Markov Chain Monte Carlo

Elliott Liu, Yumeng Guo, Yumo Ma, Mingxin Du

June 2025

1 Introduction

The Metropolis-Hastings (M-H) algorithm, a Markov chain Monte Carlo (MCMC) method, is one of the most popular techniques used by statisticians today. It is primarily used as a way to simulate observations from unwieldy distributions (Hitchcock, 2003). New samples are added to the sequence in two steps: first a new sample is proposed based on the previous sample, then the proposed sample is either added to the sequence or rejected depending on the value of the probability distribution at that point (Wikipedia, 2005).

For this final project, our group chose the emphasis of Coding and aims to improve the accuracy of decrypting a message based on the Metropolis-Hastings algorithm. In the original setup, the decrypted code defines a sample space of 56 characters, consisting of lowercase and uppercase English letters only, and iteratively proposes new permutations by swapping two characters at a time to explore better likelihoods of decoded outputs.

However, such limited permutations often fail to recover natural English phrases; for instance, a word like "while" may be incorrectly decoded as "ahile." To address this, we propose two modifications: (1) in every step, we switch 5 letters in the current permutation at a time instead of 2 letters, and (2) we introduce a temperature parameter to alter the acceptance rate. The proposal function is modified to generate more diverse permutations, potentially improving the search efficiency and output accuracy.

Preliminary testing suggests that the first modification did not improve the result of the decoding because the deciphered text was still distorted, and many common English patterns were lost; however, the second modification does help the algorithm capture common English character sequences, resulting in more readable decoded messages.

The project is organized as follows:

- Section 2 introduces the encryption method and provides examples of the permutation of the alphabet, explains the likelihood function, and describes how training is performed.
- Section 3 discusses the theory and implementation of the Metropolis–Hastings algorithm, including our proposed modification.
- Section 4 presents results comparing the performance of two modified algorithms to the standard version.
- The appendix contains all relevant code and data files.

2 Message Encryption/Decryption

This section outlines the process of encrypting and decrypting text messages using permutations on alphabets and evaluating the likelihood that a decrypted text is valid English. These mechanisms are implemented in the provided Python code, particularly in `deciphering_utils.py`, `scramble_text.py`, and `run_deciphering.py`, which use a Markov Chain Monte Carlo (MCMC) algorithm to decode encrypted messages. We describe the permutation-based encryption process, the likelihood function for assessing English text, and their implementation details, drawing from the code and supporting documents.

2.1 Permutations on Alphabets

An alphabet Σ is a finite set of characters, such as the 26 lowercase letters $\{a, b, \ldots, z\}$ or the 56-character set (lowercase, uppercase, digits, and symbols, including spaces) used in our code, as specified in the README document. A permutation $\sigma : \Sigma \to \Sigma$ is a bijective mapping that reorders the alphabet. For encryption, a plaintext message $m = m_1 m_2 \cdots m_n$ is transformed into a ciphertext $c = \sigma(m_1)\sigma(m_2) \cdots \sigma(m_n)$, where each character $m_i \in \Sigma$ is replaced by $\sigma(m_i)$. Decryption applies the inverse permutation σ^{-1}, so $\sigma^{-1}(c_i) = m_i$. In our examples, we use a 26-letter alphabet for simplicity, with non-alphabetic characters (e.g., spaces, uppercase letters) mapping to themselves, though the code supports the full 56-character set.

The `scramble_text.py` script implements encryption by generating a random permutation map via `generate_random_permutation_map` and applying it to a text file. For decryption, the `propose_a_move` function in `deciphering_utils.py` generates new permutations by swapping two characters, with transition probability:

$$q(\sigma, \sigma') = \binom{|\Sigma|}{2}^{-1} = \frac{1}{\binom{56}{2}} = \frac{1}{1540}, \tag{1}$$

for the 56-character alphabet, reflecting a simple transposition (Connor, 2003). Below is a simplified version of `propose_a_move`:

```
def propose_a_move(state):
    new_state = deepcopy(state)
    p_map = new_state["permutation_map"]
    keys = list(p_map.keys())
    a, b = random.sample(keys, 2)
    p_map[a], p_map[b] = p_map[b], p_map[a]
    new_state["permutation_map"] = p_map
    return new_state
```

Listing 1: Simplified `propose_a_move` for generating permutations

Example 1: Small Alphabet

Consider a small alphabet $\Sigma = \{a, b, c, d\}$ and permutation:

$$\sigma = \begin{pmatrix} a & b & c & d \\ c & d & a & b \end{pmatrix}. \tag{2}$$

The plaintext “bad cab” encrypts to:

- $b \to d, a \to c, d \to b$, (space unchanged), $c \to a, a \to c, b \to d$,

yielding ciphertext "dcb acd." Decryption with $\sigma^{-1} : c \to a, d \to b, a \to c, b \to d$ recovers "bad cab."

