From Monkeys to Markov Chains: Analyzing Random Character Sequences

Gabe Cardella

23 April, 2024

Introduction

Project Motivation and Goal

The Infinite Monkey Theorem, a popular concept involving probability, states that a monkey hitting random typewriter keys for an infinitely long period of time will almost definitely type any particular piece of text, even works of Shakespeare. Of course, as Banerji, Mansour, and Severini [1] note, this "monkey" can refer to any device that produces a random sequence of characters. Although humorous and extreme, it does highlight that, when given enough independent trials, even events that have exceedingly low probabilities will take place. A plausible assumption regarding the infinitely long sequence of characters typed by our hypothetical monkey is that all strings of equal length, l, have an equal probability of occurring within a given number of t keystrokes. This assumption, however, is false; see Example 1.

The fact that this assertion is a fallacy is the motivation for much of the analysis and algorithm development in this project. The goal of the project is to first understand why the statement is false. Following that is the development of efficient algorithms that calculate the average *hitting time* of an input string (how many random characters, or keystrokes, it takes for a string to first occur), the Probability Mass Function (PMF) of this hitting time, and the winning probabilities for the players in a related multiplayer game. To accomplish these tasks, one of the most important mathematical concepts that we use is the Markov chain.

Markov Chain Overview

A Markov chain, named after Russian mathematician Andrey Markov, is a mathematical system that transitions between states. Markov chains are referred to as Discrete-Time Markov chains (DTMCs) when these transitions occur at discrete time steps. A DTMC is a set of states, Q, and a transition probability function $f : Q \times Q \to \mathbb{R}$ which gives the *transition probability* P_{ab} that the next state is b given that the current state is a. The transitions are *stochastic*, meaning that the system's next state is selected randomly according to f. One of the key features of a Markov chain is that it exhibits "memoryless behavior", where the system's future behavior is not influenced by its past behavior. Instead, the probability of transitioning to any particular state depends solely on the system's current state. To illustrate many of the key concepts relating to Markov chains, we use weather prediction as a simple example.

Let $Q = \{\mathrm{S}, \mathrm{R}\}$, where S represents a sunny day and R represents a rainy day. Transitions between these states occur over discrete time steps (days), so this Markov chain is a DTMC. Let X_t be the state of the weather on day t. Note that $X_0, X_1, X_2, \ldots$ is a sequence of random variables such that $Pr(X_{t+1} = b \mid X_t = a) = P_{ab}$ for all $(a, b) \in Q^2$. Transition probabilities are typically organized in a *transition matrix*, a square matrix where each entry P_{ab} equals the probability of transitioning from state a to state b. Given our weather example, we can define the following transition probabilities:

- $P_{SS} = Pr(X_{t+1} = \text{S} \mid X_t = \text{S}) = 0.6$
- $P_{SR} = Pr(X_{t+1} = \text{R} \mid X_t = \text{S}) = 0.4$
- $P_{RS} = Pr(X_{t+1} = \text{S} \mid X_t = \text{R}) = 0.25$
- $P_{RR} = Pr(X_{t+1} = \text{R} \mid X_t = \text{R}) = 0.75$

These can be illustrated in a Markov chain, seen below in Figure 1.

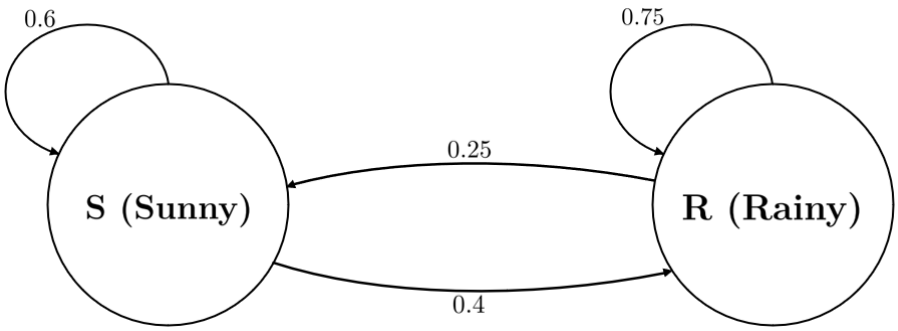

Figure 1: Markov chain from weather example.

