Lydia LaSeur
MCMC – Final Project
December 4, 2015

Final Project: Substitution Cipher

Problem:

One of the simplest ways to encrypt an alphabetical (or alpha-numeric) message is with a unique one to one substitution cipher. Such a cipher would map each character to another character in the set. For example, if the character set is {A, B, C}, one possible mapping is {A, B, C} ⇒ {C, A, B}, i.e. A to C, B to A, and C to B.

If given an encrypted message and it's known that a one to one substitution was used for the encryption, checking all possible mappings to find the unique cipher, while exhaustive, is inefficient to the point of being virtually impossible. If considering the entire 26 character English alphabet, the possible number of mappings is 26!, which is greater than 10^{26}. This is such a large solution space, or state space, that an average modern PC would take billions of years to search all possible mappings.

Consider the following message encrypted with a 1 to 1 cipher that maps the 26 characters of the alphabet, where all punctuation present in the original message was ignored, spaces between words simply map to themselves, and the there is a space between each line.

BIU V DICT ZMEF VBZFU MZ NVJ OJGFUFL MCZI ZGMJ NIUDL IB JIUUIN VCL ZUIOQDF QK ZGF HVUMJG JOUTFIC MZ UFEVMCFL V EVZZFU IB PICJMLFUVQDF LIOQZ NGFZGFU ZGF PGMDL NIODL JOURMRF ZI QFVU VCK CVEF VZ VDD MC NGMPG PVJF MZ MJ JIEFNGVZ EIUF ZGVC HUIQVQDF ZGVZ ZGFJF EFEIMUJ NIODL CFRFU GVRF VHHFVUFL IU MB ZGFK GVL ZGVZ QFMCT PIEHUMJFL NMZGMC V PIOHDF IB HVTFJ ZGFK NIODL GVRF HIJJFJJFL ZGF MCFJZMEVQDF EFUMZ IB QFMCT ZGF EIJZ PICPMJF VCL BVMZGBOD JHFPMEFC IB QMITUVHGK FYZVCZ MC ZGF DMZFUVZOUF IB VCK VTF IU PIOCZUK

Search and apply possible mappings until the message is at least readable by a human.

Method:

The exploration of the solution space can be intelligently guided if knowledge of existing examples of English text is provided to the method. For this problem, the provided knowledge is the frequencies of first order transitions from one particular letter to another letter, itself, or a space e.g. how often A immediately follows D, or A immediately follows A, or A immediately follows a space.

The only way to find all 27^2 frequencies exactly is to analyze every instance of correctly spelled English text, so estimated frequencies will have to suffice. For this problem, *War and Peace, Oliver Twist,* and the King James Bible were analyzed and all first order transitions recorded, ignoring all punctuation. The frequencies are stored in a 27x27 matrix M, where M_{ij} is the number of times character j

immediately followed the character *i,* and *i* can equal *j*. The elements of the matrix can be left as counted integer values for each transition, or they can be normalized as probabilities by dividing each element by the total number of transitions recorded.

With these approximate frequencies the likelihood that a particular mapping, *f*, will be successful in decoding the message can be defined by the following distribution function, where s_i is the i^{th} character in the encrypted message, *N* is the total number of characters in the message, and *M* is the matrix of transition frequencies. Since the transition frequencies are approximate, the likelihood will also be an approximation.

$$PI(f) = \prod_{i=1}^{N-1} M(f(s_i), f(s_{i+1}))$$

The encrypted message for this problem has a total of 505 characters, requiring 504 multiplications to obtain the likelihood. If *M* contains the counted integer values, overflow is likely, and if *M* contains probability values, underflow is likely. To avoid either of these problems, *log PI(f)* will be calculated instead of *PI(f)*.

$$logPI(f) = \sum_{i=1}^{N-1} log(M(f(s_i), f(s_{i+1})))$$

Note that if just one of the 504 total transitions in the message has a frequency of zero, the likelihood goes to zero, and thus the log of the likelihood goes to -∞.

