12

Discrete-Time Markov Chains

Why Study Discrete-Time Markov Chains (DTMCs)?

- **They model random processes that can change over time**. Many real systems evolve in discrete steps, including queues, populations, networks, games, and genetic sequences.
- **They support "memoryless" behaviour**. This Markov property captures situations where the next state depends only on the current state, not the full past.
- **They can predict long-run behaviour**. DTMCs let you determine what will happen "in the long run", via stationary distributions, recurrence/absorption probabilities, mixing times, and limiting behaviour.
- **They connect numerous areas of mathematics**. DTMCs link probability, linear algebra, graph theory, and dynamic systems.

12.1 Introduction

The state X_n of a discrete-time Markov chain changes at each time step n, but is limited to a finite set S of possible states, called the state space. When the state happens to be S_i, there is probability p_{ij} that the next state is equal to S_j. In other words,

$$p_{ij} = \mathbf{P}(X_{n+1} = S_j \mid X_n = S_i). \quad S_i, S_j \in S.$$

A key assumption is that the same transition probability p_{ij} applies whenever state S_i is visited, no matter what happened in the past and how state S_i was

reached. Put another way, the probability of the next state X_{n+1} depends *only* on the value of the present state X_n.

The most convenient way to encode a Markov chain is as a transition probability matrix whose element in the S_ith row and S_jth column is p_{ij}:

$$\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix}$$

Markov chains also lend themselves to a visual representation as directed graphs whose nodes are the states and the arcs are labeled with the transition probabilities.

Markov chains can be used to model a wide range of behaviors, such as random walks, a Gambler's ruin (e.g. section 3.4), queuing, and many types of dynamic system involving probabilistic change (e.g. in signal processing, communications, and economics).

An important example is Google's original PageRank algorithm, which was based on calculating the importance of a page in terms of the importance of the pages linking to it. PageRank can be thought of as a model of a Web user's behavior. The 'surfer' begins on a random page and keeps clicking on links, never hitting the 'back' button, but, after getting bored, will start over on a new random page. The Markov chain representing this behavior (i.e. Web pages are states, and hyperlinks are probabilistic arcs) is executed until it achieves a steady state. The importance of a page is the long-term fraction of the time spent on that page in the steady state. Details can be found in Chapter 10 of Lay's *Linear Algebra and its Applications* [**LLM21**], and it's also explained in video lecture 33 of Harvard's Statistics 110 course on YouTube (`https://www.youtube.com/watch?v=Q-pCzTpwPBU&list=PL2SOU6wwxB0uwwH80KTQ6ht66KWxbzTIo&index=33`, starting at around the 17 minute mark).

Most textbooks on probability will have at least one chapter dedicated to Markov chains. For example, Chapter 11 of Grinstead and Snell's *Introduction to Probability*, 2nd edition [**GS97**], which is also online at `https://stats.libretexts.org/Bookshelves/Probability_Theory/Introductory_Probability_(Grinstead_and_Snell)`. Another is Chapter 7 of Bertsekas and Tsitsiklis' *Introduction to Probability*, 2nd edition [**BT08**]. The book has been used in several courses at MIT, which are on YouTube at 'MIT RES.6-012 Introduction to Probability, Spring 2018' (`https://www.youtube.com/playlist?list=PLUl4u3cNGP60hI9ATjSFgLZpbNJ7myAg6`) and 'MIT 6.041SC Probabilistic Systems Analysis and Applied Probability' (`https://www.youtube.com/playlist?list=PLUl4u3cNGP60A3XMwZ5sep719_nh95qOe`).

12.1.1 Example: Weather in Oz. According to Kemeny, Snell, and Thompson [**KST74**], the Land of Oz never has two nice days in a row. If they do have a nice day, then they're just as likely to have snow or rain the next day. If they have snow or rain, then there's an even chance of the same weather tomorrow. If there's a change then its only to nice weather half of the time.

If we label the states as R (Rain), N (Nice), and S (Snow), then the transition matrix becomes:

$$\mathbf{P} = \begin{array}{c} \\ R \\ N \\ S \end{array} \begin{array}{c} \begin{array}{ccc} R & N & S \end{array} \\ \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{pmatrix} \end{array}$$

This is stored in the file `OzWeather.txt` as:

```
labels: R N S

0.5  0.25 0.25
0.5  0    0.5
0.25 0.25 0.5
```

The directed graph generated by `display.py` for this matrix is shown in Fig. 12.1.

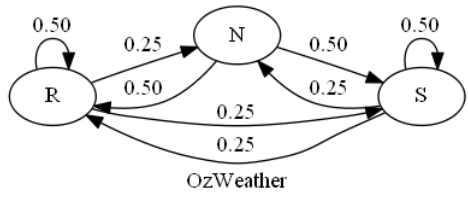


Figure 12.1. The Oz Weather Markov Chain as a Directed Graph

12.1.2 Example: Two Urns. Suppose we have two urns, labeled A and B, with N balls distributed between them. At any given moment, there are n balls in urn A and (obviously) $N - n$ balls in urn B. At each time step, we apply the procedure:

(1) Randomly choose one of the N balls (with equal probability).

(2) With probability q, move the chosen ball from its current urn into the other one. Otherwise, with probability $1 - q$, leave the ball where it is.

If there are n balls in urn A, we have probability n/N of choosing a ball from that urn, followed by a probability of q of moving it to urn B. Using similar reasoning, the three possible outcomes are:

- Move a ball from A to B: probability nq/N

- Move a ball from B to A: probability $(N-n)q/N$
- Leave the urns unchanged: probability $1-q$

We'll label the states of the system using $n \in 0, 1, ..., N$, corresponding to the number of balls in urn A. The graph for the Markov chain is shown in Fig. 12.2.

