Modeling Ludo Endgame with a Discrete-Time Markov Chain

npIomplete (Shiyi Chen)

June 26, 2026

Abstract

This note explains a compact DTMC model for the terminating path of a single Ludo token. The states are $\{0, 1, 2, 3, 4, 5, 6\}$ with destination state 6 absorbing. I justify the DTMC formulation, derive the entry distribution π_0, summarize formulas for hitting probability/time, and discuss analytical versus simulation outcomes.

1 Problem Setup

The code models only the final path, not the full multiplayer game. At each turn, one fair die is rolled. If a move overshoots state 6, the index is reflected:

$$\texttt{wrap\_index}(i, h) = \begin{cases} i + h, & i + h < 7, \\ 2(6) - (i + h), & i + h \geq 7. \end{cases}$$

State 6 is absorbing: once reached, it remains at 6.

2 Why DTMC Is a Good Modeling Choice

A DTMC is defined by a finite state space, stepwise transitions, and the Markov property. This endgame process satisfies all three:

1. **Discrete time:** the system evolves one die roll per step.
2. **Finite state space:** exactly seven states $\{0, 1, \ldots, 6\}$.
3. **Memoryless dynamics:** the next state depends only on the current state and current die roll, not on the path history.

The transition matrix is $P \in \mathbb{R}^{7\times 7}$ with

$$P_{ij} = \Pr(X_{n+1} = j \mid X_n = i),$$

derived from six equiprobable die outcomes and reflection. The absorbing row is $P_{66} = 1$, $P_{6j} = 0$ for $j \neq 6$.

This is exactly where DTMC helps: the same model gives both closed-form linear systems and straightforward Monte Carlo simulation.

3 How the Initial Distribution π_0 Is Computed

The code defines

$$\pi_0 = \left[\frac{6}{21}, \frac{5}{21}, \frac{4}{21}, \frac{3}{21}, \frac{2}{21}, \frac{1}{21}, 0\right].$$

Interpretation: before entering this path, the token is assumed to be on one of six outside nodes that are 1 to 6 steps away from entry state 0.

For each $k \in \{0, 1, 2, 3, 4, 5\}$, exactly $(6 - k)$ equally likely entry cases map to state k, so

$$\Pr(X_0 = k) = \frac{6 - k}{\sum_{r=1}^{6} r} = \frac{6 - k}{21}.$$

State 6 is excluded at entry, so $\Pr(X_0 = 6) = 0$, and normalization follows from $6 + 5 + \cdots + 1 = 21$.

In the implementation, if `start=None`, the current state is sampled from π_0 using weighted random choice.

4 Hitting Probability and Expected Hitting Time

Let t be a target state (here $t = 6$).

Hitting Probability α

Define

$$\alpha_i^{(t)} = \Pr_i(\tau_t < \infty),$$

where τ_t is the first time state t is visited and $\Pr_i$ means starting from state i. Boundary condition: $\alpha_t^{(t)} = 1$. For $i \neq t$:

$$\alpha_i^{(t)} = P_{it} + \sum_{j \neq t} P_{ij}\, \alpha_j^{(t)}.$$

The code solves this linear system by Gaussian elimination.

Expected Hitting Time β

Define

$$\beta_i^{(t)} = \mathbb{E}_i[\tau_t].$$

Boundary condition: $\beta_t^{(t)} = 0$. For $i \neq t$ (when hitting is almost sure):

$$\beta_i^{(t)} = 1 + \sum_{j \neq t} P_{ij}\, \beta_j^{(t)}.$$

If hitting is not almost sure, expected hitting time is treated as infinite in the code.

5 Results

Using 20,000 trials with target 6:

Metric	Analytical	Simulation
Hitting probability to state 6	1.000000	1.000000
Expected hitting time to state 6	6.000000	5.996200

6 Interpretation

Two points matter:

1. **Absorption is certain.** Hitting probability 1.0 means every run eventually reaches state 6.
2. **Mean completion is six turns.** The expected value is exactly 6 analytically and approximately 6 in simulation.

The simulation estimate (5.9962) is close to the analytical value (6.0), as expected from finite-sample Monte Carlo noise. Now that we see the endgame dynamics of Ludo isn't expected to be that harsh, as we always felt it was!

