Optimization: MCMC and Simulated Annealing

Lucas Childs, Kai Barker

June 2025

1 Introduction

Many challenging combinatorial tasks can be reduced to optimizing an objective over a large discrete space of permutations. The Metropolis-Hastings algorithm, a method of Markov Chain Monte Carlo (MCMC), provides a powerful foundation for tackling such problems. In this project, we focus on applying MCMC techniques to two such tasks: message decryption and the traveling salesman problem (TSP). For message decryption, given an alphabet (English in our case) and its permutations as a state space, we can use MCMC to search for the permutation that most likely produces meaningful text. To do so effectively, we draw on key concepts such as likelihood functions, permutations, and fundamental theorems of Markov Chains. In the following sections, we discuss these concepts, describe the implementation of the MCMC algorithm, and propose a modification to the Metropolis-Hastings method to further optimize the algorithm. While the ultimate goal is to decrypt a message, the algorithm seeks to maximize the likelihood that a given permutation corresponds to readable text. Building on this foundation, we further explore the application of MCMC methods to the well-known optimization challenge of the traveling salesman problem. The TSP similarly searches over permutations, but in terms of a tour of cities rather than characters. Just as in the decryption problem, the objective in the TSP is to find a permutation of cities that optimizes a given metric such as distance, mileage, or time. The salesman must visit all cities exactly once before returning to the starting position. This new optimization problem highlights another use case of MCMC and provides insight into how we can improve the Metropolis-Hastings algorithm for both the TSP and message decryption. One technique used to solve the traveling salesman problem, simulated annealing, approximates the global optimum of a function by allowing the chain to transition with a higher acceptance probability based on an initial temperature T.

Simulated annealing approximates the global optimum of a function that may contain several local optima. It does this by starting a temperature parameter high, and incrementally lowering it as time goes on, or cooling. Higher temperature gives the chain freedom to transition to any new state, whether it increases the tour length or not, by raising the acceptance probability (determines whether a proposed new state is accepted or rejected). This phase is called exploration. When temperature is cooled, the chain more often accepts moves that decrease the tour length, since towards the end of the search we want the chain to narrow in on the best optimum. Thus, with a few small changes to the existing algorithm, we seek to avoid ending up in local optima by allowing more exploration of the state space which, in theory, will prevent the chain from getting stuck in states that are similar to the true decryption key but not accurate. Furthermore, we were able to empirically validate this, discovering that simulated annealing applied to message decryption generated responses that had on average almost 1 less misspelled letter (and therefore word) compared to standard Metropolis-Hastings.

In the sections that follow, we will further explain MCMC, the technical foundations of the decryption problem and TSP, introduce our proposed changes to the Metropolis-Hastings algorithm, and document and analyze the results it has on decrypting an encrypted message.

2 Traveling Salesman Problem

2.1 Permutations on Graphs

We can define an alphabet, Σ, "by letters and symbols used to write in a language" (lec8). "A language is made up of words formed from Σ." Permutations can be used to encrypt messages in the language on Σ, e.g., to swap certain letters for others in a deterministic set, so someone with the key can decrypt a seemingly meaningless expression.

In the context of the Traveling Salesman Problem (TSP), we consider a graph $G = (V, E)$ with $|V| = n$ nodes/vertices and edges $e \in E$ that have a known distance (from one node to another). The Salesman's goal is to visit each city (node) and return to its starting position in the shortest possible route. Any tour can be represented as a permutation of the city indices (some order of visiting each city). So, given city indices $\{A, B, C, D, E\}$, a permutation $\gamma = \{B, E, D, C, A\}$ represents one such tour with a total distance assigned to it. Thus the state space $\mathbb{S}$ is the set of all $n!$ permutations of n cities, which are too many to brute force for a large n. This is where MCMC comes into play.

2.2 The likelihood of tours

In terms of encryption/decryption, the likelihood function is the product of the letter-to-letter probabilities learned from the "War and Peace" training data. "War and Peace" was used to find out the probability of an "o" following an "n" ($n \to o$) for example. Then it took all letter-to-letter probabilities and multiplied them together.