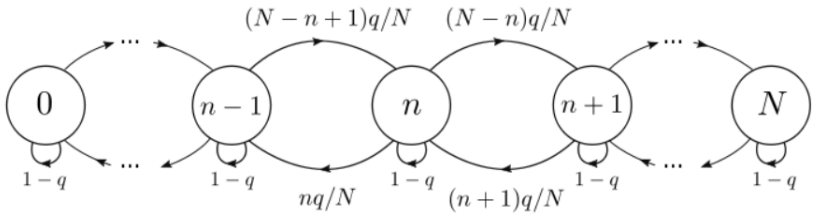


Figure 12.2. Two Urns with N balls and probability q of moving

Focusing on the n balls state in Fig. 12.2, the three possible changes are represented by the three arcs leaving that state.

urnsN.py generates a transition matrix for the urns problem given suitable values for N and q:

```
> python urnsN.py
N? 5
q=? 0.5
urnsN:
        0       1       2       3       4       5
0        0.50    0.50    0.00    0.00    0.00    0.00
1        0.10    0.50    0.40    0.00    0.00    0.00
2        0.00    0.20    0.50    0.30    0.00    0.00
3        0.00    0.00    0.30    0.50    0.20    0.00
4        0.00    0.00    0.00    0.40    0.50    0.10
5        0.00    0.00    0.00    0.00    0.50    0.50
```

The graph is shown in Fig. 12.3.

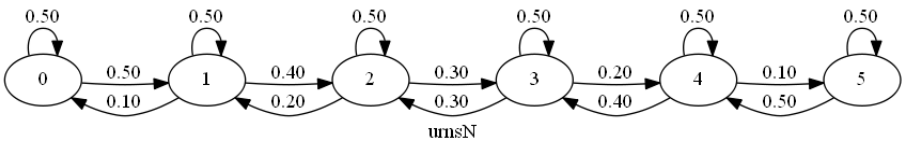


Figure 12.3. Two Urns with 5 balls and probability 1/2 of moving

The urns problem is a special case of the Ehrenfest model (`https://en.wikipedia.org/wiki/Ehrenfest_model`), which has been used to model gas diffusion.

12.2 n-Step Transitions

Most Markov chain problems require the calculation of state probability at some future time based on the current state. These n-step transition probabilities can be expressed as:

$$r_{ij}(n) = \mathbf{P}(X_n = S_j | X_0 = S_i).$$

$r_{ij}(n)$ is the probability that the state after n time steps will be S_j, given that the current state is S_i. It can be calculated recursively:

$$r_{ij}(n) = \sum_{k=1}^{m} r_{ik}(n-1)p_{kj}, \quad \text{for } n > 1, \text{ and all } S_i, S_j$$

starting with

$$r_{ij}(1) = p_{ij}$$

Fig. 12.4 is a visual interpretation of this equation.

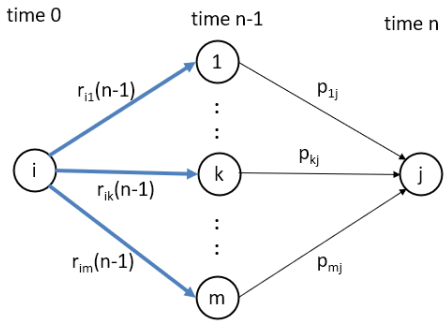


Figure 12.4. The $r_{ij}(n)$ Equation

The probability of being in state S_j at time n is the sum of the probabilities $r_{ik}(n-1)p_{kj}$ of the different ways of reaching S_j.

An equivalent way of representing $r_{ij}(n)$ is as a matrix obtained by multiplying the matrix for $r_{ik}(n-1)$ (which holds the $(n-1)$-step transition probabilities) to the one-step transition probability matrix P. This recursive definition can be rewritten as n powers of P, P^n, such that the ijth entry p_{ij}^n of P^n is the probability that the Markov chain, starting in state S_i, will be in state S_j after n steps.

We can use `powersTest.py` on the Land of Oz example to compute the successive powers of its matrix from 1 to 6. The results are shown below.

```
> python powersTest.py
fnm: OzWeather
Reading: OzWeather.txt
n: 6
```

```
mat^1:
        R       N       S
R       0.5000  0.2500  0.2500
N       0.5000  0.0000  0.5000
S       0.2500  0.2500  0.5000

mat^2:
        R       N       S
R       0.4375  0.1875  0.3750
N       0.3750  0.2500  0.3750
S       0.3750  0.1875  0.4375

mat^3:
        R       N       S
R       0.4062  0.2031  0.3906
N       0.4062  0.1875  0.4062
S       0.3906  0.2031  0.4062

mat^4:
        R       N       S
R       0.4023  0.1992  0.3984
N       0.3984  0.2031  0.3984
S       0.3984  0.1992  0.4023

mat^5:
        R       N       S
R       0.4004  0.2002  0.3994
N       0.4004  0.1992  0.4004
S       0.3994  0.2002  0.4004

mat^6:
        R       N       S
R       0.4001  0.2000  0.3999
N       0.3999  0.2002  0.3999
S       0.3999  0.2000  0.4001
```

After six time steps, the weather probabilities on each row are the same, to 2 dp accuracy, indicating that this chain has reached a *steady state*. The probabilities in any row can be interpreted as the long run fraction of time that the system will spend in each state, and the fact that each row is identical means that the probability of X_{n+1} no longer depends on X_n. For example, the probability of a nice day tomorrow is approximately $\frac{1}{5}$ no matter what the weather is today.

The following results are obtained when calling powers() on the urns matrix generated by urnsN.py, with $n = 40$:

```
> python urnsN.py
N? 5
q=? 0.5
Power n: 40
```

```
mat^1:
        0       1       2       3       4       5
0       0.5000  0.5000  0.0000  0.0000  0.0000  0.0000
1       0.1000  0.5000  0.4000  0.0000  0.0000  0.0000
2       0.0000  0.2000  0.5000  0.3000  0.0000  0.0000
3       0.0000  0.0000  0.3000  0.5000  0.2000  0.0000
4       0.0000  0.0000  0.0000  0.4000  0.5000  0.1000
5       0.0000  0.0000  0.0000  0.0000  0.5000  0.5000

  : many steps not shown

mat^39:
        0       1       2       3       4       5
0       0.0313  0.1563  0.3126  0.3124  0.1562  0.0312
1       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
2       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
3       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
4       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
5       0.0312  0.1562  0.3124  0.3126  0.1563  0.0313

mat^40:
        0       1       2       3       4       5
0       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
1       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
2       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
3       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
4       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
5       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
```

This chain also reaches a steady state, although it takes more time steps to do so.