Example 2: Realistic Alphabet

For $\Sigma = \{a, \ldots, z\}$, consider a permutation with swaps $t \leftrightarrow x$, $h \leftrightarrow w$, others unchanged, generated by repeated calls to `propose_a_move`. The plaintext "pstat is hard" (lowercase for simplicity, though original is "PSTAT") encrypts to:

- $p \to p, s \to s, t \to x, a \to a, t \to x$, (space unchanged), $i \to i, s \to s$, (space unchanged), $h \to w$, $a \to a, r \to r, d \to d$,

yielding "psxax is ward." Decryption with $\sigma^{-1} : x \to t, w \to h$, others unchanged, restores "pstat is hard."

These examples illustrate how permutations enable reversible encryption, with the code efficiently generating and testing permutations for decryption.

2.2 The Likelihood of Text

The likelihood function $\mu_T(\sigma)$ quantifies how likely a decrypted text σ_T is to be valid English, using a first-order Markov model:

$$\mu_T(\sigma) = \prod_{k=1}^{n} \ell(\sigma_{t_{k-1}}, \sigma_{t_k}), \tag{3}$$

where $\ell(a, b)$ is the probability of character b following a. For numerical stability, the code computes the log-likelihood:

$$\log \mu_T(\sigma) = \sum_{k=2}^{n} \log \ell(\sigma_{t_{k-1}}, \sigma_{t_k}) + \log P(\sigma_{t_0}), \tag{4}$$

where $P(\sigma_{t_0})$ is the initial character's frequency. The `compute_log_probability_by_counts` function in `deciphering_utils.py` implements this, using a transition matrix trained on "War and Peace" (via `-i` in `run_deciphering.py`). Below is a simplified version:

```
def compute_log_probability_by_counts(transition_counts, text,
    permutation_map, char_to_ix, frequency_statistics, transition_matrix):
    c0_perm_idx = char_to_ix[permutation_map[text[0]]]
    p = np.log(frequency_statistics[c0_perm_idx] + 1e-12)
    perm_indices = [char_to_ix[c2] for c1, c2 in permutation_map.items()]
    permuted_tm = transition_matrix[perm_indices, :][:, perm_indices]
    p += np.sum(transition_counts * np.log(permuted_tm + 1e-12))
    return p
```

Listing 2: Simplified `compute_log_probability_by_counts` for likelihood

The transition matrix is built by `compute_statistics`, which processes "War and Peace" to count character transitions (`compute_transition_counts`) and normalize them into probabilities. The code adds a small constant (`1e-12`) to avoid zero probabilities for rare transitions, a practical smoothing technique.

Example

Consider the ciphertext T = "psxax is ward" with $\Sigma = \{a, \ldots, z\}$ for simplicity, though the code supports 56 characters, including spaces and uppercase letters, which map to themselves. Test two permutations:

- $\sigma_1 : x \to t, w \to h$, others unchanged, yields $\sigma_1 T$ = "pstat is hard".
- $\sigma_2 : x \to n, w \to m$, others unchanged, yields $\sigma_2 T$ = "psnax is mard".