7 Appendix

Below is the implementation of the modelings above.

```
import random

class MC:

    def __init__(self, num_s):
        self.P = [[0.0 for _ in range(num_s)] for _ in range(num_s)]
        self.ns = num_s

    @staticmethod
    def _solve_linear_system(A, b, eps=1e-12):
        """Solve A x = b via Gaussian elimination with partial pivoting."""
        n = len(A)
        M = [row[:] + [b_i] for row, b_i in zip(A, b)]

        for col in range(n):
            pivot = max(range(col, n), key=lambda r: abs(M[r][col]))
            if abs(M[pivot][col]) < eps:
                raise ValueError("singular matrix")
            if pivot != col:
                M[col], M[pivot] = M[pivot], M[col]

            pivot_val = M[col][col]
            for j in range(col, n + 1):
                M[col][j] /= pivot_val

            for r in range(n):
                if r == col:
                    continue
                factor = M[r][col]
                if abs(factor) < eps:
                    continue
                for j in range(col, n + 1):
                    M[r][j] -= factor * M[col][j]

        return [M[i][n] for i in range(n)]

    def add_edge(self, s: int, t: int, p: float):
        assert 0 <= s < self.ns and 0 <= t < self.ns and 0 <= p <= 1
        self.P[s][t] = p

    def alpha(self, s: int, t: int): # hitting probability of eventually reaching t, starting
        assert 0 <= s < self.ns and 0 <= t < self.ns
        if s == t:
            return 1.0

        unknown_states = [i for i in range(self.ns) if i != t]
        index = {state: k for k, state in enumerate(unknown_states)}
        n = len(unknown_states)
        A = [[0.0 for _ in range(n)] for _ in range(n)]
        b = [0.0 for _ in range(n)]

        # a_i = P(i,t) + sum_{j != t} P(i,j) a_j  for i != t
        for i in unknown_states:
```

```
            row = index[i]
            A[row][row] = 1.0
            b[row] = self.P[i][t]
            for j in unknown_states:
                A[row][index[j]] -= self.P[i][j]

        sol = self._solve_linear_system(A, b)
        return sol[index[s]]

    def beta(self, s: int, t: int): # expected hitting time of eventually reaching t, startin
        assert 0 <= s < self.ns and 0 <= t < self.ns
        if s == t:
            return 0.0
        if self.alpha(s, t) < 1.0 - 1e-10:
            return float("inf")

        unknown_states = [i for i in range(self.ns) if i != t]
        index = {state: k for k, state in enumerate(unknown_states)}
        n = len(unknown_states)
        A = [[0.0 for _ in range(n)] for _ in range(n)]
        b = [1.0 for _ in range(n)]

        # h_i = 1 + sum_{j != t} P(i,j) h_j  for i != t (with h_t = 0)
        for i in unknown_states:
            row = index[i]
            A[row][row] = 1.0
            for j in unknown_states:
                A[row][index[j]] -= self.P[i][j]

        sol = self._solve_linear_system(A, b)
        return sol[index[s]]


def wrap_index(i, h, num_s):
    if i + h < num_s:
        return i+h
    else:
        return 2 * (num_s - 1) - (i + h)

def dice():
    return random.randint(1, 6)

class Ludo(MC):

    pi_0 = [6/21, 5/21, 4/21, 3/21, 2/21, 1/21, 0]

    def __init__(self, num_s = 7, start = None):
        super().__init__(num_s)
        for i in range(num_s - 1):
            for h in range(1, 7):
                self.P[i][wrap_index(i, h, num_s)] += 1/6
        for i in range(num_s - 1):
            self.add_edge(6, i, 0)
        self.add_edge(6, 6, 1)
        if start is None:
            # Sample starting state according to the initial distribution pi_0.
```

```
            if num_s != len(self.pi_0):
                raise ValueError("pi_0 is defined for num_s=7 only")
            self.curr = random.choices(range(num_s), weights=self.pi_0, k=1)[0]
        else:
            assert 0 <= start < num_s
            self.curr = start

    def _step(self, s):
        d = dice()
        self.curr = wrap_index(self.curr, d, self.ns)


if __name__ == "__main__":
    trials = 20000
    max_steps = 2000
    target = 6

    chain = Ludo()
    analytic_hitting_prob = sum(chain.pi_0[i] * chain.alpha(i, target) for i in range(chain.
    analytic_hitting_time = sum(chain.pi_0[i] * chain.beta(i, target) for i in range(chain.n

    hits = 0
    total_steps = 0

    for _ in range(trials):
        game = Ludo(start=None)
        steps = 0

        while game.curr != target and steps < max_steps:
            game._step(game.curr)
            steps += 1

        if game.curr == target:
            hits += 1
            total_steps += steps

    sim_hitting_prob = hits / trials
    sim_hitting_time = (total_steps / hits) if hits > 0 else float("inf")

    print(f"Analytical hitting probability to {target}: {analytic_hitting_prob:.6f}")
    print(f"Analytical expected hitting time to {target}: {analytic_hitting_time:.6f}")
    print(f"Simulation hitting probability to {target}: {sim_hitting_prob:.6f}")
    print(f"Simulation expected hitting time to {target}: {sim_hitting_time:.6f}")
```