Now that a probability distribution has been defined, a Markov Chain Monte Carlo method can use said distribution to explore the state space intelligently and efficiently. For this problem, the basic structure of the method used is based on Metropolis MCMC. Each MC step will start with current mapping *f* and consist of a proposal to a new mapping *f'* followed by a comparison of *log PI(f)* and *log PI(f')* to determine if the proposed mapping is accepted. This is repeated for a set number of MC steps or until the message is readable. The initial mapping is randomly generated. If the chain fails to decode the message in a certain amount of steps/time, run the method again with a different initial guess.

To propose a new mapping *f'*, two letters, *p* and *q* (only letters, cannot be 'SPACE'), are randomly selected and their positions in the current mapping *f* are swapped. The new mapping is applied to the original encrypted message and *log PI(f')* is evaluated. The acceptance function will be discussed in detail in the next section.

<u>Sampling vs. Optimization:</u>

Metropolis MCMC can be used in two different ways to explore the state space and find the best mapping. First, one could use it to draw samples from the distribution *PI(f)*, keep track of the most likely mapping (or mappings) and stop once the message is readable. Second, one could assume that the best mapping (or one that at least makes the message readable) is the state that marks the global maximum of the likelihood *PI(f)*, and use MCMC as an optimization technique to find that mapping/maximum.

The determining factor of whether the method will act as a sampling technique or an optimization technique is the choice of acceptance criteria. For sampling, the acceptance function should accept all proposals that increase the likelihood *PI(f)* and decide whether to accept or reject proposals that decrease the likelihood, in such a manner that a proposal with a relatively large decrease in likelihood, $\Delta PI = |PI(f') - PI(f)|$ where $PI(f') < PI(f)$, is less likely to be accepted than a proposal with a relatively small decrease. This allows the chain to spend the most time exploring states with a higher likelihood, while still spending an appropriate amount of time exploring states with lower likelihoods, hopefully resulting in a set of samples that accurately represents the distribution.

When sampling with Metropolis MCMC, the ratio $PI(f')/PI(f)$ describes the change in likelihood and acts as the probability of acceptance. If $PI(f') > PI(f)$ the ratio is greater than or equal to 1, thus the probability of acceptance is 1 and the proposal is always accepted. If $PI(f') < PI(f)$ the ratio is less than 1, thus the probability of acceptance is less than 1. A random uniform number between 0 and 1 is drawn, if it's less than the ratio the move is accepted, otherwise it is rejected.

The acceptance function is simpler if optimization is the goal. The acceptance criteria is based on the difference between the proposed and current likelihoods, $\Delta PI = PI(f') - PI(f)$, instead of the ratio between the two likelihoods. If ΔPI is positive, the proposal is always accepted. Conversely, if ΔPI is negative the proposal is always rejected. Thus, the probability of acceptance can only be 0 or 1. This choice of criteria dictates that the chain will only explore states that continuously increase the likelihood and will never "go back" to explore any states that would decrease the likelihood.

Since the goal of the problem is to find the single "best" mapping, there is no need to save the set of samples, or mappings, drawn from *PI(f)* or even be concerned if the set is accurately described by *PI(f)*. One only needs to keep track of the best mapping found so far. Therefore, optimization seems to be the smarter approach. However, optimization will only be successful if the best mapping actually corresponds to the location of either the global maximum or maybe a local maximum. No aspect of this problem can guarantee that those two states are equal or even similar. The benefit of sampling is a much more comprehensive search of the state space; the drawback is that the work put towards drawing a "good" set of samples is essentially wasted. Ultimately, optimization was chosen as the strategy to tackle this specific problem.

Code Implementation:

Coding the method for optimization is relatively simple and easy. This section will examine the functions written for the program and how the mapping and the message are stored and manipulated.

Mapping: A mapping *f* is a function where a specific letter, *x*, is the input and the output, *y,* is the letter that is mapped to input *x,* i.e. *f(x) = y.* A simple way to implement this function in the code is as a 1-D array with 26 elements, where the index of an element is the input letter, *x*, and the value of the element at a specific index, *x,* is the output letter, *y*. Of course, the letters {A, B, C, …, Z} are represented with the integers {0, 1, 2, …, 25} accordingly. The integer values start at zero since the indexing of arrays in Python always starts at zero. For example, the mapping for the letter "D" is the letter/value at the 4th element in the map array, or f[3].