For $\sigma_1 T$, key transitions like $t \to a$ (in "stat"), $i \to s$ (in "is"), and $h \to a$ (in "hard") are common in English, with high probabilities (e.g., $\ell(t, a) \approx 0.08$, $\ell(i, s) \approx 0.1$, $\ell(h, a) \approx 0.03$, estimated from "War and Peace" via `compute_statistics`. The log-likelihood, computed by `compute_log_probability_by_counts`, is high. For $\sigma_2 T$, transitions like $n \to a$ (in "snax") and $m \to a$ (in "mard") are less likely ($\ell(n, a) \approx 0.005$, $\ell(m, a) \approx 0.004$, with smoothing via `1e-12` for rare transitions, resulting in a lower logarithmic likelihood. The initial probability $P(p)$ is similar for both, as both start with "p". The MCMC algorithm, implemented in `metropolis_hastings.py`, iteratively proposes permutations and favors σ_1 over time because:

$$(p_2 - p_1) > \log u, \tag{5}$$

where $p_1 = \log \mu_T(\sigma)$, $p_2 = \log \mu_T(\sigma')$, and $u \sim \text{Uniform}(0, 1)$, assuming a temperature of 1 for simplicity (the code includes simulated annealing).

Training and Practical Considerations

Training on "War and Peace" provides robust transition probabilities but may bias toward formal English. The 56-character alphabet increases computational complexity, requiring efficient matrix operations. Smoothing ensures numerical stability but may skew probabilities for rare characters. A potential modification, explored in Section 3, could involve proposing multiple swaps (as in `deciphering_utils_switch5.py`'s `propose_a_move`, which swaps five characters) to accelerate convergence, impacting how permutations are tested.

These mechanisms, rooted in the code, enable the MCMC algorithm to decode encrypted messages by maximizing the likelihood of valid English text, bridging to Section 3.

3 Markov Chain Monte Carlo

Our decryption algorithm is based on the Metropolis–Hastings method, which is a specific Markov Chain Monte Carlo (MCMC) algorithm. In this project, the state space consists of all possible permutations of the English alphabet, and the likelihood function μ reflects how likely a decoded message is to resemble English.

A new permutation σ' is generated from the current permutation σ with a probability of $q(x, y)$, and the encrypted message is decoded using the optimal permutation σ, and the optimal permutation is determined by the likelihood function. A likelihood function $\mu(\sigma)$ is computed based on the accuracy of the decoded message. This likelihood function is derived by studying and analyzing the English language usage habits from the book War and Peace. The acceptance probability: $a(\sigma, \sigma') = min(1, \frac{\mu(\sigma')q(\sigma', \sigma)}{\mu(\sigma)q(\sigma, \sigma')})$ means that if the likelihood of the new permutation $\mu(\sigma')$ is greater than the current permutation $\mu(\sigma)$, it will automatically move to σ'; however, if the likelihood of the new permutation $\mu(\sigma')$ is smaller than the current permutation $\mu(\sigma)$, it may move to the new permutation with the probability of $\frac{\mu(\sigma')q(\sigma', \sigma)}{\mu(\sigma)q(\sigma, \sigma')}$; otherwise it will stay in the current permutation.

For our algorithm, the idea is to construct a Markov Chain that converges to its stationary distribution ($\lim_{k\to\infty} p_k(x, y) = \pi(y)$ for all $x, y \in S$). This is guaranteed under several conditions: the chain is finite, irreducible, and aperiodic (Connor, 2003).

3.1 How it Works

Check Conditions

The total number of permutations is $26! \approx 4 * 10^{26}$. Each permutation represents a unique state in the Markov chain. Although the state space is large, it is finite and countable, implying that the Markov chain is finite. Since any permutation can be accessible from any other permutations by switching letters, the Markov chain is irreducible. Each permutation has probability to reject transiting to other states, which implies that each state has a self-loop. This ensures the Markov Chain is aperiodic. Therefore, the Markov Chain is finite, irreducible, and aperiodic, and it converges to its stationary distribution.

Local Balance Equation Verification

$$\pi(\sigma)p(\sigma, \sigma') = \pi(\sigma')p(\sigma', \sigma) \quad \sigma \neq \sigma' \quad (Connor, 2003)$$

Let $\mu(\sigma)$ denote the unnormalized version of the stationary distribution $\pi(\sigma)$, and the transition probability $p(\sigma, \sigma') = q(\sigma, \sigma')a(\sigma, \sigma')$

$$\mu(\sigma)p(\sigma, \sigma') = \mu(\sigma')p(\sigma', \sigma)$$

Since

$$\mu(\sigma)q(\sigma, \sigma')a(\sigma, \sigma') = \mu(\sigma)q(\sigma, \sigma')min(1, \frac{\mu(\sigma')q(\sigma', \sigma)}{\mu(\sigma)q(\sigma, \sigma')}) = min(\mu(\sigma)q(\sigma, \sigma'), \mu(\sigma')q(\sigma', \sigma))$$