What we do in the context of TSP is gather the training data from a complete weighted graph, which is the complete distance table $D = \{d_{ij}\}_{i \neq j}$ between every pair of cities, and convert distances into probabilities where

$$p_{ij} = \frac{\exp(-\beta d_{ij})}{\sum_{k \neq i} \exp(-\beta d_{ik})}, \quad i \neq j.$$

p_{ij} is the probability of an edge connecting city i to j. β represents inverse temperature ($\frac{1}{T}$), a large β means that only short edges get a meaningfully high probability and a small β means that all edges are almost equally likely to be seen. Then we initialized β on the smaller side to induce "exploration" in the algorithm. Similar to the exploration/exploitation problem in reinforcement learning (RL), we want the algorithm to find potential optima and explore the environment when starting the algorithm and gradually narrow in and converge more to known optima to find the best (global) optima. This technique is known as simulated annealing, in which our algorithm increases β over time [1]. For a tour (permutation) $\gamma = (v_1, .., v_n)$ where $v_{n+1} = v_1$, we define the likelihood function

$$\mathcal{L}(\gamma) = \prod_{k=1}^{n} p_{v_k v_{k+1}}$$

where the following is equivalent:

$$\log \mathcal{L}(\gamma) = \sum_{k=1}^{n} \log p_{v_k v_{k+1}}.$$

So, given some graph G and permutation γ, the likelihood that γ_G is the shortest tour is $\mathcal{L}(\gamma)$. We want to maximize the log likelihood which is the same as minimizing $\beta \sum d_{ij}$, the tour length, since $\log p_{ij} = -\beta d_{ij} - \log \sum_k e^{-\beta d_{ik}}$.

Then we construct a Markov chain with $\mathcal{L}$ and let the stationary distribution be $\pi(\gamma) = \frac{\mathcal{L}(\gamma)}{\sum_{z \in \mathbb{S}} \mathcal{L}(z)}$. We cannot directly evaluate π because the state space is so large, so normalization is infeasible. But, as the chain runs, its entropy/likelihood increases and gets closer to the stationary distribution π and thus closer to the permutations that find the shortest tour of graph G.

3 Markov Chain Monte Carlo

3.1 How it Works

In the traveling salesman problem, we seek the shortest tour (permutation) of the n cities. One such candidate is the tour γ of the cities $(v_1, ..., v_n) \in V$. Then our target distribution is

$$\pi(\gamma) = \frac{\mathcal{L}(\gamma)}{\sum_{z \in \mathbb{S}} \mathcal{L}(z)}.$$

But, once again, we cannot directly evaluate π because $\mathbb{S}$ is large for a large n. We construct a Markov chain on $\mathbb{S}$ using Metropolis-Hastings in which our stationary measure $\mu = \mathcal{L}$ (meaning μ is not normalized) should converge to π, the most optimal state, after a long time. We let define a proposal function:

$$q(\gamma, \gamma') = \begin{cases} \dfrac{1}{\binom{n}{2}}, & \text{if } \gamma' \text{ is obtained from } \gamma \text{ by swapping the cities in two distinct positions,} \\ 0, & \text{otherwise} \end{cases}$$

so that proposal function is symmetric: $q(\gamma, \gamma') = q(\gamma', \gamma)$. This symmetry results in $q(\gamma', \gamma)/q(\gamma, \gamma') = 1$. Then the Metropolis-Hastings acceptance rule is:

$$a(\gamma, \gamma') = \min\Big\{1, \frac{\mu(\gamma')q(\gamma', \gamma)}{\mu(\gamma)q(\gamma, \gamma')}\Big\} = \min\Big\{1, \exp\big[-\beta\big(\mathcal{L}(\gamma') - \mathcal{L}(\gamma)\big)\big]\Big\}.$$

Next, by [2], the transition probabilities or transition kernel is defined as

$$P_\beta(\gamma, \gamma') = \begin{cases} q(\gamma, \gamma')\, a_\beta(\gamma, \gamma'), & \gamma' \neq \gamma, \\ 1 \;-\; \displaystyle\sum_{z \neq \gamma} q(\gamma, z)\, a_\lambda(\gamma, z), & \gamma' = \gamma. \end{cases}$$

So, from state γ, some permutation, we propose a new permutation γ', and accept the transition to γ' with probability $a(\gamma, \gamma')$. This process is Markov chain monte carlo, since we are finding states that maximize $\mu(\gamma)$ to search for the most optimal tour of n cities. And, due to the Fundamental Limit Theorem [3], finite, irreducible, and aperiodic Markov chains guarantee eventual convergence to the chain's stationary distribution π such that

$$\sum_j \pi_j = 1, \quad \pi_j = \sum_i \pi_i P_{ij}.$$

Here, the state space is finite: $|\mathbb{S}| = n! < \infty$, irreducible because every state (permutation) communicates with one another since a sequence of swaps can transform any tour into any other, and aperiodic since each state has a "self loop", $P(\gamma, \gamma) > 0$. Thus, when we simulate this Markov chain for a long time (many iterations), states with the largest π are likely to be seen and the chain converges to this stationary distribution.