Message: The message is also stored as a 1-D array with 505 elements, one for each character in the message. Again each letter is represented by the integers {0, 1, 2, …, 25} and 'SPACE' is represented with the integer 26. To convert the message to this integer format, each character is converted to its ASCII integer value. The ASCII values for uppercase letters range from 'A' = 65, …, 'Z' = 90, and the ASCII value for space is 32. After converting all characters to ASCII integers, subtract 65 so 'A' = 0, …, 'Z' = 25. Afterwards replace all negative values with 26 so 'SPACE' = 26. Once the message is converted to integers, it is not converted back to alphabetical characters until the program has completed the optimization. Code shown below.

```
#import the initial message from file as an array of characters
msg_file = 'coded_msg.dat'
infile = open(msg_file,'r')
msg_alpha = np.loadtxt(infile,dtype = 'c',delimiter = False)
infile.close()

#convert characters to integers
msg_int = np.copy(msg_alpha.view(np.int8)) - 65
space_loc = np.nonzero(msg_int < 0)
msg_int[space_loc] = 26
```

function - `applyMap(f,msg,map_size)`

Given a mapping *f* and a message, this function will apply the mapping to each character in the message. This function is used only when searching for a good initial, random mapping. In order to calculate the likelihood of an initial mapping, it must be applied to the original message and the likelihood is calculated based on the transition pairs found in the updated message. Instead of looping through each character in the message, the function loops through all possible letters, finds all locations of the current letter, and replaces them with the mapped equivalent. Code shown below.

```
def applyMap(f,msg,map_size):

    new_msg = np.copy(msg)

    #for each of the 26 letters
    for i in range(map_size):
```

```
      #find the location of current letter in the message
      letter_loc = np.nonzero(msg == i)[0]
      #replace the current letter i at those locations according to map
f
      new_msg[letter_loc] = f[i]

    return new_msg
```

function – `logPI(msg,M,N)`
This function calculates the log of the likelihood *PI(f)* of a specific mapping *f*. The input *msg* is the encrypted message after the mapping *f* has been applied, *M* is the transition frequency matrix, and *N* is the total number of characters in the message. Instead of using two nested for loops to get all 504 transitions, the function uses an array of pair wise matrix indices to get pull all 504 frequency values from *M*. To avoid taking the log of zero, if at least one of the 504 frequencies is zero, log_PI is set to -∞ and immediately returned. If all frequencies are non-zero, take the log of each one and sum to get log_PI. Code shown below.

```
def logPI(msg,M,N):
    #find the beginning of each transition
    first = np.copy(msg[0:N-2])
    #find the end of each transition
    second = np.copy(msg[1:N-1])

    #get the frequency of all transitions
    pair_freq = M[first,second]


    #if any of the frequencies are zero, the PI = 0, and logPI = -inf
    if np.sum(pair_freq == 0) != 0:
      log_PI = float('-inf')

    #else, take the log of each frequency and sum for the likelihood PI
    else:
      log_pair_freq = np.log(pair_freq)
      log_PI = np.sum(log_pair_freq)

    return log_PI
```

function - `letterSwap(letter_i,letter_j,f,msg)`
This is the proposal function. Earlier, the proposal was described as swapping two different, randomly selected letters, *p* and *q*, in the current mapping *f* to obtain the new mapping *f'*, then the new mapping is applied to the original encrypted message to implement the swap in the previous update of the message for the next update of the message. However, it is computationally cheaper to simply swap *p* and *q* in the most recent update and then update the mapping accordingly. Again, for loops were avoided with the use of arrays of indices. Code shown below.

```
def letterSwap(letter_p,letter_q,f,msg):
```

```
    #swap for message
    #find the locations of p and q in the most recent msg
    p_loc = np.nonzero(msg == letter_p)[0]
    q_loc = np.nonzero(msg == letter_q)[0]

    new_msg = np.copy(msg)
    #at the p locations, assign letter q
    new_msg[p_loc] = letter_q
    #at the q locations, assign letter p
    new_msg[q_loc] = letter_p


    #swap for mapping
    #find the index where f = p, and the index where f = q
    p_idx = np.nonzero(f == letter_p)[0][0]
    q_idx = np.nonzero(f == letter_q)[0][0]

    new_f = np.copy(f)
    #assign q to p's index
    new_f[p_idx] = letter_q
    #assign p to q's index
    new_f[q_idx] = letter_p

      return new_msg, new_f
```