$$\mu(\sigma')q(\sigma', \sigma)a(\sigma', \sigma) = \mu(\sigma')q(\sigma', \sigma)min(1, \frac{\mu(\sigma)q(\sigma, \sigma')}{\mu(\sigma')q(\sigma', \sigma)}) = min(\mu(\sigma')q(\sigma', \sigma), \mu(\sigma)q(\sigma, \sigma'))$$

we have

$$\mu(\sigma)p(\sigma, \sigma') = \mu(\sigma')p(\sigma', \sigma)$$

Local balance equation holds!
Since the local balance equation holds, it follows that the chain is time-reversible with respect to the stationary distribution π. This proves that the Metropolis-Hastings algorithm can be used to construct a time-reversible Markov chain whose stationary distribution is

$$\pi(j) = \frac{b(j)}{B}, \quad j = 1, 2, \ldots \quad \text{where } B = \sum_{j=1}^{\infty} b(j)$$

(Ross, 2019).

3.2 Proposed modification

Motivation

Initial experiments using the standard Metropolis-Hastings (MH) algorithm with a basic two-character provided by Professor Shkolnik swap proposal yielded moderate results but suffered from slow convergence

and limited exploration. To address this, we implemented more aggressive proposals involving the swapping of three or five character pairs (see `deciphering_utils_switch5.py`), intending to accelerate mixing. However, these larger proposals significantly disrupted the current state, leading to extremely low acceptance rates. This resulted in the Markov chain becoming trapped in low-likelihood regions, as evident from the poor-quality outputs such as "Aface!s Aufer Peavy" instead of "SpaceX's Super Heavy" after thousands of iterations (see `switch5.txt`).

To overcome this stagnation, we shifted focus from modifying the *proposal distribution* to altering the *acceptance criterion*. Inspired by *simulated annealing*, we introduced a **temperature parameter** to make the algorithm more permissive toward worse solutions in early stages, enabling broader exploration before settling into local optima.

Implementation Details

The standard Metropolis-Hastings acceptance probability is defined as:

$$\alpha(x \to x') = \min\left(1, \frac{\pi(x')}{\pi(x)}\right)$$

We modified this to incorporate a temperature T as follows:

$$\alpha_T(x \to x') = \min\left(1, \left(\frac{\pi(x')}{\pi(x)}\right)^{1/T}\right)$$

This was implemented in a custom `acceptance_prob` function in `metropolis_hastings.py`. We compute the log-likelihood ratio between the proposed and current state, scale it by $1/T$, and compare it to $\log(u)$, where $u \sim \text{Uniform}(0, 1)$.

A cooling schedule was introduced:

$$T = \frac{10}{\sqrt{\text{iteration} + 1}}$$

This ensures high exploration at the beginning and gradual convergence over time. A lower bound (e.g., $T_{\min} = 0.1$) may be applied to prevent premature convergence.

Theoretical Considerations

Introducing a temperature parameter breaks the *detailed balance condition* when $T \neq 1$, rendering the Markov chain *non-reversible*. While detailed balance ensures convergence to the stationary distribution, our aim is high-quality decoding rather than exact posterior sampling. This trade-off is acceptable, particularly in combinatorial spaces such as permutations, where the landscape is rugged and multimodal.

Simulated annealing techniques are well-known for escaping local optima by allowing occasional downhill moves, which helps in reaching more globally optimal configurations.

Expected Outcomes

The modified strategy is expected to yield the following benefits:

- **Higher Acceptance Rates**: Especially during early iterations, the algorithm becomes more exploratory and is more likely to accept worse moves.
- **Improved Convergence**: The chain avoids getting stuck in poor-quality local optima, improving the probability of reaching better permutations.
- **Higher Output Quality**: The deciphered text quality improves. For example, "Aface!s Aufer Peavy" evolves into "SpaceX's Super Heavy rocket exploded during a test flight".
- **Smoother Log-Posterior Trends**: The log-posterior should increase and stabilize, reflecting convergence and improved mixing.
- **Adaptability**: The temperature schedule allows the algorithm to adjust its behavior depending on the difficulty of the input text.
- **Trade-offs**: The approach may require more iterations, and improperly tuned cooling schedules may result in premature convergence or excessive runtime.