12.3 Simulation

A good way to obtain a feel for the behavior of a Markov chain is to start it in a random state and execute the chain over multiple time steps. This should be repeated several times to see how different start states affect the long-term behavior.

Listing 12.1 (simMarkov.py) implements this approach after loading a Markov chain from a text file.

```
RUN_LEN = 20

fnm = input("fnm? ")
labels, P = readMarkov(fnm)
print("P:")
matLabels(P, labels)
# graph(fnm, labels, P)
```

```
p = P.data
n = len(labels)
count = [0] * n   # count of states visited
for _ in range(NUM_RUNS):
  print("> ", end=' ')
  currState = random.choice(range(n)) # random start
  for _ in range(RUN_LEN):
    r = random.random()
    for j in range(n):
      # decrement prob from r until it reaches 0 (or less)
      r -= p[currState][j]
      if r < 0:  # what state are we in?
        currState = j
        print(labels[currState], end=' ')
        break
    count[currState] += 1
  print()

print("\n\nLabel  Counts  Fraction")
totSteps = NUM_RUNS*RUN_LEN
for i in range(n):
  print(f"{labels[i]:<7}{count[i]:<8}{(count[i] / totSteps):.3f}")
```

Listing 12.1. Simulate the Behavior of a Markov Chain

For the Markov Chain in OzWeather.txt, its behavior over multiple runs is remarkably consistent, suggesting that the starting state has little effect on the chain's long-term behavior:

```
> python simMarkov.py
fnm? OzWeather
Reading: OzWeather.txt
P:
        R       N       S
R        0.50    0.25    0.25
N        0.50    0.00    0.50
S        0.25    0.25    0.50

>  S S S R R R R R R N R S R S S S S R R
>  S N S N S S R R N R N R N S S S R S N S
>  S S S R N S S S S S R R R R S R R R R N
>  S S S S S S S S R N S S S S S N R S R N S
>  S N S N S N R R R R R R R N S S R R R N
>  S N R R R R S S S S R N R R N R R R N S
     : many lines not shown
>  S N S N S S N S S S S N R R N S N R N S
>  N R N R R R N S S S S R N R S R R R R S
>  R R R R R N R N S N S S S N R R S S R R N
>  S R R R S R R R S N R R N S R R S S R R
```

```
Label  Counts  Fraction
R      398     0.398
N      189     0.189
S      413     0.413
```

The final output is a tally of the states at the end of the runs. After 50 runs, each one involving 20 time-steps, the fraction of times that the final state is R, N, or S are close to the steady-state values in P^n.

The behavior is quite different for drunk6.txt, an example that we'll discuss in Section 12.7. It appears to get 'stuck' in either state 0 or 5 depending on which one is nearer to the start state:

```
> python simMarkov.py
fnm? drunk6
Reading: drunk6.txt
P:
        0       1       2       3       4       5
0        1.00    0.00    0.00    0.00    0.00    0.00
1        0.50    0.00    0.50    0.00    0.00    0.00
2        0.00    0.50    0.00    0.50    0.00    0.00
3        0.00    0.00    0.50    0.00    0.50    0.00
4        0.00    0.00    0.00    0.50    0.00    0.50
5        0.00    0.00    0.00    0.00    0.00    1.00

>  5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
>  2 3 4 3 2 3 4 3 4 3 4 3 4 3 2 1 0 0 0 0
>  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
>  4 3 2 1 2 3 4 3 2 3 4 5 5 5 5 5 5 5 5 5
>  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
>  2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      : many lines not shown
>  4 3 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
>  5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
>  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
>  2 3 4 3 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

Label  Counts  Fraction
0      463     0.463
1      17      0.017
2      39      0.039
3      53      0.053
4      36      0.036
5      392     0.392
```

This example shows the utility of running a Markov chain with different start states.

12.4 Classification of States

Before we proceed, it's useful to define a few types of state found in Markov chains.

12.4.1 Accessible States. A state S_j is *accessible* from a state S_i if for some n, the n-step transition probability $r_{ij}(n)$ is positive, i.e., there is a chance of reaching S_j from S_i after some number of time steps.

12.4.2 Recurrent States. Let $A(i)$ be the set of states that are accessible from S_i. We say that S_i is *recurrent* if for every S_j that's accessible from S_i, then S_i is also accessible from S_j.

When we start in a recurrent state, S_i, there's always some probability of returning to S_i and, given enough time, that's certain to happen. Also, if a recurrent state is reached once, then it's certain to be revisited an infinite number of times.

12.4.3 Transient States. A state is *transient* if it's not recurrent. Thus, a state S_i is transient if there's a state $S_j \in A(i)$ such that S_i is *not* accessible from S_j. After each visit to state S_i, there's a positive probability that we'll visit state S_j, and once that occurs then state S_i can no longer be visited. As a consequence, a transient state can only be visited a finite number of times.

Fig. 12.5 shows a transition graph labeled with its recurrent and transient states. State 2 is transient since the system can eventually move to state 1 or 3, and state 2 is thereafter inaccessible. States 3 and 4 are recurrent since they're always accessible from each other.

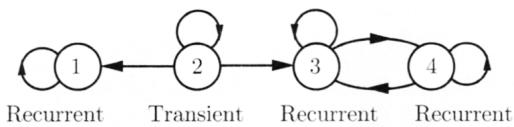


Figure 12.5. Recurrent and Transient States

12.4.4 Recurrent Classes. If S_i is a recurrent state, then the set of states $A(i)$ that are accessible from S_i form a *recurrent class*, meaning that the states in $A(i)$ are all accessible from each other, and no state outside $A(i)$ is accessible from them.