function - `acceptance(old_PI,new_PI)`

Obviously this is the acceptance function. Input the log of the likelihood for both the current and new states, accept the move if the log of the new likelihood, new_PI, is greater than the log of the old likelihood. In this program *M* was normalized to probability values, so all elements in *M* are contained in [0,1) and therefore the log of those frequencies, as well as the log of the likelihood, are contained in $(-\infty, 0)$. With so many additions required to get log_PI, the magnitude of log_PI is typically of the order -10^3, which is guaranteed to lead to underflow if one took the exponential of log_PI in an attempt to find PI. The logs can simply be compared since the only concern is if the likelihood increased or decreased, and not how much it increased or decreased. Code shown below.

```
def acceptance(old_PI,new_PI):

    #if new log_PI is -inf, new PI is zero, new map is rejected
    if new_PI == float('-inf'):
      acc = False
      return acc

    #if new logPI is greater than old logPI, the new PI is higher and
the move is accepted
    if new_PI > old_PI:
      acc = True
    else:
      acc = False
```

function - `decodeDriver(MCS,f,msg,M,N)`
The driver function has the standard structure of Metropolis MCMC: a for loop to complete the desired number of MC steps, each step begins with a current mapping, message, and likelihood, two letters are randomly selected for the swap, after the swap is completed the new likelihood is calculated, and a comparison of the current and new likelihoods determines if the map is accepted or rejected after the swap. If accepted, the new mapping becomes the current mapping, if rejected, the current mapping remains the same. Continue until all MC steps are completed. One could monitor the message as it changes according the new mappings and stop the chain once the message is readable, but the program runs fast enough that no significant time is wasted when its run for a set number of steps. Code shown below.

```
def decodeDriver(MCS,f,msg,M,N):

    num_succ = 0

    #initial mapping, message, and likelihood
    old_f = f
    old_msg = np.copy(msg)
    old_PI = logPI(old_msg,M,N)

    for i in range(MCS):

      #randomly pick two letters to swap
      swapped = np.random.randint(26,size = (2,1))
      #apply the swap to the message/map
      new_msg, new_f = letterSwap(swapped[0],swapped[1],old_f,old_msg)
      #get the new likelihood
      new_PI = logPI(new_msg,M,N)
      #check for acceptance
      accept = acceptance(old_PI,new_PI)

      if accept:
            num_succ += 1
            #new becomes current/old
            old_f = new_f
            old_msg = new_msg
            old_PI = new_PI


    acc_ratio = float(num_succ)/float(MCS)
    final_msg = old_msg
    final_map = old_f

      return final_msg,final_map,acc_ratio
```

function - `randMap(map_size,msg,M,N)`
This function generates the initial guess for a mapping. Instead of accepting any random mapping, the function works to find a random mapping that has a non-zero *PI(f)*. Using a while loop with a contained if-statement, the function generates and checks random mappings until one with non-zero likelihood is found. This function the limiting factor on performance since this process is much slower than the actual

optimization. However, the time required is worth it since a good starting point allows the method to successfully and quickly find the optima. (The importance of the initial mapping will be discussed further after the results.) Code shown below.

```
def randMap(map_size,msg,M,N):

     print 'Begin search for valid initial mapping...'
     #set valid map flag to false
     valid = False

     #while a valid initial mapping is not found
     while not valid:
       #generate a random mapping
       map = np.random.permutation(map_size)
       #apply mapping to the message
       new_msg = applyMap(map,msg,map_size)
       #get likelihood of mapping
       log_PI = logPI(new_msg,M,N)


       #if the log of the likelihood is not -inf
       if log_PI != float('-inf'):
            #the map has a non-zero likelihood PI, and is considered a
valid starting point
            valid = True
            print 'Valid map found.'

       return map, new_msg
```

Results:

Below is the printed output for three runs of the decoder.