Experimental Comparison

Our results confirm that the temperature-based acceptance approach significantly outperforms both the baseline and the switch5 variant. While switch5 made more radical proposals, its near-zero acceptance rate rendered the Markov chain almost static. In contrast, our modified approach enabled steady progress and produced coherent English phrases within a few thousand iterations.

This demonstrates the effectiveness of modifying the acceptance probability to balance exploration and exploitation—an important consideration for future improvements in MCMC-based decoding algorithms.

4 Conclusion

In this project, our group modified the proposal step in the Metropolis-Hastings algorithm to switch 5 letters at a time instead of 2. The original idea behind switching 5 letters is to allow larger modifications in the states (permutations); therefore, the algorithm can explore the sample space more broadly to get the optimal permutation. However, the result differs from what we expected because the result generated by switching 5 letters at a time is even worse than that generated by switching 2 letters at a time. The deciphered text was more distorted, and many common English patterns were lost. For example, the word "Aface" is supposed to be deciphered as "Space", "Ruesdays" as "Tuesdays", "bor" as "for", and "develofing" as "developing", and so on.

A likely reason is that switching 5 letters may destroy the common letter pairs and original language structures, such as "th", "re", "er", "ed", and "st". Therefore, the likelihood that is derived by studying and analyzing the English language usage habits from the book can be very low, which leads to low acceptance rate. This prevents the chain from converging to its stationary distribution and leads to random or worse permutations being accepted.

4.1 Decryption Effect Comparison

We compared the improved Metropolis-Hastings (MH) algorithm with the original version, which only swaps two characters and does not incorporate a temperature parameter, in terms of decryption effectiveness. Both

algorithms were run independently using the same encrypted text as input to assess their ability to recover the original English content.

In the original algorithm, while some character arrangements approached correct positions locally, the overall output remained far from natural language. The resulting text frequently included non-English words, broken sentences, and various garbled symbols, rendering it fragmented and difficult to read.

In contrast, the temperature-controlled MH algorithm, employing a simulated annealing strategy, produced output text significantly closer to real English. For example:

"SpaceX's Super Heavy booster and Starship rocket exploded during a test flight. . . "

Although a few character substitutions persisted (e.g., "Plon Busks" instead of "Elon Musk"), the subject-verb structure was clear, the semantics were coherent, and proper nouns were correctly recovered. Moreover, outputs across multiple independent runs exhibited high consistency, indicating strong stability and convergence of the improved algorithm.

4.2 Decryption Effect Comparison

We also compared the acceptance rates and runtime performance of the two algorithms. The results are summarized in Table 1.

Metric	**Original MH**	**Simulated Annealing MH (with temperature)**
Initial Acceptance Rate	∼0.02–0.05	∼0.10–0.30
Total Iterations	∼1500 (early termination)	∼5000 (completed normally)
Output Readability	Unstable, often garbled	Stable, close to natural language
Decryption Consistency	Highly random	Nearly identical across runs

Table 1: Comparison of original MH and simulated annealing MH algorithms.

The original MH algorithm only accepts proposals that increase the log-posterior probability, making it prone to getting stuck in local optima. Once trapped, escape is nearly impossible. Conversely, the simulated annealing MH algorithm permits acceptance of worse proposals in the early phase and gradually converges later, offering greater exploration capacity and more stable overall performance.

4.3 Analysis and Summary

We believe the core advantage of simulated annealing lies in its dynamic balance between exploration and convergence:

- In the early phase (high temperature), the algorithm can accept worse permutations, helping it escape local optima.
- In the later phase (low temperature), the acceptance behavior becomes more selective, focusing on higher-probability regions.

Unlike the original MH algorithm, which only performs local adjustments, the improved version is capable of "broad exploration first, stable convergence later." Moreover, although it requires more runtime, the overall decryption quality is clearly superior. This suggests that for combinatorial optimization problems

like character permutation, improving the "quality of each move" is more critical than merely accelerating iterations.

5 Strengths and Limitations

5.1 Strengths

- Output resembles daily English we use, with coherent semantics.
- Significantly higher acceptance rate, leading to more active Markov chains.
- High stability, with consistent results across runs.
- Easy to combine with other strategies (e.g., block-based proposals).