In Fig. 12.5, states 3 and 4 form a single recurrent class, and state 1 a second class.

12.4.5 Periodicity. A recurrent class is said to be *periodic* if its states can be grouped in $d > 1$ disjoint subsets $Y_1, \cdots, Y_d$ so that all the transitions in one subset point to the next subset. An example, where $d = 3$, is shown in Fig. 12.6.

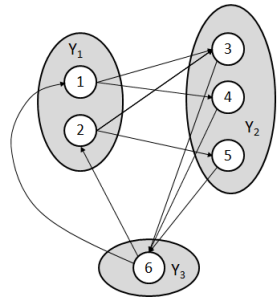


Figure 12.6. A Periodic Recurrent Class

More precisely:

$$\text{if } i \in Y_k \text{ and } p_{ij} > 0 \quad \text{then} \begin{cases} j \in Y_{k+1} & \text{if } k = 1, \cdots, d-1, \\ j \in Y_1 & \text{if } k = d. \end{cases}$$

We cycle through the subsets, and after a multiple of d steps return to the subset where we started.

A recurrent class that isn't periodic, is said to be *aperiodic*.

12.5 Steady-State Behavior

We can now be more precise about the meaning of steady-state behavior when the n-step transitions $(r_{ij}(n))$ converge (as occurs in the Oz Weather and urns examples in section 12.3).

Clearly, if a chain contains two or more recurrent classes then the convergence to steady-state values will depend on the initial state – the possibility of visiting S_j depends on whether it's in the same recurrent class as the current state. We'll therefore restrict our attention to chains with a single recurrent class, possibly with additional transient states. This isn't as restrictive as it may seem, since the behavior of chains with multiple recurrent classes can be analyzed by considering each class separately.

We also want to avoid periodicity since it makes the possibility of visiting a state S_j depend on whether it's located some multiple of the chain's periodicity d away from the current state.

If we assume that our Markov chain has a single recurrent class, and is also aperiodic, then its states S_j have steady-state probabilities π_j with the following properties:

(a) For each S_j, we can express the convergence of the n-step transitions as

$$\lim_{n\to\infty} r_{ij}(n) = \pi_j, \text{ for all } i.$$

(b) The π_j values are unique solutions to the following system of equations:

$$\begin{aligned} \pi_j &= \sum_{k=1}^{m} \pi_k p_{kj}, \quad j = 1, ..., m, \\ 1 &= \sum_{k=1}^{m} \pi_k. \end{aligned}$$

The equations defined by the first line are called *balance equations*, and the last equality is a *normalization equation*.

(c) We have:

$$\begin{aligned} \pi_j &= 0 \quad \text{for all transient states } j, \\ \pi_j &> 0 \quad \text{for all recurrent states } j. \end{aligned}$$

The balance and normalization equations can be solved to obtain the π_j steady-state probabilities for a chain, as illustrated in the following examples.

12.5.1 Weather in Oz. The OzWeather chain (see Fig. 12.1) has a single aperiodic recurrent class, so converges to a steady state. The balance equations are obtained by defining π_j using each row of the transition probability matrix. The result:

$$\pi_R = \frac{1}{2}\pi_N + \frac{1}{4}\pi_S + \frac{1}{2}\pi_R \tag{12.1}$$

$$\pi_N = \frac{1}{4}\pi_R + \frac{1}{4}\pi_S \tag{12.2}$$

$$\pi_S = \frac{1}{2}\pi_N + \frac{1}{4}\pi_R + \frac{1}{2}\pi_S \tag{12.3}$$

Simplifying π_R in 12.1:

$$\pi_R = \pi_N + \frac{1}{2}\pi_S \tag{12.4}$$

Substitute 12.2 into 12.4 to remove π_N:

$$\begin{aligned} \pi_R &= \frac{1}{4}\pi_R + \frac{3}{4}\pi_S \\ \pi_R &= \pi_S \end{aligned} \tag{12.5}$$

Substitute 12.5 into 12.2 to remove π_R:

$$\begin{aligned} \pi_N &= \frac{1}{4}\pi_S + \frac{1}{4}\pi_S \\ &= \frac{1}{2}\pi_S \end{aligned} \tag{12.6}$$

We now substitute equations 12.5 and 12.6 into the Normalization equation ($\pi_S + \pi_R + \pi_N = 1$) to obtain $\pi_S = 2/5$, and so $\pi_R = 2/5$ and $\pi_N = 1/5$. Note that these steady-state probabilities correspond to the results of calling `powersTest.py` on the OzWeather matrix in section 12.2.

12.6 The Two Urns

In the general case when there are N balls in the two urns, and a probability of q of moving a ball from urn A to B, then the balance equation for n balls in A is:

$$\begin{aligned}\pi_n &= \frac{N-(n-1)}{N}q\pi_{n-1} + (1-q)\pi_n + \frac{n+1}{N}q\pi_{n+1} \\ &= \frac{N-(n-1)}{N}\pi_{n-1} + \frac{n+1}{N}\pi_{n+1}\end{aligned}$$

The easiest way to see this is by considering the three arcs entering state S_n (the n node) in Fig. 12.2 from states S_{n-1}, S_n, and S_{n+1}. The probabilities on these arcs are assigned to π_{n-1}, π_n, and π_{n+1}.

Note that after simplification, π_n is independent of q. Intuitively, q governs how "quickly" we transfer balls from one urn to the other. Therefore, it should affect the speed at which the system reaches a steady state, but not the values in that state.