We can organize these into a transition matrix:

$$P = \begin{bmatrix} P_{SS} & P_{SR} \\ P_{RS} & P_{RR} \end{bmatrix} = \begin{bmatrix} 0.6 & 0.4 \\ 0.25 & 0.75 \end{bmatrix}$$

The i^{th} row of P gives the probability distribution of the successor states given that the current state is i. So, each row of P sums to 1. A transition matrix can be used to calculate the probabilities of being in each state at time t. These probabilities are referred to as the *probability distribution*, λ_t (Norris, [2]). For instance, if we want to find the probability distribution in our weather example when $t = 3$, we can perform the following calculation. Note that here, λ_0 represents the *initial distribution*, the probabilities of being in each state at time $t = 0$. The value of λ_t depends on the initial distribution. Given that there is a 70% chance of it being sunny and a 30% chance of it being rainy on day 0, we have that $\lambda_0 = \begin{bmatrix} 0.7 & 0.3 \end{bmatrix}$. So,

$$\lambda_t = \lambda_0 \cdot P^t = \begin{bmatrix} 0.7 & 0.3 \end{bmatrix} \cdot \left(\begin{bmatrix} 0.6 & 0.4 \\ 0.25 & 0.75 \end{bmatrix} \right)^3 = \begin{bmatrix} 0.7 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.411 & 0.589 \\ 0.368125 & 0.631875 \end{bmatrix} = \begin{bmatrix} 0.3981375 & 0.6018625 \end{bmatrix}$$

This result tells us that on day 3, the probability of it being sunny is approximately 0.398, and the probability of it being rainy is approximately .602.

Note that there is a special type of state called an *absorbing state*. A state i is absorbing if and only if $P_{ii} = 1$. An *absorbing Markov chain* has at least one absorbing state. This also implies that $P_{ij} = 0$, for all states $j \in Q$, such that $j \neq i$. So, once a Markov chain reaches an absorbing state, it can never leave it. Normally, a transition matrix is indexed so that the absorbing states are placed last.

In this project, the DTMCs that are used to model a *target string* appearing in a random sequence of letters contain an absorbing state. This state is reached if each of the target string's characters appear consecutively. So, the last entry in the probability distribution, λ_t, equals the probability that the target string appears somewhere within the first t random keystrokes. With an understanding of Markov chains established, we now discuss how target strings, and their appearance in a random sequence of characters, can be represented as Markov chains.

Representing Target Strings as Markov Chains

We start the representation of a target string as a Markov chain by considering the alphabet Σ, the set of all possible characters that can occur in the random string $r = r_1\ r_2...$ typed by our hypothetical monkey. We denote a target string using s, where $s = c_1...c_l$ Next, let Q, the Markov chain's state space, be the set of prefixes of s, with $Q = \{q_0, q_1, q_2, ..., q_l\}$, where $q_i = c_1...c_i$. Note that q_0 is the empty string, denoted by ε.

For $t \geq 0$, if s has not appeared as a substring of $r_1\ r_2...r_t$, then state of the Markov chain at time t equals the largest suffix of $r_1\ r_2...r_t$ that is found in $Q - \{q_l\}$. If, however, s has appeared as a substring of $r_1\ r_2...r_t$, the Markov chain at time t is in state q_l. Note that the Markov chain enters state q_l, and remains in state q_l, directly after c_l is typed, given that the characters $c_1...c_{l-1}$ appeared consecutively and directly before c_l. Thus, state q_l of the Markov chain that represents target string s is absorbing.

An example of the representation of a target string as a Markov chain is seen below in Figure 2. This figure gives both the transition matrix P and the associated Markov chain of the example where $s =$"ABA", $\Sigma = \{\text{A}, \text{B}, \text{C}\}$, and $Q = \{\varepsilon, \text{A}, \text{AB}, \text{ABA}\}$

$$\begin{array}{c} \\ \varepsilon \\ \text{A} \\ \text{AB} \\ \text{ABA} \end{array} \begin{array}{c} \begin{array}{cccc} \varepsilon & \text{A} & \text{AB} & \text{ABA} \end{array} \\ \begin{pmatrix} \frac{2}{3} & \frac{1}{3} & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{2}{3} & 0 & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