5.2 Limitations

- Longer run-time compared to the original MH.
- Temperature scheduling requires careful tuning (e.g., initial temperature, cooling rate).
- Some decoding errors remain, and full recovery of the original text is not guaranteed.

This improvement demonstrates the practical value of simulated annealing in Markov Chain Monte Carlo sampling. By dynamically adjusting the acceptance probability through a temperature parameter, the approach effectively overcomes the limitations of the original MH algorithm.

We recommend adopting temperature control as a fundamental design strategy in future tasks involving combinatorial optimization or language modeling. Additionally, the technique can be further enhanced by integrating more sophisticated proposal functions to improve sampling efficiency and decryption quality.

Appendix

A Modified Metropolis-Hastings with Simulated Annealing

```
import numpy as np
import time
import shutil
import random

# This version includes Simulated Annealing logic
def metropolis_hastings(initial_state, proposal_function, log_density, iters=1000, print_every=10, t
    """
    Runs a metropolis hastings algorithm with simulated annealing.
    """

    p1 = log_density(initial_state)
    errors = []
```

```
cross_entropies = []

state = initial_state
cnt = 0
accept_cnt = 0
error = -1
states = [initial_state]
it = 0

start_time = time.time()

while it < iters:
    new_state = proposal_function(state)
    p2 = log_density(new_state)

    u = random.random()
    cnt += 1

    # Acceptance condition with temperature
    if (p2 - p1) / temperature > np.log(u):
        state = new_state
        it += 1
        accept_cnt += 1
        p1 = p2

        # Cool down the temperature
        temperature *= cooling_rate

        cross_entropies.append(p1)
        states.append(state)
        if error_function is not None:
            error = error_function(state)
            errors.append(error)

        if it % print_every == 0:
            acceptance = float(accept_cnt) / float(cnt)
            elapsed = time.time() - start_time
            print(f"Iter: {it}/{iters} | Entropy: {p1:.2f} | Temp: {temperature:.4f} | Acceptanc

            if acceptance < tolerance and it > (iters / 10):
                print("Acceptance rate too low, stopping early.")
                break

            # Reset counters for acceptance rate calculation
            cnt = 0
            accept_cnt = 0
```

```
    if pretty_state is not None:
        print("\n" + pretty_state(state))

    if error_function is None:
        errors = None

    return states, cross_entropies, errors
```