3.2 Proposed Modification

With such a large state space in the TSP, the likelihood will peak around the shortest tour, but will also contain plenty of similar permutations that may differ by only a few switched cities (local optima). To limit the number of transitions, we only allow a transition in the Metropolis-Hastings algorithm if the new permutation differs by the swap of the two cities' order, which means that we will progress towards the global optimum–the shortest tour–at a slow rate. Thus, we propose an update to the existing Metropolis-Hastings algorithm in order to approximate the permutation giving the shortest tour with less of a chance of ending up in a sub-optimal tour. We propose simulated annealing, which, in theory, should perform better than other 2-opt iterative-based improvement solutions in the context of the TSP [1]. We will adjust the algorithm's acceptance probability such that:

$$a(\gamma, \gamma') = \min\Big\{1, \exp\big[-\beta_k\big(\mathcal{L}(\gamma') - \mathcal{L}(\gamma)\big)\big]\Big\}$$

where β_k, inverse temperature, is continuously increased, representing a cooling of temperature. A large β means that mostly only short edges will be accepted in transitions and a small β means that all edges are almost equally likely to be accepted, inducing exploration of different tours. So, starting β low, or temperature high, at the beginning lets the chain find many possible permutations and explore different routes to visit cities. Then, as time goes on, temperature cools, freezing the structure of the tour more as the chain becomes biased towards visiting cities that decrease the tour length until it converges on that approximate shortest tour. However, the effectiveness of this technique depends on the choice of the cooling schedule, a method by which we decrease temperature. We used an exponential cooling schedule which decreases temperature by a fraction each iteration known as α. A larger α results in faster cooling, but might not allow for the algorithm to find the global optimum. A smaller α results in slower cooling, allowing for more exploration but can slow down convergence. Additionally, we specify the total number of steps to run the MCMC and the number of steps to run each block before cooling the system.

As simulated annealing is known to improve Metropolis-Hastings for the TSP, it can similarly be used to enhance the algorithm for message decryption following the same theory. We can begin our Markov chain at a high temperature so that acceptance probability is high for any proposed state, letting us explore many different permutations further from our starting point and essentially avoiding permutations that may appear similar to the decryption key, but are not optimal, trapping our original algorithm. As time goes on, the cooled temperature chain will more often accept moves that improve the likelihood the text is in English. In principle, this change will keep us closer to the true decryption key, yielding a more accurate decryption than in the more basic existing algorithm.

4 Results and Conclusions

After implementing our proposed modifications to the existing Metropolis-Hastings algorithm, we can compare the effects to running simulated annealing against baseline Metropolis-Hastings in the context of message decryption. In both, we use "War and Peace" as the "training data" and 'secret_message.txt' as the encrypted text to decrypt. Through many trial and error runs with the scheduled temperature parameter of simulated annealing, various different temperatures, cooling constants, and iterations, the impact was slightly better than the baseline performance. Simulated annealing thus behaved as expected, exploring permutations before gradually cooling off and narrowing down on the best option. However, in both cases of running the original code and the code with our updated acceptance probability, we encountered relatively similar deciphers, with certain words appearing to be spelled consistently wrong across methods, e.g. "Tuesday", "heavy", "Spacex", "March", and "Musk". The consistent mistakes across both methods could be due to the fact that these words do not occur frequently in "War and Peace" (we searched for key words within the book), so, the likelihood of this specific permutation of letters had a diminished probability after training. After running each method twice and averaging over each of the number of incorrect letters across all six guesses, here were the results: the average of 2 runs of baseline Metropolis-Hastings was 22.3 and the same computation for simulated annealing gave 21.6. Although results are very similar, simulated annealing demonstrated slightly better results than our baseline algorithm on average.

It is worth noting that we initialized each algorithm differently, running the baseline with its default parameter initializations (5000 iterations) but altering simulated annealing to run for 50,000 iterations. In addition, we set our default β to 0.3, the cooling factor α to 0.995 (cooling by 0.5% each block), and each block to 5000 iterations. When we ran the baseline for more iterations (50,000) it performed worse, likely because the chain had time to fully mix and converge to its stationary distribution, which—without a high inverse temperature parameter—still assigns noticeable probability to many suboptimal permutations. So, the algorithm is sampling from the correct stationary distribution, but that distribution does not concentrate tightly around the global optimum. On the contrary, simulated annealing improves with longer runtime. The longer we let it run, the more gradually it can cool, allowing it to broadly explore the permutations space and then slowly focus on increasingly optimal permutations as the temperature decreases. So, while a long run of fixed-temperature Metropolis-Hastings may continue bouncing around less optimal permutations due to its fixed level of randomness, simulated annealing gradually reduces that randomness by design.

