Chutes and Ladders and MCMC

Prompt

Original Prompt can be found here. A copy of the prompt along with the completed exercise can be found under /Applications.

Summary

This investigation concerns the boardgame Chutes and Ladders. Detailed instructions and some code have been provided in original prompt; For detailed explaination, please be sure to read the full prompt carefully.

The board has 100 spaces, labeled 1, 2, ..., 100. A player starts off the board. A player generally moves on the board according to the roll of a fair six-sided die. For example, if the player is currently on space 13 and they roll a 5, then they move to space 18. However, the board also has 9 ladders which help the player climb the board and 10 chutes (slides) which knock the player back down. The game ends when the player makes it to space 100. (We'll assume only one player.)

Problem 1

We are interested in T, the number of moves (rolls) needed until spot 100 is reached (the player doesn't need to land on 100 exactly). The position of the player after the nth move can be modeled as a Markov chain with transition matrix `Pgame` defined in the code from the prompt.

Problem 2

Suppose you were designing a new Chutes and Ladders board. How does the placement of the chutes and ladders on the board affect the expected value of T? In particular, is there a way to place the chutes/ladders to minimize the expected number of moves? In this problem, you'll write an MCMC algorithm to find the board which minimizes $E(T)$.

Application

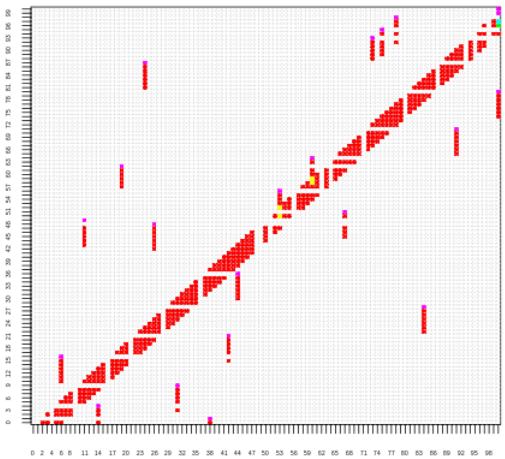
1.

First, create the Pgame matrix to use.

```
#use this to allow for running R within Python
%load_ext rpy2.ipython
```

```
%%R
N = 100 # number of spaces on board

s = N + 1  # number of states

k = 6  # number of sides on die

# P0 is the transition matrix if there were no chutes/ladders
P0 = matrix(rep(0, s * s), nrow = s)

for (i in 1:(N - 1)){
  for (j in min(i + 1, N):min(i + k, N)){
    if (j == N){
      P0[i, j] = (i - N + k + 1) / k
      # don't need to land on 100 exactly
    } else {
      P0[i, j] = 1 / k
    }
  }
}

P0[N, N] = 1  # absorbing state

P0[s, 1:k] = 1 / k  # initial state
```

```
%%R
# The make_board function takes as an input the starting spaces
# for chutes and ladders and outputs the transition matrix
# add the chutes/ladders by swapping appropriate columns
# with an annoying little detail for the two short chutes
# e.g. you can get from 50 to 53 by rolling a 3
# or by rolling a 6 and then sliding down the chute from 56 to 53

make_board <- function(ladder_start, chute_start, plot = FALSE){
  ladder_length = c(8, 10, 16, 20, 20, 21, 22, 37, 56)
  ladder_end = ladder_start + ladder_length
  chute_length = c(3, 4, 10, 20, 20, 20, 22, 38, 43, 63)
  chute_end = chute_start - chute_length
  P = P0
  for (j in 1:length(ladder_start)){
    i = which(P[, ladder_start[j]] > 0)
    P[i, ladder_end[j]] = P[i, ladder_start[j]]
    P[i, ladder_start[j]] = 0
    P[ladder_start[j], ] = rep(0, s)
    P[ladder_start[j], ladder_end[j]]=1
  }
  for (j in 1:length(chute_start)){
    i = which(P[, chute_start[j]] > 0)
    i1 = i[which(i <= chute_end[j])]
    P[i1, chute_end[j]] = P[i1, chute_start[j]] +
      P0[i1, chute_end[j]]
    P[i1, chute_start[j]] = 0
    i2 = i[which(i > chute_end[j])]
    P[i2, chute_end[j]] = P[i2, chute_start[j]]
    P[i2, chute_start[j]] = 0
    P[chute_start[j], ] = rep(0, s)
    P[chute_start[j], chute_end[j]] = 1
  }
  if (plot == TRUE){
    image(1:s, 1:s, t(P[c(s, 1:(s - 1)), c(s, 1:(s - 1))]),
          xlab = "", ylab = "",
          zlim = c(1 / k, 1), xaxt = "n", yaxt = "n",
          col = rainbow(k))
    axis(1, at = 1:s, labels = 0:(s - 1), cex.axis=0.4)
    axis(2, at = 1:s, labels = 0:(s - 1), cex.axis=0.4)
    grid(s, s)
  }
  return(P)
}


#run in R environment but export output variable to python
%%R -o Pgame
# generate the transition matrix for the actual game
Pgame = make_board(
  ladder_start = c(36, 4, 51, 71, 80, 21, 9, 1, 28),
  chute_start = c(56, 64, 16, 93, 95, 98, 48, 49, 62, 87),
  plot = TRUE)
```



```
%%R
# Check that all row sums are 1
which(!rowSums(Pgame) == 1)
```

```
integer(0)
```