B Modified Proposal Function with 5-character Random Switch

```
import numpy as np
import random
from utils import *

def compute_log_probability(text, permutation_map, char_to_ix, frequency_statistics, transition_matr
    """
    Computes the log probability of a text under a given permutation map (switching the
    charcter c from permutation_map[c]), given the text statistics

    Arguments:
    text: text, list of characters

    permutation_map[c]: gives the character to replace 'c' by

    char_to_ix: characters to index mapping

    frequency_statistics: frequency of character i is stored in frequency_statistics[i]

    transition_matrix: probability of j following i

    Returns:
    p: log likelihood of the given text
    """
    t = text
    p_map = permutation_map
    cix = char_to_ix
    fr = frequency_statistics
    tm = transition_matrix

    i0 = cix[p_map[t[0]]]
    p = np.log(fr[i0])
    i = 0
    while i < len(t)-1:
        subst = p_map[t[i+1]]
```

```
        i1 = cix[subst]
        p += np.log(tm[i0, i1])
        i0 = i1
        i += 1

    return p

def compute_transition_counts(text, char_to_ix):
    """
    Computes transition counts for a given text, useful to compute if you want to compute
    the probabilities again and again, using compute_log_probability_by_counts.

    Arguments:
    text: Text as a list of characters

    char_to_ix: character to index mapping

    Returns:
    transition_counts: transition_counts[i, j] gives number of times character j follows i
    """
    N = len(char_to_ix)
    transition_counts = np.zeros((N, N))
    c1 = text[0]
    i = 0
    while i < len(text)-1:
        c2 = text[i+1]
        transition_counts[char_to_ix[c1],char_to_ix[c2]] += 1
        c1 = c2
        i += 1

    return transition_counts

def compute_log_probability_by_counts(transition_counts, text, permutation_map, char_to_ix, frequenc
    """
    Computes the log probability of a text under a given permutation map (switching the
    charcter c from permutation_map[c]), given the transition counts and the text

    Arguments:

    transition_counts: a matrix such that transition_counts[i, j] gives the counts of times j follow
                       see compute_transition_counts

    text: text to compute probability of, should be list of characters

    permutation_map[c]: gives the character to replace 'c' by
```

```
    char_to_ix: characters to index mapping

    frequency_statistics: frequency of character i is stored in frequency_statistics[i]

    transition_matrix: probability of j following i stored at [i, j] in this matrix

    Returns:

    p: log likelihood of the given text
    """
    c0 = char_to_ix[permutation_map[text[0]]]
    p = np.log(frequency_statistics[c0])

    p_map_indices = {}
    for c1, c2 in permutation_map.items():
        p_map_indices[char_to_ix[c1]] = char_to_ix[c2]

    indices = [value for (key, value) in sorted(p_map_indices.items())]

    p += np.sum(transition_counts*np.log(transition_matrix[indices,:][:, indices]))

    return p

def compute_difference(text_1, text_2):
    """
    Compute the number of times to text differ in character at same positions

    Arguments:

    text_1: first text list of characters
    text_2: second text, should have same length as text_1

    Returns
    cnt: number of times the texts differ in character at same positions
    """

    cnt = 0
    for x, y in zip(text_1, text_2):
        if y != x:
            cnt += 1

    return cnt

def get_state(text, transition_matrix, frequency_statistics, char_to_ix):
    """
    Generates a default state of given text statistics
```

```
    Arguments:
    pretty obvious

    Returns:
    state: A state that can be used along with,
           compute_probability_of_state, propose_a_move,
           and pretty_state for metropolis_hastings

    """
    transition_counts = compute_transition_counts(text, char_to_ix)
    p_map = generate_identity_p_map(char_to_ix.keys())

    state = {"text" : text, "transition_matrix" : transition_matrix,
             "frequency_statistics" : frequency_statistics, "char_to_ix" : char_to_ix,
            "permutation_map" : p_map, "transition_counts" : transition_counts}

    return state

def compute_probability_of_state(state):
    """
    Computes the probability of given state using compute_log_probability_by_counts
    """

    p = compute_log_probability_by_counts(state["transition_counts"], state["text"], state["permutat
                                          state["char_to_ix"], state["frequency_statistics"], state[

    return p

def propose_a_move(state):
    """
    Proposes a new move for the given state,
    by moving one step (randomly swapping two characters)
    """
    new_state = {}
    for key, value in state.items():
        new_state[key] = value
    p_map = dict(state["permutation_map"])
    keys = list(p_map.keys())

    keys_to_swap = random.sample(keys, 5)

    values_to_swap = [p_map[k] for k in keys_to_swap]

    random.shuffle(values_to_swap)
```

```
    for i, key in enumerate(keys_to_swap):
        p_map[key] = values_to_swap[i]

    new_state["permutation_map"] = p_map
    return new_state

def pretty_state(state, full=True):
    """
    Returns the state in a pretty format
    """
    if not full:
        return pretty_string(scramble_text(state["text"][1:200], state["permutation_map"]), full)
    else:
        return pretty_string(scramble_text(state["text"], state["permutation_map"]), full)
```

Supplements

New Guess Results from modified Metropolis-Hastings with Simulated Annealing

========== Best Guesses : ==========

Guess 1:

Space's Super Heavy booster and Starship rocket exploded during a test flight on Ruesday, the third consecutive setback for Plon Busks company.

Ruesdays un-crewed mission was the ninth test flight. Rhe system suffered prior explosions in Nanuary and Barch. Rogether, the Starship rocket and Super Heavy booster are around 100 feet tall when stacked for a launch.

Space' has been developing the Starship system to transport people and equipment around Parth, and to the Boon. Busk also wants Space' to use Starship to colonize Bars.

" livestream of Ruesdays test flight, which ran on the Space' website and on social media, showed the first-stage booster exploding and the second-stage Starship spacecraft undergoing a major fuel leak, then spinning out of control and blowing up during reentry.

****

Guess 2:

Space's Super Heavy booster and Starship rocket exploded during a test flight on Ruesday, the third consecutive setback for Plon Busks company.

Ruesdays un-crewed mission was the ninth test flight. Rhe system suffered prior explosions in Nanuary and Barch. Rogether, the Starship rocket and Super Heavy booster are around 100 feet tall when stacked for a launch.