The balance equations becomes a little clearer if we let $p_n = n/N$ and $q_n = 1 - n/N$ (which is $(N-n)/N$). This allows us to write:

$$\pi_n = q_{n-1}\pi_{n-1} + p_{n+1}\pi_{n+1}$$

The π_i equations become:

$$\begin{aligned}\pi_0 &= p_1\pi_1 \\ \pi_1 &= q_0\pi_0 + p_2\pi_2 \\ \pi_2 &= q_1\pi_1 + p_3\pi_3 \\ &\vdots \\ \pi_N &= q_{N-1}\pi_{N-1}\end{aligned}$$

Induction shows that π_n can be rewritten nonrecursively as:

$$\pi_n = \frac{q_{n-1}q_{n-2}\cdots q_0}{p_n p_{n-1}\cdots p_1}\pi_0$$

So:

$$\begin{aligned}\pi_n &= \frac{(N-(n-1))(N-(n-2))\cdots N}{n(n-1)\cdots 1}\pi_0 \\ &= \frac{N!}{n!(N-n)!}\pi_0 \\ &= \binom{N}{n}\pi_0\end{aligned}$$

Combining this and the Normalization equation ($\pi_0 + \pi_1 + \cdots + \pi_N = 1$) implies that $\pi_0 = \frac{1}{2^N}$, and so:

$$\pi_n = \binom{N}{n}2^{-N}$$

We can check this against specific cases of the urns problem, such as the $N = 5$ run of `urnsN.py`. The steady state vector Π is:

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$$
$$\mathbf{\Pi} = \begin{bmatrix} 1/32 & 5/32 & 10/32 & 10/32 & 5/32 & 1/32 \end{bmatrix}$$

This tallies with the output from PowersTest.py at the end of section 12.2.

12.7 Absorption

Another common use of Markov chains is to determine short-term behavior, such as the first recurrent state to be entered, and when that occurs. This information is frequently required when a state is *absorbing*, which means that the state can never be left once entered. An equivalent definition of an absorbing state is one which has a transition back to itself with probability 1; i.e.,

$$p_{kk} = 1, \qquad p_{kj} = 0 \text{ for all } j \neq k.$$

Such a transition may occur naturally in a Markov chain (as in the examples below). Another situation is when we rewrite a recurrent class consisting of several states as a single absorbing state. This captures the fact that a recurrent class is never left once entered, and has the side-benefit of simplifying the analysis of the chain.

Typical questions for such chains include:

(a) What is the probability that the system will eventually reach an absorbing state?

(b) What is the probability that the system will end in a particular absorbing state?

(c) On average, how many time steps will it take for the system to reach an absorbed state?

(d) On average, how long will the system be in each transient state before being absorbed?

12.7.1 Example: The Drunkard's Walk. A man walks along a stretch of a city street with six 'stops' marked as '0' to '5' (see Fig. 12.7).

If the drunk is at stop 1, 2, 3, or 4, then he walks to the left or right with equal probability. He continues until he reaches stop 5, which is a bar, or stop 0, which is his home. If he reaches either home or the bar, he stays there. In other words, 0 and 5 are absorbing states and 1, 2, 3, and 4 are transient. The transition matrix

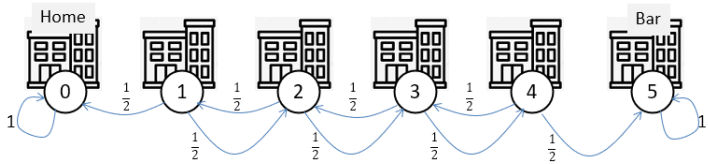


Figure 12.7. The Drunkard's Walk with 6 Stops

in drunk6.txt is:

$$\mathbf{P} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

12.7.2 Example: Searching a Maze. The maze consist of nine rooms with connections as indicated in Fig. 12.8.

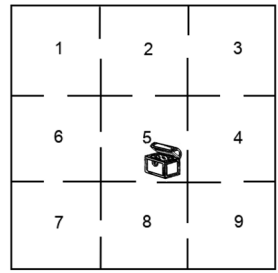


Figure 12.8. Searching a Maze for Treasure

The explorer is searching for room 5 which holds a prize, and chooses a doorway in the current room with equal probability. The transition matrix is in maze.txt:

```
> python display.py
fnm: maze
Reading: maze.txt
P:
        1       2       3       4       5       6       7       8       9
1        0.00    0.50    0.00    0.00    0.00    0.50    0.00    0.00    0.00
2        0.33    0.00    0.33    0.00    0.33    0.00    0.00    0.00    0.00
```

```
3       0.00    0.50    0.00    0.50    0.00    0.00    0.00    0.00    0.00
4       0.00    0.00    0.33    0.00    0.33    0.00    0.00    0.00    0.33
5       0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
6       0.33    0.00    0.00    0.00    0.33    0.00    0.33    0.00    0.00
7       0.00    0.00    0.00    0.00    0.00    0.50    0.00    0.50    0.00
8       0.00    0.00    0.00    0.00    0.33    0.00    0.33    0.00    0.33
9       0.00    0.00    0.00    0.50    0.00    0.00    0.00    0.50    0.00
```

Room 5 is the absorbing state, as confirmed by its self-transition probability of 1.

12.8 Finding Absorbing Probabilities

It's possible to calculate the probability a_i of eventually reaching an absorbing state s, starting from state i, by formulating recursive equations:

$$\begin{aligned} a_s &= 1, \\ a_i &= 0 \quad \text{for all absorbing states } i \neq s, \\ a_i &= \textstyle\sum_{j=1}^{m} p_{ij} a_j, \quad \text{for all transient states } i. \end{aligned}$$

In a similar way, the expected times to absorption, $\mu_1, ..., \mu_m$, are:

$$\begin{aligned} \mu_i &= 0 \quad \text{for all recurrent states } i \\ \mu_i &= 1 + \textstyle\sum_{j=1}^{m} p_{ij}\mu_j \quad \text{for all transient states } i. \end{aligned}$$

An alternative approach involves reorganizing the transition matrix into a *canonical* form which can be manipulated using matrix operations to find these probabilities and times.