5 Code

Listing 1: Original Code with Proposed Changes

```
from metropolis_hastings import *
import shutil
from deciphering_utils import *

#!/usr/bin/python

import sys
from optparse import OptionParser

```

```
10
11 def main(argv):
12     inputfile = None
13     decodefile = None
14     parser = OptionParser()
15
16     parser.add_option("-i", "--input", dest="inputfile",
17                       help="input file to train the code on")
18
19     parser.add_option("-d", "--decode", dest="decode",
20                       help="file that needs to be decoded")
21
22     # changed default iters to 15k
23     parser.add_option("-e", "--iters", dest="iterations", type="int
           ",
24                       help="total accepted moves to run", default
                           =15000)
25     ################### new code ###################
26     # this is inverse temperature, we can just choose a random
           default
27     # value for this after playing around with what seems to work
28     parser.add_option("-b", "--beta0", dest="beta0", type="float",
29                       help="intial inverse temp", default=0.3)
30
31     # this is the cooling factor
32     parser.add_option("-a", "--alpha", dest="alpha", type="float",
33                       help="cooling factor alpha (<1)", default
                           =0.995)
34
35     # this is how many moves are accepted before we cool the
           temperature
36     parser.add_option("-k", "--block", dest="block", type="int",
37                       help="accepted moves per temp level", default
                           =3000)
38     ################### end ###################
39
40     parser.add_option("-t", "--tolerance", dest="tolerance",
41                       help="percentate acceptance tolerance, before
                           we should stop", default=0.02)
42
43     parser.add_option("-p", "--print_every", dest="print_every",
44                       help="number of steps after which diagnostics
                           should be printed", default=10000)
45
46     (options, args) = parser.parse_args(argv)
47
48     filename = options.inputfile
49     decode = options.decode
50
51     if filename is None:
52         print("Input file is not specified. Type -h for help.")
53         sys.exit(2)
54
55     if decode is None:
56         print("Decoding file is not specified. Type -h for help.")
```

```
57      sys.exit(2)
58
59  char_to_ix, ix_to_char, tr, fr = compute_statistics(filename)
60
61  s = list(open(decode, 'r').read())
62  scrambled_text = list(s)
63
64  ################## new code here as well #################
65  # define the variables based on the inputs
66  beta = float(options.beta0)       # inverse temperature
67  alpha = float(options.alpha)      # cooling factor, a scalar (0,
        1)
68  budget = int(options.iterations) # total number of MCMC steps
69  block = int(options.block)        # number of steps between each
        temperature update
70
71  # start at our initial state with no entropies
72  initial_state = get_state(scrambled_text, tr, fr, char_to_ix)
73  state = initial_state
74  states = [state]
75  entropies = []
76
77
78  while budget > 0:
79      # run up to 'block' MCMC steps or budget if it's smaller
80      iters = min(block, budget)
81
82      # log probability of being in a state, multiplied by
            scheduled beta
83      logdens = lambda st: beta * compute_probability_of_state(st)
84
85      # run metropolis hastings algorithm with our logdens (which
            includes the beta)
86      state_run, lps_run, _ = metropolis_hastings(
87          state, proposal_function=propose_a_move, log_density=
                logdens,
88          iters=iters, print_every=int(options.print_every),
                tolerance=options.tolerance,
89          pretty_state=pretty_state
90      )
91
92      state = state_run[-1]
93      states.extend(state_run[1:])
94      entropies.extend(lps_run)
95      budget -= iters
96
97      # cool the system (decrease temperature --> increase beta)
98      beta /= alpha
99      print(f"[SA] Beta updated to {beta:.4f}")
100
101 ################### end code ###################
102
103 p = list(zip(states, entropies))
104 p.sort(key=lambda x:x[1])
105
```

```
    print("␣Best␣Guesses␣:␣\n")

    for j in range(1,4):
        print(f"Guess␣{j}:␣\n")
        print(pretty_state(p[-j][0], full=True))
        print(shutil.get_terminal_size().columns*'*')

if __name__ == "__main__":
    main(sys.argv)
```

References

[1] Bertsimas, Dimitris, and John Tsitsiklis. 'Simulated annealing.' Statistical science 8.1 (1993): 10-15. https://projecteuclid.org/journals/statistical-science/volume-8/issue-1/Simulated-Annealing/10.1214/ss/1177011077.full

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

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