Run #1: fully decoded

```
wc-dhcp169d000:MCMCFinal Lydia$ python finalProj.py
Message Decoder now running...


Initial encrypted message
Message length is 505 characters (including spaces)

BIU V DICT ZMEF VBZFU MZ NVJ OJGFUFL MCZI ZGMJ NIUDL IB JIUUIN VCL
ZUIOQDF QK ZGF HVUMJG JOUTFIC MZ UFEVMCFL V EVZZFU IB PICJMLFUVQDF
LIOQZ NGFZGFU ZGF PGMDL NIODL JOURMRF ZI QFVU VCK CVEF VZ VDD MC NGMPG
PVJF MZ MJ JIEFNGVZ EIUF ZGVC HUIQVQDF ZGVZ ZGFJF EFEIMUJ NIODL CFRFU
GVRF VHHFVUFL IU MB ZGFK GVL ZGVZ QFMCT PIEHUMJFL NMZGMC V PIOHDF IB
HVTFJ ZGFK NIODL GVRF HIJJFJJFL ZGF MCFJZMEVQDF EFUMZ IB QFMCT ZGF EIJZ
PICPMJF VCL BVMZGBOD JHFPMEFC IB QMITUVHGK FYZVCZ MC ZGF DMZFUVZOUF IB
VCK VTF IU PIOCZUK


Begin search for valid initial mapping...
```

```
Valid map found.
Map size is 26 characters long
Map generated in 4.895330 seconds
Initial mapping:
[14 19 15 18 21  7 20  6  0  4  2  1 13 11 24 12 25  8 16  5 10  3  9
23 17 22]


Ready to start MCMC for optimization...
Number of MC steps is 5000
All MC steps completed in 0.660510 seconds.
Acceptance ratio = 0.018400


Best mapping found
[ 9  5 13 11 12  4  7 15 14 18 24  3  8 22 20  2  1 21 16  6 17  0 25
10 23 19]

Apply the best mapping found to initial encrypted message...
Decoded message:
FOR A LONG TIME AFTER IT WAS USHERED INTO THIS WORLD OF SORROW AND
TROUBLE BY THE PARISH SURGEON IT REMAINED A MATTER OF CONSIDERABLE
DOUBT WHETHER THE CHILD WOULD SURVIVE TO BEAR ANY NAME AT ALL IN WHICH
CASE IT IS SOMEWHAT MORE THAN PROBABLE THAT THESE MEMOIRS WOULD NEVER
HAVE APPEARED OR IF THEY HAD THAT BEING COMPRISED WITHIN A COUPLE OF
PAGES THEY WOULD HAVE POSSESSED THE INESTIMABLE MERIT OF BEING THE MOST
CONCISE AND FAITHFUL SPECIMEN OF BIOGRAPHY EXTANT IN THE LITERATURE OF
ANY AGE OR COUNTRY


---Program fully executed---
```