The canonical form is obtained by rearranging the transition matrix's states so that the transient states come first. If there are r absorbing states and t transient states, the matrix becomes:

$$\mathbf{P} = \begin{array}{c} \\ \text{TR} \\ \text{ABS.} \end{array} \begin{array}{c} \begin{array}{cc} \text{TR.} & \text{ABS.} \end{array} \\ \left(\begin{array}{c|c} Q & R \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right) \end{array}$$

$\mathbf{I}$ is an r-by-r identity matrix, $\mathbf{0}$ is an r-by-t zero matrix, R is a nonzero t-by-r matrix, and Q is a t-by-t matrix. The first t states are transient and the last r states are absorbing.

The entry p_{ij}^n of the nth power of the rearranged matrix P is the probability of being in the state j after n steps, when the chain starts in state i. A standard matrix algebra argument shows that P^n will have the form:

$$\mathbf{P^n} = \begin{array}{c} \\ \text{TR.} \\ \text{ABS.} \end{array} \begin{array}{c} \begin{array}{cc} \text{TR.} & \text{ABS.} \end{array} \\ \left(\begin{array}{c|c} Q^n & * \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right) \end{array}$$

The asterisk * stands for the t-by-r matrix in the upper right-hand corner which can be written in terms of Q and R, but isn't needed here.

The entries in Q^n give the probabilities for being in each of the transient states after n steps for each possible transient starting state. Since the probability of being in a transient state after n steps approaches zero, every entry in Q^n will tend to zero as n approaches infinity.

For an absorbing Markov chain, the inverse of the matrix $I - Q$ is

$$\begin{aligned} N &= (I-Q)^{-1} \\ &= I + Q + Q^2 + \cdots . \end{aligned}$$

This states that the ij-entry n_{ij} of N is the expected number of times the chain is in state j, given that it starts in state i. This can be proved by letting $(I-Q)x = 0$; that is $x = Qx$. By iterating we see that $x = Q^n x$. However, $Q^n \to 0$, so $Q^n x \to 0$, and so $x = 0$ since $x = Qx$. A fundamental result of linear algebra is that a matrix M has an inverse if and only if $Mx = 0$ implies $x = 0$, which means that $(I-Q)^{-1}$ exists. Next, note that:

$$(I—Q)(I + Q + Q^2 + \cdots + Q^n) = I - Q^{n+1}.$$

Multiplying both sides by N gives

$$I + Q + Q^2 + \cdots + Q^n = N(I - Q^{n+1}).$$

Letting n go to infinity produces:

$$N = I + Q + Q^2 + \cdots .$$

N is called the *Fundamental Matrix.*

12.8.1 A Canonical Drunkard's Walk. For the Drunkard's walk example (Fig. 12.7), the transition matrix in canonical form is:

$$\mathbf{P} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ \hline 0 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \left(\begin{array}{cccc|cc} 0 & 1/2 & 0 & 0 & 1/2 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 0 & 1/2 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \end{array} \begin{array}{c} \\ \\ \\ \\ \\ \begin{array}{cc} 0 & 5 \end{array} \end{array}$$

So

$$\mathbf{Q} = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 & 0 \end{bmatrix}$$

and

$$\mathbf{I} - \mathbf{Q} = \begin{bmatrix} 1 & -1/2 & 0 & 0 \\ -1/2 & 1 & -1/2 & 0 \\ 0 & -1/2 & 1 & -1/2 \\ 0 & 0 & -1/2 & 1 \end{bmatrix}$$

Computing $(I - Q)^{-1}$ with Mat.py:

```
from Mat import *

Q = Mat([[0,   0.5, 0,   0],
         [0.5, 0,   0.5, 0],
         [0,   0.5, 0,   0.5],
         [0,   0,   0.5, 0] ])
N = (Mat.identity(4)-Q).inverse()   # 4-by-4 identity
print(N)
```

The result:

$$\mathbf{N} = (\mathbf{I} - \mathbf{Q})^{-1} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1.60 & 1.20 & 0.80 & 0.40 \\ 1.20 & 2.40 & 1.60 & 0.80 \\ 0.80 & 1.60 & 2.40 & 1.20 \\ 0.40 & 0.80 & 1.20 & 1.60 \end{pmatrix} \end{array}$$

From the second row of N, we see that if the system starts in state 2, then the expected number of times it returns to states 1, 2, 3, and 4 before being absorbed are 1.2, 2.4, 1.6, and 0.8.

12.9 Absorption Time and Probabilities

Given that a chain starts in state i, what is the expected number of steps t before the chain is absorbed? It can be expressed as:

$$t = Nc,$$

where c is a column vector whose entries are all 1's. The effect of the matrix multiplication is to sum the entries in the ith row of N, to get the total expected number of times state i visits the transient states.

Let b_{ij} be the probability that the system reaches absorbing state j if it starts in the transient state i. B, containing the b_{ij} entries, is generated using:

$$B = NR,$$

R is the nonzero t-by-r matrix from the canonical matrix. Each b_{ij} will be the sum of all the transitions through the transient states in Q ending in transient state k, followed by a transition to the absorbing state j with probability r_{kj}. This is:

$$\begin{aligned} b_{ij} &= \sum_n \sum_k q_{ik}^n r_{kj} \\ &= \sum_k \sum_n q_{ik}^n r_{kj} \\ &= \sum_k n_{ik} r_{kj} \\ &= (NR)_{ij}. \end{aligned}$$

12.9.1 Analyzing the Drunkard's Walk. In the six-stop Drunkard's walk,

$$\mathbf{N} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1.60 & 1.20 & 0.80 & 0.40 \\ 1.20 & 2.40 & 1.60 & 0.80 \\ 0.80 & 1.60 & 2.40 & 1.20 \\ 0.40 & 0.80 & 1.20 & 1.60 \end{pmatrix} \end{array}$$

The expected number of time steps, t, before absorption:

$$\mathbf{t} = \mathbf{Nc} = \begin{bmatrix} 1.60 & 1.20 & 0.80 & 0.40 \\ 1.20 & 2.40 & 1.60 & 0.80 \\ 0.80 & 1.60 & 2.40 & 1.20 \\ 0.40 & 0.80 & 1.20 & 1.60 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4.00 \\ 6.00 \\ 6.00 \\ 4.00 \end{bmatrix}.$$

Thus, starting in states 1, 2, 3, and 4, the expected times to absorption are 4, 6, 6, and 4, respectively. This reflects the fact that the closer the transient state is to an absorbing state, the less time it takes to be absorbed.