Space' has been developing the Starship system to transport people and equipment around Parth, and to the Boon. Busk also wants Space' to use Starship to colonize Bars.

" livestream of Ruesdays test flight, which ran on the Space' website and on social media, showed the first-stage booster exploding and the second-stage Starship spacecraft undergoing a major fuel leak, then spinning out of control and blowing up during reentry.

****

Guess 3:

Space's Super Heavy booster and Starship rocket exploded during a test flight on Ruesday, the third consecutive setback for Plon Busks company.

Ruesdays un-crewed mission was the ninth test flight. Rhe system suffered prior explosions in Nanuary and Barch. Rogether, the Starship rocket and Super Heavy booster are around 100 feet tall when stacked for a launch.

Space' has been developing the Starship system to transport people and equipment around Parth, and to the Boon. Busk also wants Space' to use Starship to colonize Bars.

" livestream of Ruesdays test flight, which ran on the Space' website and on social media, showed the first-stage booster exploding and the second-stage Starship spacecraft undergoing a major fuel leak, then spinning out of control and blowing up during reentry.

****

New Guess Results from Modified Proposal Function with 5-character Random Switch

Guess 1:

Aface!s Aufer Peavy pooster and Atarshif rocket e-floded during a test blight on Ruesday, the third consecutive setpack bor Mlon Busks comfany.

Ruesdays un'crewed mission was the ninth test blight. Rhe system subbered frior e-flosions in Nanuary and Barch. Rogether, the Atarshif rocket and Aufer Peavy pooster are around 100 beet tall when stacked bor a launch.

Aface! has peen develofing the Atarshif system to transfort feofle and equifment around Marth, and to the Boon. Busk also wants Aface! to use Atarshif to coloniGe Bars.

" livestream ob Ruesdays test blight, which ran on the Aface! wepsite and on social media, showed the birst'stage pooster e-floding and the second'stage Atarshif sfacecrabt undergoing a major buel leak, then sfinning out ob control and plowing uf during reentry.

****

Guess 2:

Aface!s Aufer Peavy pooster and Atarshif rocket e-floded during a test blight on Ruesday, the third consecutive setpack bor Mlon Busks comfany.

Ruesdays un'crewed mission was the ninth test blight. Rhe system subbered frior e-flosions in Nanuary and Barch. Rogether, the Atarshif rocket and Aufer Peavy pooster are around 100 beet tall when stacked bor a launch.

Aface! has peen develofing the Atarshif system to transfort feofle and equifment around Marth, and to the Boon. Busk also wants Aface! to use Atarshif to coloniGe Bars.

" livestream ob Ruesdays test blight, which ran on the Aface! wepsite and on social media, showed the birst'stage pooster e-floding and the second'stage Atarshif sfacecrabt undergoing a major buel leak, then sfinning out ob control and plowing uf during reentry.

****

Guess 3:

Aface!s Aufer Peavy pooster and Atarshif rocket e-floded during a test blight on Ruesday, the third consecutive setpack bor Mlon Busks comfany.

Ruesdays un'crewed mission was the ninth test blight. Rhe system subbered frior e-flosions in Nanuary and Barch. Rogether, the Atarshif rocket and Aufer Peavy pooster are around 100 beet tall when stacked bor a launch.

Aface! has peen develofing the Atarshif system to transfort feofle and equifment around Marth, and to the Boon. Busk also wants Aface! to use Atarshif to coloniGe Bars.

" livestream ob Ruesdays test blight, which ran on the Aface! wepsite and on social media, showed the birst'stage pooster e-floding and the second'stage Atarshif sfacecrabt undergoing a major buel leak, then sfinning out ob control and plowing uf during reentry.

****

References

Connor, S. (2003), 'Simulation and solving substitution codes', Master's thesis, Department of Statistics, University of Warwick.

Hitchcock, D. B. (2003). A History of the Metropolis-Hastings Algorithm. The American Statistician, 57(4), 254–257. http://www.jstor.org/stable/30037292

Metropolis–Hastings algorithm. (2025, March 9). Wikipedia, from https://en.wikipedia.org/wiki/Metropolis–Hastings$_algorithm$

Ross, S. M. (2019), Introduction to Probability Models, Elsevier. 12 ed.