a.

Solve for $E(T)$ without first finding the distribution of T.

```
%%R
mean_time_to_absorption <- function(transition_matrix, state_names = NULL) {

  absorbing_states = which(diag(transition_matrix) == 1)

  if (length(absorbing_states) == 0) stop("There are no absorbing states.")

  n_states = nrow(transition_matrix)

  transient_states = setdiff(1:n_states, absorbing_states)

  Q = transition_matrix[transient_states, transient_states]

  mtta = solve(diag(nrow(Q)) - Q, rep(1, nrow(Q)))

  if (is.null(state_names)) state_names = 1:n_states

  data.frame(start_state = state_names[transient_states],
             mean_time_to_absorption = mtta)
}
```

```
%%R
mu = mean_time_to_absorption(Pgame)
mu[100,]
```

```
         start_state mean_time_to_absorption
    100         101               36.19307
```

Above we can see the mean absorbtion time, $E(T)$, from off the board to spot 100.

b.

Solve for the exact distribution of T and plot it. (Technically, T can take infinitely many values, but feel free to cut off when the probabilities become sufficiently small.) Find $E(T)$ based on this distribution. Compare the expected value to the previous part.

```
%%R
install.packages('expm')
install.packages('kableExtra')
install.packages('tidyverse')
```

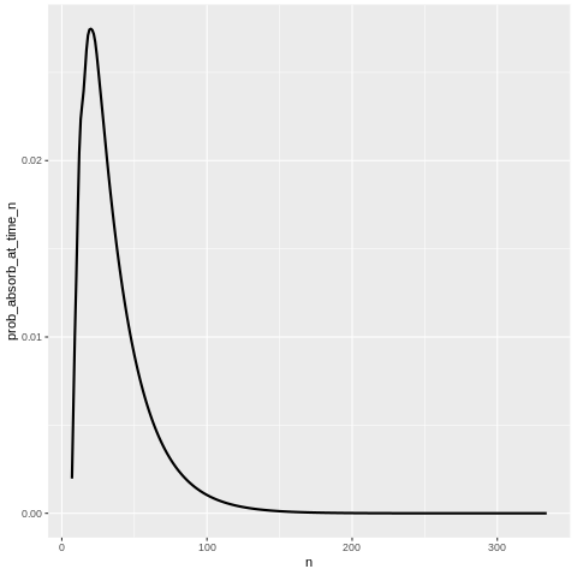
```
%%R
library(expm)
library(kableExtra)

pmf_of_time_to_absorption <- function(transition_matrix, state_names = NULL, start_state) {

  absorbing_states = which(diag(transition_matrix) == 1)

  if (length(absorbing_states) == 0) stop("There are no absorbing states.")

  n_states = nrow(transition_matrix)

  transient_states = setdiff(1:n_states, absorbing_states)

  if (is.null(state_names)) state_names = 1:n_states

  if (which(state_names == start_state) %in% absorbing_states) stop("Initial state is an absorbing state; absorption at time 0.")

  n = 1

  TTA_cdf = sum(transition_matrix[which(state_names == start_state), absorbing_states])

  while (max(TTA_cdf) < 0.999999) {

    n = n + 1

    TTA_cdf = c(TTA_cdf, sum((transition_matrix %^% n)[which(state_names == start_state), absorbing_states]))
  }

  TTA_pmf = TTA_cdf - c(0, TTA_cdf[-length(TTA_cdf)])

  data.frame(n = 1:length(TTA_pmf),
             prob_absorb_at_time_n = TTA_pmf)
}
```

```
%%R
T_pmf = pmf_of_time_to_absorption(Pgame, start_state = 101)

T_pmf |> head(100)
```