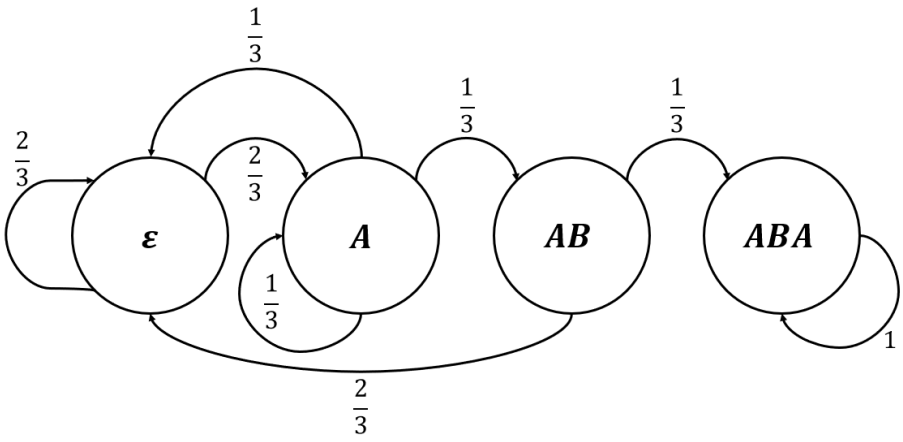

Figure 2: Transition matrix and Markov chain given example target string.

As a miscellaneous note, there are various points in the "Algorithm Development" section where we refer to the *progress* to a next state in, or through, a Markov chain. This term is used to describe the process of consecutive characters of s being found in r, and thus, moving rightward to subsequent states in the associated Markov chain (referring to Figure 2). We think of reaching the absorbing state

Algorithm Development

The motivation for the project's algorithm development comes from the fact that we can have two target strings of equal length, s_1 and s_2, that have different probabilities of appearing in a sequence of random characters, r. To demonstrate this, see the example below.

Example 1. To disprove that all strings of equal length have an equal probability of appearing in r, consider two strings s_1 and s_2, where s_1 = "ABA", and s_2 = "ABC". Say that the last 2 random characters of r are "AB", and neither s_1 or s_2 have appeared in r. Consider the two cases of us looking for the average hitting times of s_1 and s_2 below:

Case 1: We are looking at the average hitting time of s_1:

Observe that if the next character in r is "A", then all 3 characters of s_1 have appeared consecutively in r, and s_1 has occurred. If the next character is anything else, we must start over, as no consecutive characters of s_1 have been typed. We need all 3 characters of s_1 to occur consecutively for s_1 to occur.

Case 2: We are looking at the average hitting time of s_2:

Observe that if the next character in r is "C", then all 3 characters of s_2 have appeared consecutively in r, and s_2 has occurred. If the next character is "A", then s_2 does not occur, however we do not start over. We instead start from the first "A" in s_2, and we only need the last 2 characters of s_2 to appear for s_2 to occur fully. If neither "C" nor "A" is typed, then we start over, and need all 3 characters of s_1 to occur consecutively for s_1 to occur.

Thus, the average hitting time for s_2 is lower than it is for s_1, and we have disproved the original statement that all strings of equal length have the same average hitting time and equal probability of occurring within a given number of keystrokes. □

Note that this idea can be generalized when comparing any 2 strings of equal length. If one ends in a *prefix*, meaning that its last j characters match its first j characters, and the other does not, the string the ends in a prefix will have a higher average hitting time. We revisit this example in Figure 3 and Figure 4, where the hitting time PMFs of the two aforementioned example target strings are derived from one of the implemented algorithms.

To begin the algorithm development portion of the project, we create a program that takes the input of Σ, s, and a list of character-odds pairs O (that maps each $a \in \Sigma$ to a user-specified probability that a will occur at any given keystroke of r), and outputs the transition matrix P, of s. We prefer to derive P for any input of Σ, s, and O, opposed to the user inputting P directly, to enhance ease-of-use for those who are new to Markov chains. After this program is created, we develop additional ones to calculate the PMF of the hitting time of s, the average hitting time of s, and play a related multiplayer game. In this game, we calculate the probability that one target string s_1 appears before another target string s_2 (or, if applicable, the probability that both occur at the same time). We also extend this game to n players, each with their own target string s_i, for $i \in \{1, 2, \ldots n\}$. In the following subsections, the implementation of these programs are explained in greater detail.