The probabilities of absorption:

$$\mathbf{B} = \mathbf{NR} = \begin{bmatrix} 1.60 & 1.20 & 0.80 & 0.40 \\ 1.20 & 2.40 & 1.60 & 0.80 \\ 0.80 & 1.60 & 2.40 & 1.20 \\ 0.40 & 0.80 & 1.20 & 1.60 \end{bmatrix} \begin{bmatrix} 0.5 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0.5 \end{bmatrix}$$

$$= \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 5 \end{array} \\ \begin{pmatrix} 0.80 & 0.20 \\ 0.60 & 0.40 \\ 0.40 & 0.60 \\ 0.20 & 0.80 \end{pmatrix} \end{array}$$

The shows that the probability of being absorbed into states 0 or 5 depends on how close the transient state is to one of them. The spread of probabilities in each column mirror each other since the drunk has equal chance of walking to the left or right.

12.10 Implementing Absorption

The absorb() function in MarkovLib.py carries out all of the calculations detailed in the previous section. It is passed the Q and R subarrays of the canonical matrix, and returns the fundamental matrix N, the absorption probabilities matrix B, and a vector t holding the expected times to absorption.

```
def absorb(Q, R):
  n = Q.nRows
  N = (Mat.identity(n)- Q).inverse()
  t = N*Mat.fill(n, 1, 1)  # column vector of 1's
```

```
  B = N*R
  return N, B, t
```

The values reported in the previous section for the Drunkard's Walk were obtained by passing drunk6.txt to absorbTests.py. It reads in the matrix, rearranges it to obtain the Q and R sub-matrices, calls absorb(), and prints the results:

```
fnm = input("fnm? ")
pLabels, P = readMarkov(fnm)
Q, qLabels, R, rLabels = getQR(P, pLabels)
N, B, t = absorb(Q, R)
```

getQR() makes the assumption that the matrix passed to it has absorbing states in the first and last row.

12.11 Implementing Steady State Using Matrices

Although section 12.5 solved balance equations with a normalization equation, it's also possible to obtain solutions using a matrix approach similar to that used in absorption problems.

The idea of a fundamental matrix (Z^*) for steady-state problems was first proposed by Kemeny and Snell in the first edition of their text *Finite Markov Chains* [**KS83**], where they defined:

$$Z^* = (I - P + e\pi)^{-1}$$

P is the transition probability matrix, e a column vector of ones, and π a row vector holding the steady state distribution.

Many properties of the Markov chain, such as the mean passage time and its variance, can be found using this fundamental matrix. The drawback is the cost of calculating π by solving n equations. This was addressed in a subsequent paper by Kemeny, which appeared as an appendix in later editions of the textbook. The matrix became:

$$Z_\beta = (I - P + e\beta)^{-1},$$

where β is any row vector such that $\beta \cdot e = 1$. Also, the steady-state vector π becomes much faster to calculate since $\pi = \beta \cdot Z_\beta$.

This approach is implemented in the getZPi() function in MarkovLib.py:

```
def getZPi(P):
  n = P.nRows
  e = Mat.fill(n, 1, 1)  # column vector, all 1's
  beta = Mat.fill(1, n, 0)  # row vector
  beta[0][0] = 1  # so that beta * e == 1

  X = Mat.identity(n) - P + (e*beta)
  Z = X.inverse()
  pi = beta*Z
  return Z, pi.row(0)  # return pi as a list
```

12.11.1 Using the Steady-State Matrix. If a recurrent Markov chain begins in state i, the expected number of steps to reach state j for the first time is called the *mean first passage time* from states i to j. It's denoted by m_{ij}, with the convention that $m_{ii} = 0$.

A quantity closely related to the mean first passage time is the *mean recurrence time*, r_i. Assume that we start in state i, and calculate the length of time before we return to i for the first time. We either stay in i or go to some other state j, and from there we'll eventually return to i because the chain is recurrent.

The mean first passage time from states i to j can be calculated recursively. If the first step is to state j, the expected number of steps is 1; if it goes to some other state k, the expected number of steps is m_{kj}, plus 1 for the step just taken. In addition, these steps must be multiplied by their probabilities of occurring. Thus,

$$m_{ij} = p_{ij} + \sum_{k \neq j} p_{ik}(m_{kj} + 1)$$

Since $\sum_k p_{ik} = 1$,

$$m_{ij} = 1 + \sum_{k \neq j} p_{ik} m_{kj}.$$

The mean recurrence times are obtained in a similar way but by considering the steps needed to return to state i:

$$\begin{aligned} r_i &= \textstyle\sum_k p_{ik}(m_{ki} + 1) \\ &= 1 + \textstyle\sum_k p_{ik} m_{ki}. \end{aligned}$$

If we have the fundamental matrix Z_β and steady-state vector π (which are both returned by getZPi() in MarkovLib.py), then a more convenient way to determine these values is by using:

$$\begin{aligned} m_{ij} &= \frac{z_{jj} - z_{ij}}{\pi_j} \\ r_i &= 1/\pi_i \end{aligned}$$

This approach is utilized in steady() in MarkovLib.py:

```
def steady(P):
  Z, pi = getZPi(P)
  n = P.nRows
  M = Mat.fillSq(n, 0)     # M = mean first passage times
  r = [0]*n     # r = mean recurrence times
  for i in range(n):
    for j in range(n):
      if pi[j] < EPS:   # ignore transient states
        M[i][j] = 0
      else:
        M[i][j] = (Z[j][j] - Z[i][j]) / pi[j]
    if pi[i] < EPS:  # ignore transient states
```

```
      r[i] = 0
    else:
      r[i] = 1 / pi[i]
  return Z, pi, r, M
```