Run #2: fully decoded

```
wc-dhcp169d000:MCMCFinal Lydia$ python finalProj.py
Message Decoder now running...

Initial encrypted message
Message length is 505 characters (including spaces)
BIU V DICT ZMEF VBZFU MZ NVJ OJGFUFL MCZI ZGMJ NIUDL IB JIUUIN VCL
ZUIOQDF QK ZGF HVUMJG JOUTFIC MZ UFEVMCFL V EVZZFU IB PICJMLFUVQDF
LIOQZ NGFZGFU ZGF PGMDL NIODL JOURMRF ZI QFVU VCK CVEF VZ VDD MC NGMPG
PVJF MZ MJ JIEFNGVZ EIUF ZGVC HUIQVQDF ZGVZ ZGFJF EFEIMUJ NIODL CFRFU
GVRF VHHFVUFL IU MB ZGFK GVL ZGVZ QFMCT PIEHUMJFL NMZGMC V PIOHDF IB
HVTFJ ZGFK NIODL GVRF HIJJFJJFL ZGF MCFJZMEVQDF EFUMZ IB QFMCT ZGF EIJZ
PICPMJF VCL BVMZGBOD JHFPMEFC IB QMITUVHGK FYZVCZ MC ZGF DMZFUVZOUF IB
VCK VTF IU PIOCZUK


Begin search for valid initial mapping...
Valid map found.
Map size is 26 characters long
Map generated in 34.209148 seconds
```

```
Initial mapping:
[11 15 18  1 13 14  4  7  3  2 25  8  0  5 22 21 17 20 10  6 12 19  9
16 23 24]

Ready to start MCMC for optimization...
Number of MC steps is 5000
All MC steps completed in 0.661271 seconds.
Acceptance ratio = 0.012000


Best mapping found
[25  5 13 11 12  4  7 15 14 18 24  3  8 22 20  2  1 21 10  6 17  0  9
16 23 19]

Apply the best mapping found to initial encrypted message...
Decoded message:
FOR A LONG TIME AFTER IT WAS USHERED INTO THIS WORLD OF SORROW AND
TROUBLE BY THE PARISH SURGEON IT REMAINED A MATTER OF CONSIDERABLE
DOUBT WHETHER THE CHILD WOULD SURVIVE TO BEAR ANY NAME AT ALL IN WHICH
CASE IT IS SOMEWHAT MORE THAN PROBABLE THAT THESE MEMOIRS WOULD NEVER
HAVE APPEARED OR IF THEY HAD THAT BEING COMPRISED WITHIN A COUPLE OF
PAGES THEY WOULD HAVE POSSESSED THE INESTIMABLE MERIT OF BEING THE MOST
CONCISE AND FAITHFUL SPECIMEN OF BIOGRAPHY EXTANT IN THE LITERATURE OF
ANY AGE OR COUNTRY


---Program fully executed---
```

Run #3: partially decoded

```
wc-dhcp169d000:MCMCFinal Lydia$ python finalProj.py
Message Decoder now running...

Initial encrypted message
Message length is 505 characters (including spaces)
BIU V DICT ZMEF VBZFU MZ NVJ OJGFUFL MCZI ZGMJ NIUDL IB JIUUIN VCL
ZUIOQDF QK ZGF HVUMJG JOUTFIC MZ UFEVMCFL V EVZZFU IB PICJMLFUVQDF
LIOQZ NGFZGFU ZGF PGMDL NIODL JOURMRF ZI QFVU VCK CVEF VZ VDD MC NGMPG
PVJF MZ MJ JIEFNGVZ EIUF ZGVC HUIQVQDF ZGVZ ZGFJF EFEIMUJ NIODL CFRFU
GVRF VHHFVUFL IU MB ZGFK GVL ZGVZ QFMCT PIEHUMJFL NMZGMC V PIOHDF IB
HVTFJ ZGFK NIODL GVRF HIJJFJJFL ZGF MCFJZMEVQDF EFUMZ IB QFMCT ZGF EIJZ
PICPMJF VCL BVMZGBOD JHFPMEFC IB QMITUVHGK FYZVCZ MC ZGF DMZFUVZOUF IB
VCK VTF IU PIOCZUK


Begin search for valid initial mapping...
Valid map found.
Map size is 26 characters long
Map generated in 5.039200 seconds
Initial mapping:
[ 9  5  2 14 12 24  6 17  8 18  0 13 20 21  7  3 25  1 11 22 15  4 23
16 19 10]
```

```
Ready to start MCMC for optimization...
Number of MC steps is 5000
All MC steps completed in 0.603045 seconds.
Acceptance ratio = 0.012800


Best mapping found
[ 9  5 13 15  1  4  7 11 14 18 24  3  8 22 20  2 12 21 25  6 17  0 10
16 23 19]

Apply the best mapping found to initial encrypted message...
Decoded message:
FOR A PONG TIBE AFTER IT WAS USHERED INTO THIS WORPD OF SORROW AND
TROUMPE MY THE LARISH SURGEON IT REBAINED A BATTER OF CONSIDERAMPE
DOUMT WHETHER THE CHIPD WOUPD SURVIVE TO MEAR ANY NABE AT APP IN WHICH
CASE IT IS SOBEWHAT BORE THAN LROMAMPE THAT THESE BEBOIRS WOUPD NEVER
HAVE ALLEARED OR IF THEY HAD THAT MEING COBLRISED WITHIN A COULPE OF
LAGES THEY WOUPD HAVE LOSSESSED THE INESTIBAMPE BERIT OF MEING THE BOST
CONCISE AND FAITHFUP SLECIBEN OF MIOGRALHY EXTANT IN THE PITERATURE OF
ANY AGE OR COUNTRY


---Program fully executed---
```