Transition Matrix

To begin the program that outputs the transition matrix P of s with length l, we create a matrix of all zeros M. We the replace the elements of this $(l+1) \times (l+1)$ matrix, row-by-row, by iterating through both s and Σ. Before these iterations, we obtain a list of s's prefixes. This list of length l contains, for each $j \in \{1, 2, ..., l\}$, the first j characters of s. Using nested for-loops, we replace the elements of M. The outer loop iterates through the letters of s, and the inner loop iterates through the elements of Σ. In each operation, if the current element of Σ matches the current letter of s, then the current letter's associated probability (given by O) is added to element $M_{i,i-1}$ of M, where i is the index of the current letter of s. This result represents progress to the next state in Q, the associated state space of s.

If the current element of Σ does not match the current letter of s, then we must determine what state of the Markov chain to revert back to. This is done by first creating a temporary string s^*, containing the first i letters of s concatenated with the current letter of the iteration through Σ. We then find the length of longest prefix l^*, of s found at the end of s^*. If s^* does not end in a prefix of s, then $l^* = 0$. After determining the correct value of l^*, we add the current letter of Σ's associated probability (given by O) to element M_{i-1,l^*} of M.

After iterating through all of the letters of s, we add 1 to the bottom right element of M and return M. At this point, M matches P, the transition matrix representation of target string s.

Hitting Time PMF

To calculate the hitting time PMF, given by $p_H(t)$, we traverse values of t, calculating $Pr(H = t)$, the probability that the hitting time is exactly t, at each value. The expression for calculating $Pr(H = t)$ is given by $\lambda_0 P^t \vec{e_l} - \lambda_0 P^{t-1} \vec{e_l}$. Here, λ_0 is the initial distribution, and is set equal to the first row of the $(l+1) \times (l+1)$ identity matrix I. This is necessary because we assume that we are in the initial state before the first random keystroke. As well, $\vec{e_l}$ represents the last column of the identity matrix I, used to return only the probability that we are in the final (absorbing) state of s's Markov chain after t random keystrokes, opposed to the entire probability distribution.

This expression gives $Pr(H = t)$ by calculating $P(H \leq t) - P(H \leq t - 1)$. Note that in the implemented algorithm, this expression is factored into $\lambda_0 P^{t-1}(P - I)\vec{e_l}$. Using this, $p_H(t)$ is calculated by iterating through values of t until a stopping condition is met. Note that if $l = 0$, representing an empty string, then the first and only value of $p_H(t)$ is $p_H(0) = 1$. If $l \neq 0$, then $p_H(0) = 0$.

In order to calculate the Hitting Time PMF, we need P, derived using the transition matrix algorithm detailed above, s, and a *stopping probability* C. C is a user-defined stopping condition parameter. This stopping condition is met whenever $Pr(H = t) < C$, for $t \in \mathbb{N}_0$, and when $Pr(H = t-1) > Pr(H = t)$. The second condition is needed to ensure the stopping condition is not met when calculating H for low values of t. This condition, although an engineering choice, is supported by mathematical evidence. It assumes that the values in a target string's PMF do not increase as we increase values of t. We know this is true because, when increasing t, the probability that a target string appears somewhere in the first $t - 1$ characters also increases. Thus, the highest value in each string's PMF is found at l keystrokes, as there is no possibility for the target string to occur in the first $l - 1$ keystrokes. Note that, depending on the length of the target string, along with how many prefixes it ends in, this maximum PMF value can also be repeated at values of t keystrokes, where $l + 1 \leq t \leq 2l - 1$. It does not, however, repeat again at values of t where $t \geq 2l$.