12.11.2 Oz Weather Again. steadyTests.py reads in a Markov chain, calls steady(), and prints the various generated matrices:

```
> python steadyTests.py
fnm: OzWeather
Reading: OzWeather.txt
P:
        R       N       S
R        0.50    0.25    0.25
N        0.50    0.00    0.50
S        0.25    0.25    0.50

Fundamental matrix Z:
        R       N       S
R        0.93   -0.20   -0.40
N       -0.13    0.60   -0.13
S       -0.40   -0.20    0.93

Stationary Distribution pi:
        R       N       S
         0.4000  0.2000  0.4000

Mean Recurrence Times r:
        R       N       S
         2.50    5.00    2.50

Mean First Passage Times M:
        R       N       S
R        0.00    4.00    3.33
N        2.67    0.00    2.67
S        3.33    4.00    0.00
```

The M and r data reveal that the first nice day is a long time coming (4 days) and reoccurs infrequently (every 5 days).

Exercises

(1) A car insurance company groups its customers into three categories: 'poor', 'satisfactory', and 'preferred'. No one moves from 'poor' to 'preferred', or from 'preferred' to 'poor' in a single year. However, 40% of the customers in the 'poor' category become 'satisfactory', 30% of those in the 'satisfactory' category

moves to 'preferred', while 10% become 'poor'; 20% of those in the 'preferred' category are downgraded to 'satisfactory'.

(a) Write the transition matrix for the model.

(b) What is the steady-state fraction of car owners in each of these categories?

(2) In his book, *Wahrscheinlichkeitsrechnung und Statistik* [**Eng76**], the first author proposed an algorithm for finding the steady state for a Markov chain when the transition probabilities are rationals.

For each state i, let a_i be the least common multiple of the denominators of the non-zero entries in the ith row. The algorithm operates in terms of moving chips around on the states – indeed, for small examples, the algorithm can be implemented this way.

Start by putting a_i chips on state i for all i. Then, at each state, redistribute the a_i, chips, sending $a_i p_{ij}$ to state j. For each state i, add just enough chips to bring the number of chips at state i up to a multiple of a_i. Then redistribute the chips in the same manner.

This process will eventually reach a point where the number of chips at each state, after the redistribution, is the same as before redistribution. At this point, we have found a steady state. Here's an example:

$$\mathbf{P} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/2 & 1/4 & 1/4 \end{pmatrix} \end{array}$$

We start with $a_1 = (4, 2, 4)$. The chips after successive redistributions are:

```
(4 2 4)  (5 2 3)   (8 2 4)   (7 3 4)   (8 4 4)  (8 3 5) (8 4 8)
(10 4 6) (12 4 8)  (12 5 7)  (12 6 8)  (13 5 8) (16 6 8)
(15 6 9) (16 6 12) (17 7 10) (20 8 12) (20 8 12).
```

$a = (20, 8, 12)$ is a steady state.

Write a program to implement this algorithm.

(3) A highly simplified game of "Monopoly" is played on a board with four squares shown in Fig. 12.9. You start at "GO". You roll a die and move clockwise around the board a number of squares equal to the number that turns up on the die. You collect or pay an amount indicated on the square on which you land. You then roll the die again and move around the board in the same manner from your last position.

Calculate the amount you should expect to win in the long run playing this version of Monopoly.

-5 B	20 C
-30 A	15 GO

Figure 12.9. Simplified "Monopoly"

(4) A professor gives tests that are hard, medium, or easy. If she gives a hard test, her next test will be either medium or easy, with equal probability. However, if she gives a medium or easy test, there is a 0.5 probability that her next test will be of the same difficulty, and a 0.25 probability for each of the two other difficulty levels. Construct an appropriate Markov chain, and find the steady-state probabilities.

(5) SQSnakes and Ladders is played on a board of nine squares shown in Fig. 12.10.

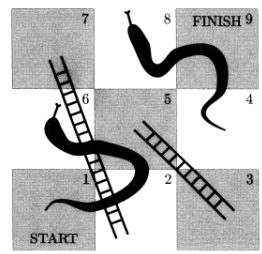


Figure 12.10. Small "Snakes and Ladders"

On each turn, a player tosses a fair coin and advances one or two places according to whether the coin lands heads or tails. If you land at the foot of a ladder you climb to the top, but if you land on the head of a snake you slide down its tail.

(a) How many turns on average does it take to complete the game?
(b) What is the probability that a player who has reached the middle square will complete the game without slipping back to square 1?

(6) A fair coin is tossed repeatedly and independently. Find the expected number of tosses required until the pattern HTHH appears.

(7) In tennis the winner of a game is the first player to win four points, unless the score is 4-3, in which case the game must continue until one player wins by having two more points than the other player.

Suppose that the game has reached the stage where one player is trying to get two points ahead to win, and that the server will independently win the point with probability 0.6.

What is the probability that the server will win the game if the score is currently tied at 3-3?; if she is ahead by one point?; behind by one point?

(8) Mary and John have a three-card deck marked with the numbers 1, 2, and 3, and a spinner with the numbers 1, 2, and 3 written on it. The game begins by dealing the cards out so that the dealer gets one card and the other person gets two. A move in the game consists of a spin of the spinner. The person having the card with the number that comes up on the spinner gives that card to the other person. The game ends when someone has all the cards.

(a) Set up the transition matrix for this absorbing Markov chain. Have the states correspond to the number of cards that Mary has.

(b) Calculate the fundamental matrix.

(c) On average, how many moves will the game last for?

(d) If Mary deals, what is the probability that John will win the game?

(9) A deck of cards has 3 red and 3 blue cards. At each stage, a card is selected at random. If it's red, it's taken from the deck and discarded. If it's blue then the card stays in the deck, and the game continues.

Find the average number of steps till there are no red cards left in the deck.