43	43	0.012123458
44	44	0.011612542
45	45	0.011123253
46	46	0.010654628
47	47	0.010205616
48	48	0.009775312

```
92   92           0.001466858
93   93           0.001404966
94   94           0.001345685
95   95           0.001288906
96   96           0.001234522
97   97           0.001182433
98   98           0.001132542
99   99           0.001084756
100 100           0.001038986
```

```
%%R
library(tidyverse)

ggplot(T_pmf |>
         filter(prob_absorb_at_time_n > 0),
       aes(x = n,
           y = prob_absorb_at_time_n)) +
  geom_line(linewidth = 1)
```




```
%%R
sum(T_pmf[, 1] * T_pmf[, 2])
```

```
[1] 36.19272
```

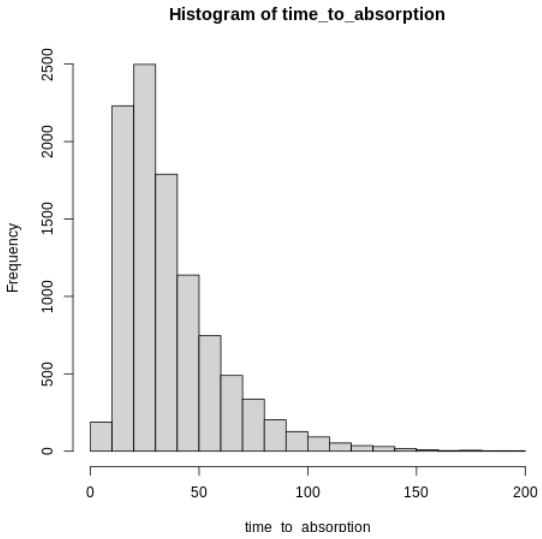
Both computation through absorbing state and cumulative PMF to find average result in values that agree. They both come out to be about 36.2 steps.

c.

Write code to run the chain and simulate the distribution of T. Plot the simulated distribution, use it to estimate the expected value, and compare to the previous part.

```
%%R
simulate_single_DTMC_path <- function(initial_distribution, transition_matrix, last_time){

  n_states = nrow(transition_matrix) # number of states

  states = 1:n_states # state space

  X = rep(NA, last_time + 1) # state at time n; +1 to include time 0

  X[1] = sample(states, 1, replace = TRUE, prob = initial_distribution) # initial state

  for (n in 2:(last_time + 1)){

    X[n] = sample(states, 1, replace = TRUE, prob = transition_matrix[X[n-1], ])

  }

  return(X)

}
```

```
%%R
pi0 <- rep(0, 101)
pi0[101] <- 1

absorbing_states = which(diag(Pgame) == 1)

n_rep = 10000
time_to_absorption = rep(NA, n_rep)

for (i in 1:n_rep) {
  x = simulate_single_DTMC_path(pi0, Pgame, last_time = 200)
  time_to_absorption[i] = min(which(x %in% absorbing_states))
}

hist(time_to_absorption)
```




```
%%R
summary(time_to_absorption)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      8      21      31     Inf      47     Inf
```

```
%%R
mean(time_to_absorption)
```

```
[1] Inf
```

```
%%R
sd(time_to_absorption)
```

```
[1] NaN
```

2.

First, think about what the optimal placement might look like. Then, write an MCMC algorithm to find the board that minimizes $E(T)$. Your MCMC algorithm should involve:

- Proposing a new state, that is, proposing a new board. A board is identified by the starting spaces of the chutes and the starting spaces of the ladders (that is, the inputs to the `make_board` function).
- Finding the expected value of T for the proposed board and then deciding whether or not to accept the proposed board. Note: if the proposed board is not valid (e.g., chutes/ladders land off the board), then it should be rejected.

Run the algorithm until you think it has converged and you have found the optimal board. Identify the the starting spaces for the chutes and ladders for this board.