In Figures 3 and 4 below, we compare the hitting time PMFs of the two target strings "ABA" and "ABC" given the alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ from Example 1. Observe that the maximum value (equal to $\frac{1}{27}$) is repeated in the PMF for "ABA" only at $t = 3$ and $t = 4$ keystrokes. This is because, when finding all of the different combinations of $r_1 ... r_5$ that result in "ABA" appearing at r_5, the combination $r_1 ... r_5 =$ "ABABA" is removed. It is removed because the target string, in this specific combination, appears at the third keystroke first. The maximum value of $\frac{1}{27}$ in the PMF for "ABC", however, repeats at $t = 3, 4,$ and 5 keystrokes as, if "ABC" appears for the first time at r_i keystrokes, for $i = 3, 4,$ or 5, there is no possibility for it to also appear somewhere in the first $i - 1$ keystrokes.

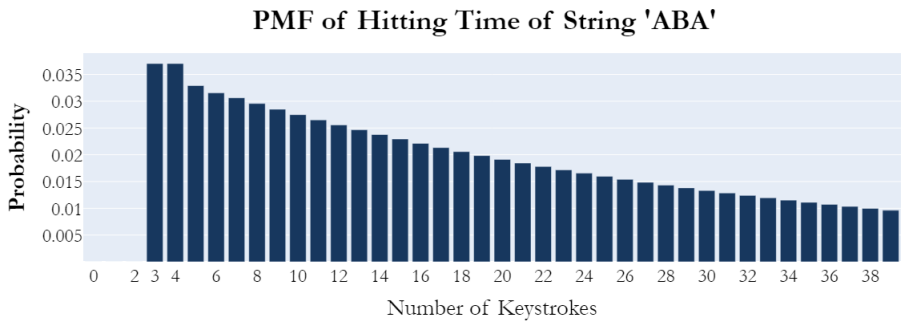

Figure 3: PMF for the target string "ABA".

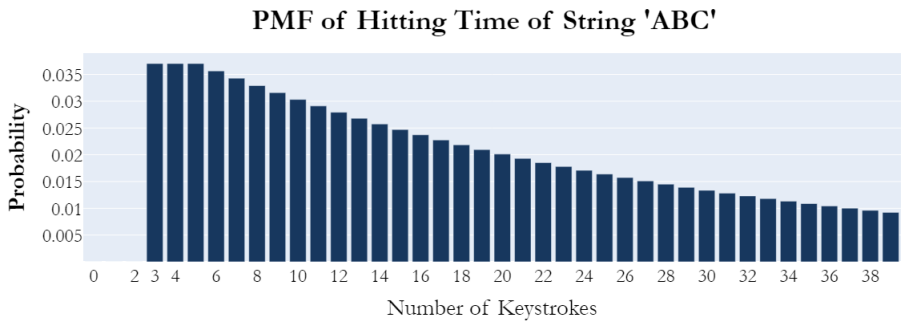

Figure 4: PMF for the target string "ABC".

Average Hitting Time

Note that an inefficient way to calculate the average hitting time of s is to calculate the hitting time's *Expected Value*, $E[H]$, given by $E[H] = \sum_{t=0}^{N} t * P(H = t)$, where N is the length of the derived PMF from the above algorithm. This, also, is just an approximation, as the true PMF of s as a domain of $\mathbb{N}_0$. Instead, we use a method explained by Jeffrey Rosenbluth [3] in his blog post titled "Martingale Measure" to obtain the exact average hitting time of a target string efficiently. The motivation for the post comes from martingale theory, where gamblers place equal wagers before each random keystroke is typed. Rosenbluth establishes an equivalence between the average hitting time of a target string and the amount of money that a casino would payout to all gamblers in a given scenario. We detail this scenario below.

Rosenbluth's Scenario

To begin, assume that there is a monkey typing a random sequence of characters r, where $r = r_1\ r_2\ ...\ r_t$, with each $r_i \in \Sigma$. Before r_1 is typed, one gambler comes to the table and bets 1 dollar that the monkey will type the first character of a target string, s. That is to say, this gambler bets 1 dollar that $r_1 = c_1$. If their bet wins, they are paid out the fair amount by the casino of k dollars, where $k = |\Sigma|$. Then, they roll over these winnings, now betting k dollars that $r_2 = c_2$. If this bet wins, they are paid out the fair amount of k^2 dollars. The gambler keeps betting, rolling over their winnings each time, until the entire target string, s, has been typed (in which case their total winnings are k^l dollars and the monkey stops typing characters), or any of their bets lose. If any of their bets lose, they are down their initial 1 dollar bet, and leave the table. New gamblers, one at a time, come to the table before each new r_i is typed, employing the same strategy as the first gambler. So, for example, the second gambler would bet 1 dollar that $r_2 = c_1$, rolling over their winnings of the bet wins, and leaving the table if the bet loses. Each bet that each gambler places (regardless of the stake or the potential payout) has an expected payout 0 dollars. Letting W_g represent the

