='E')

plt.legend(loc="upper right")
plt.show()

print(A[-1], B[-1], C[-1], D[-1], E[-1])
```

Problem 3:

- **Introduction**
  Problem 3 simulates an absorbing Markov chain. States 0 and 4 can only return to themselves, therefore the state will remain 0 or 4 if either are reached.

- **Methodology**



Given this Markov chain and the provided modified probabilities
a=2/3; b=3/5; c=3/10

P = [[1, 0,0,0,0],

[2/3, 0, 1/3, 0, 0],

[0, 3/5, 0, 2/5, 0],

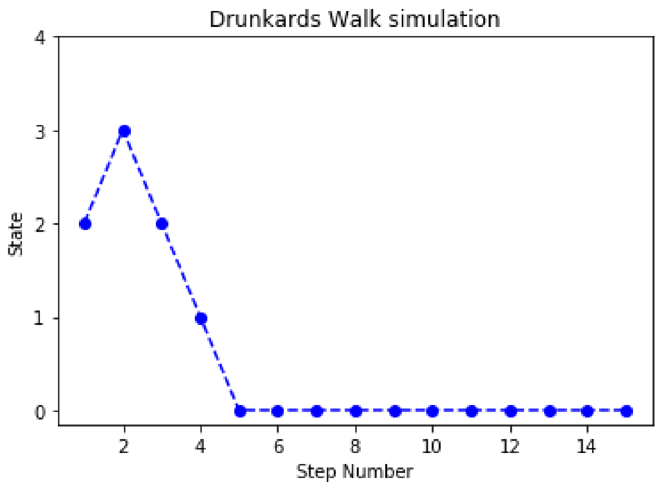
[0, 0, 3/10, 0, 7/10],

[0, 0, 0, 0, 1]]

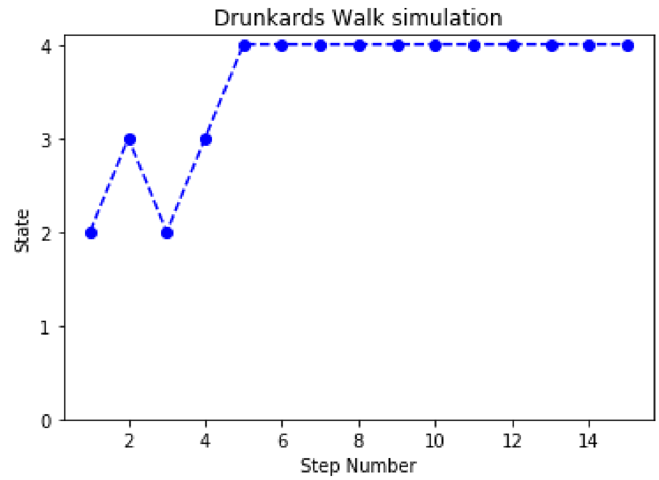
We will run two simulations by stepping through each step using the nSidedDie() function, starting from a random state on the chain.

The first will show absorption at state 0, the second at state 4.

- **Results**



Drunkards Walk simulation
State
Step Number
0
1
2
3
4
2
4
6
8
10
12
14



Drunkards Walk simulation
State
Step Number
0
1
2
3
4
2
4
6
8
10
12
14

Appendix

```
#Problem 3

n = 15
bot = []
for num in range(n):
    bot.append(num+1)
P = [[1, 0,0,0,0],
     [2/3, 0, 1/3, 0, 0],
     [0,3/5, 0, 2/5, 0],
     [0, 0, 3/10, 0, 7/10],
     [0,0,0,0,1]
     ]

start = random.randint(2,3)
initial = [0]*5

initial[start-1] = 1

current = []
current.append(nSidedDie(initial)-1)
for num in range(n-1):
    if current[-1] == 0:
        current.append(nSidedDie(P[0])-1)
    elif current[-1] == 1:
        current.append(nSidedDie(P[1])-1)
    elif current[-1] == 2:
        current.append(nSidedDie(P[2])-1)
    elif current[-1] == 3:
        current.append(nSidedDie(P[3])-1)
    elif current[-1] == 4:
        current.append(nSidedDie(P[4])-1)


plt.title("Drunkards Walk simulation")
plt.xlabel("Step Number")
plt.ylabel("State")

plt.plot(bot, current, 'b--', marker = "o")
plt.yticks([0,1,2,3,4])
plt.show()
```

Problem 4:

- **Introduction**
  Problem 4 repeats the previous simulation 10,000 times using the initial vector [0, 0, 1, 0, 0]. We will count the number of times the state is absorbed by 0 and 4 and calculate their probabilities.
  Given the modified state transition matrix probabilities, we'd expect that 0 will have a higher probability than 4.

- **Methodology**
  Repeat the previous simulation 10,000 times using the initial vector [0, 0, 1, 0, 0]. Count the number of times the state is absorbed by 0 and 4 and calculate their probabilities.

- **Results**

Absorption probabilities (via simulations)			
b_{20}	0.5868	b_{24}	0.4127

Appendix

```
#Problem 4

initial = [0, 0, 1, 0, 0]

zero = 0

four = 0

for num in range(big_n):
    current = []
    current.append(nSidedDie(initial)-1)
    for num in range(n-1):
        if current[-1] == 0:
            current.append(nSidedDie(P[0])-1)
        elif current[-1] == 1:
            current.append(nSidedDie(P[1])-1)
        elif current[-1] == 2:
            current.append(nSidedDie(P[2])-1)
        elif current[-1] == 3:
            current.append(nSidedDie(P[3])-1)
        elif current[-1] == 4:
            current.append(nSidedDie(P[4])-1)

    if current[-1] == 0:
        zero+=1
    elif current[-1] == 4:
        four+=1

av0 = zero/big_n
av4 = four/big_n

print(av0, av4)
```