return on a single bet made by a gambler and j represent the number of consecutive bets that each gambler has won since sitting down at the table, we have that $E[W_g] = -(k^j) + \frac{1}{k} * k^{j+1} = 0$.

The "big trick", as Rosenbluth puts it, is to say that the casino has expected gains of 0 dollars, regardless of time (given by the current length of r) and how many gamblers are at the table. Let W_c represent the net gain by the casino after s has been typed (at time t), H represent the hitting time of the string, and P_c represent the total payout that the casino distributes to winning gamblers. We have, $W_c = H - P_c$, as H dollars in total have been bet at time t. Because of the fact that, regardless of time, the casino has an expected net gain of 0, we know that $E[W_c] = 0$, and thus, by linearity of expectation, $E[W_c] = E[H] + E[-P_c] = 0$. Because P_c is a constant, and for each s, there is a pre-determined value of P_c that the casino will payout to all winning gamblers after s first appears, we have that $E[H] - P_c = 0$, and $E[H] = P_c$. So, the expected hitting time of a string is equal to the total amount that the casino pays out to all winning gamblers. Below, we examine two separate examples, using the familiar target strings of "ABC" and "ABA".

Total Payout of the Target String "ABC"

Let $r = r_1\ r_2 ... r_{t-2}\ r_{t-1}\ r_t = r_1\ r_2$...A B C, where no 3-character substring $r_i\ r_j\ r_k$ = A B C, except for when $i = t-2,\ j = t-1$, and $k = t$. All gamblers, except for the one that arrived to the table directly before time $t-2$, lose 1 dollar. The winning gambler however, wins 27 dollars. At time $t-2$, they wagered 1 dollar that r_{t-2} = A, and won that bet, turning their 1 dollar wager into 3 dollars. They then rolled over these winnings, betting 3 dollars that r_{t-1} = B, which netted them 9 dollars. Finally, they bet 9 dollars that r_t = C, giving them a total payout of 27 dollars at time t. Because this is the only gambler that won any money in this scenario, $P_c = 27$, and thus, $E[H] = 27$.

Total Payout of the Target String "ABA"

Let $r = r_1\ r_2 ... r_{t-2}\ r_{t-1}\ r_t = r_1\ r_2$...A B A, where no 3-character substring $r_i\ r_j\ r_k$ = A B A, except for when $i = t-2,\ j = t-1$, and $k = t$. Similar to before, the gambler that arrived at the table directly before time $t-2$ wins 27 dollars. However, the gambler that arrived directly before time t also is a winner. They bet, at time t, that the next character typed would be "A". It was, so they turned their 1 dollar bet into 3 dollars. Here, $P_c = 27 + 3 = 30$, so $E[H] = 30$.

Analysis

From the post, we see that all that is needed to calculate the average hitting time of a string are the lengths of all of its substrings that are both prefixes and suffixes For example, "AB**A**" has one ("A"), and "ABRACAD**ABRA**" has two ("A" and "ABRA"). In the algorithm, s is also considered a prefix of itself, so all nonempty strings, regardless of their content, end in at least one of their prefixes.

To calculate the average hitting time, each of the prefixes found at the end of s are traversed letter-by-letter. The product of the probabilities of each letter in a prefix occurring is then calculated, with the reciprocal of the product being taken. These reciprocals, for each prefix, are added together, returning the average hitting time of s.

Multiplayer Game

Imagine two competing strings, s_1 and s_2 with respective lengths of l_1 and l_2. Which one is more likely to appear first? Is it always one with the smaller average hitting time? Before discussing how we develop the program to answer these questions and output the probability that one target string appears before another, we must first understand how to form the Markov chain and state space Q, for a process with two target strings. The most significant difference between the states of a Markov chain modelling a process with one target string and one modelling two target strings is what the states themselves represent. Instead of a state

q_i representing the first i characters of s, it now is given by the coordinate pair (q_{1i}, q_{2j}), where q_{1i} indicates the first i characters of s_1 and q_{2j} indicates the first j characters of s_2, for $i \leq l_1$ and $j \leq l_2$. Thus, the progression to a new state after another character is typed by the hypothetical monkey is now determined by the longest prefixes of both s_1 and s_2 that appear at the end of r.

In this project, the first attempt at defining the state space of a Two Target String Markov Chain (TTSMC) was given by all $(l_1 + 1) * (l_2 + 1)$ pairs of prefixes of s_1 and s_2 (including the empty string, denoted by ε, and both complete target strings). So, the state space of the TTSMC where $s_1 =$ "ABA" and $s_2 =$ "ABC" is given below. Note that Q is still a set, however it is easier to understand Q's contents when it is organized into a 4×4 matrix.

$$Q = \begin{Bmatrix} (\varepsilon, \varepsilon) & (\mathrm{A}, \varepsilon) & (\mathrm{AB}, \varepsilon) & (\mathrm{ABA}, \varepsilon) \\ (\varepsilon, \mathrm{A}) & (\mathrm{A}, \mathrm{A}) & (\mathrm{AB}, \mathrm{A}) & (\mathrm{ABA}, \mathrm{A}) \\ (\varepsilon, \mathrm{AB}) & (\mathrm{A}, \mathrm{AB}) & (\mathrm{AB}, \mathrm{AB}) & (\mathrm{ABA}, \mathrm{AB}) \\ (\varepsilon, \mathrm{ABC}) & (\mathrm{A}, \mathrm{ABC}) & (\mathrm{AB}, \mathrm{ABC}) & (\mathrm{ABA}, \mathrm{ABC}) \end{Bmatrix}$$

The names of these states are formatted such that all characters before the comma represent the Markov chain's current progress towards s_1 appearing, and the characters after the comma represent the Markov chain's current progress towards s_2. Notice that there are several of states in Q that cannot actually be reached. For example, state AB, A can never be reached, as it is impossible for the first two characters of s_1 and only the first character of s_2 to be matched simultaneously at the end of r. So, the first part of the program is to create a new state space Q_v of only the states that can actually be reached given s_1 and s_2. An interesting discovery is that there can be at most $(l_1 + l_2 + 1)$ states in Q_v, which allows the algorithm to operate in significantly less time-complexity. This is because, for each of the prefixes, s_{1p}, given by the first i characters of s_1, where $1 \leq i \leq l_1$, there is exactly one prefix of s_2 that is a also a suffix of s_{1p}, representing the simultaneous progress towards a complete s_2. Likewise, for each of the prefixes, s_{2p}, given by the first j characters of s_2, where $1 \leq j \leq l_2$, there is exactly one prefix of s_1 that is a also a suffix of s_{2p}, representing the simultaneous progress towards a complete s_1. Because we always start in state $(\varepsilon, \varepsilon)$, we have, at most, $l_1 + l_2 + 1$ possible states that the TTSMC can actually reach. As well, Q_v now can contain multiple absorbing states, which are utilized to obtain the probability of one target string appearing before another, or, in some cases, the probability that the two target strings first appear at the same time.

After deriving Q_v, P, can now be formed. Similarly to the one target string case, the TTSMC's transition matrix is calculated by first creating a $|Q_v| \times |Q_v|$ matrix of all zeros, and then iterating through the states in Q_v to update one row at a time. With TTSMCs, however, determining which state the process transitions to after a new random character r_t is revealed is determined by how r_t affects the progression towards a complete s_1 and a complete s_2. Thus, for each letter in the alphabet, Σ, a temporary string s^* is calculated to determine the progression towards both s_1 and s_2. Note that s^* is the same temporary string in the TTSMC scenario as it is in the one target string scenario, where it matches the first i (where i is the index of our iteration through one of the target strings) letters of a target string concatenated with the current letter of our iteration through the elements of Σ. The only difference is it now does this two times, once for each of our target strings.

After the transition matrix P is derived, we then calculate its probability distribution by $\lambda_t = \vec{P_0} \times P^t$, where λ_0 is the initial distribution, set equal to the first row of the $|P| \times |P|$ identity matrix, and t is a user-specified parameter representing the number of random keystrokes. The probabilities in λ_t associated with the absorbing states of Q_v represent the probabilities that one target string appears before another or, in some cases, the probabilities that the two target strings appear at the same time. For large enough t, the probabilities of being in a non-absorbing state are essentially 0. Interestingly, given our example s_1 of "ABA" and s_2 of "ABC", despite the fact that, when $\Sigma = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$, the average hitting time of s_1 is 3 keystrokes larger than the average hitting time of s_2, they both have a .50 probability to occur first in the same r when modeled as a TTSMC.

Note that an alternative implementation of this algorithm allows for any number of target strings, $s_1, s_2, \ldots, s_n$, to be input, with the associated probability distribution at time t being returned. The only significant difference between the program for the TTSMC scenario and the n target string Markov

chain scenario is that any operations that were performed twice (once for s_1 and once for s_2) are instead performed n times, with for loops being utilized to do so. Each state, q_i, for this n target string scenario is given by the coordinate $(q_{1m_1}, q_{2m_2}, ..., q_{nm_n})$, where each q_{im_j} gives the first m_j characters of string i.

Conclusion & Discussion

The motivation for the work completed in this project came from the Infinite Monkey Theorem and the idea of the generation of long sequences of random characters. The goal of this project was to develop and implement algorithms to represent a target string as a Markov chain's transition matrix, calculate a target string's PMF and average hitting time, and play a related multiplayer game. The algorithms that were developed allow for user-specified target strings, alphabets, and random character probabilities, showcasing the flexibility of their implementation. To achieve this goal, an extensive understanding of Markov chains, their properties, and applications were needed and learned.

An interesting finding from this project were the similarities in time-complexity when playing the two-player multiplayer game versus deriving the PMF for just one target string. There are at most $(l_1 + l_2 + 1)$ valid states in the Markov chain that represents a multiplayer game with two target strings s_1 and s_2 with lengths l_1 and l_2. This finding is interesting because there is not a significant time-complexity difference when compared to the number of valid states in the Markov chain that represents one target string s_1 with length l_1. Due to the additive nature of the number of valid states when increasing the number of players in the game, it is plausible to test the developed algorithms for an incredibly large number of players, each with long target strings.

Future Work

This project served as an exploration into Markov chains, probability, and the infinite generation of random characters. Because of this explorative aspect, there is a plethora of possible future work to expand on what has been performed here. For one, the algorithms that were developed can be improved for further efficiency and readability. Due to the nature of the project, the limited time available was spent mostly developing and implementing the algorithms rather than fully optimizing them. There are certainly optimizations that can be made, including using different data structures and methods to achieve faster runtimes or obtaining *exact* values for the probabilities that each player wins the related multiplayer game (due to the parameter of t representing the number of keystrokes typed, these methods return the probabilities that each player wins at character t, rather than if infinite characters were to be typed. Although t can be set to a very large integer, say $2^{80} + 1$, the output values are still an approximation).

As well, a front-end application can be created that allows for an easy interaction with the back-end code that was written. Such a tool can be useful for gamblers playing a game that involves a random sequence of outcomes either at a casino or online. They could leverage the algorithms to simulate either single-player or multiplayer games to observe their expected payout, or play these games with their friends. A front end tool can also be useful for someone who simply wants to learn more about probability and Markov chains. The easy-to-understand and lighthearted application of the infinite generation of random characters (and the humorous example of the Infinite Monkey Theorem) makes a potential front end application beneficial to these beginners.

References

[1] Christopher R. S. Banerji, Toufik Mansour, and Simone Severini. 2013. A notion of graph likelihood and an infinite monkey theorem. *Journal of Physics A: Mathematical and Theoretical* 47, 3 (2013). https://doi.org/10.1088/1751-8113/47/3/035101

[2] J. R. Norris. 1998. *Markov chains.* Cambridge University Press, Cambridge, England.

[3] Jeffrey Rosenbluth. 2014. Monkey typing ABRACADABRA. Retrieved from https://martingalemeasure.wordpress.com/2014/02/02/monkey-typing-abracadabra-14/