Discussion:

From runs #1 and #2, it is obvious to see that optimization using MCMC is a successful and quick method for decoding this particular message with the help of the particular transition matrix *M* provided. However, there are reasons why this approach may not work for any encoded message given any transition matrix *M*. First, the solution to this specific problem will be discussed, followed by reasons why optimization worked for this problem, and examples of instances when optimization would fail.

While it's not immediately apparent to the human eye, only 22 of all 26 possible letters appear in the original encrypted message. These 4 missing letters include A, S, W, and X. Since the cipher is a one to one mapping, the absence of 4 letters in the encrypted message indicates that 4 letters will also be absent from the decoded message. From the above results one can observe that the 4 letters absent from the decoded message are J, K, Q, and Z. This is no surprise for such a short piece of text since 3 of those 4 letters (J, Q, and Z) are the 3 least common letters in the English language (a fact that anyone who's ever played a couple games of Scrabble can attest to). These absent letters almost act as degrees of freedom in our solution. In fact, there are 4! = 24 different possible mappings that will perfectly decode the original encoded message. As long as *f* has the correct mapping for all 22 other indices, the values at indices 0, 18, 22, and 23 (corresponding to the missing letters in the encoded message A, S, W, and X) can correspond to any possible mapping of the values 9, 10, 16, and 25 (corresponding to the missing letters in the decoded message J, K, Q, and Z). If one noticed that 4 letters were missing the encoded message, they could cleverly assume the missing letters would map to very

uncommon letters and craft the initial mapping with this knowledge instead of just using a random permutation.

In fact, the initial mappings for all three runs feature at least two instances of a missing letter in the encoded message mapped to a missing letter in the coded message.

Run #1 initial *f*:

```
[14 19 15 18 21  7 20  6  0  4  2  1 13 11 24 12 25  8 16  5 10  3  9 23 17 22]
```

f[18] = 16 i.e. 18 ⇒ 16 -OR- S ⇒ Q
f[22] = 9 i.e. 22 ⇒ 9 -OR- W ⇒ J

Run#2 initial *f*:

```
[11 15 18  1 13 14  4  7  3  2 25  8  0  5 22 21 17 20 10  6 12 19  9 16 23 24]
```

f[18] = 10 i.e. 18 ⇒ 10 -OR- S ⇒ K
f[22] = 9 i.e. 22 ⇒ 9 -OR- W ⇒ J
f[23] = 16 i.e. 23 ⇒ 16 -OR- X ⇒ Q

Run#3 initial *f*:

```
[ 9  5  2 14 12 24  6 17  8 18  0 13 20 21  7  3 25  1 11 22 15  4 23 16 19 10]
```

f[0] = 9 i.e. 0 ⇒ 9 -OR- A ⇒ J
f[23] = 16 i.e. 23 ⇒ 16 -OR- X ⇒ Q

Since run #3 was not as successful as #1 and #2, it is clear that a good initial mapping is not the only requirement for success. The chain for each run was only 5000 MC steps long, which is relatively short. The low acceptance ratios from all three runs reveal that only 60 to 90 of the 5000 proposed swaps were accepted. Since the swaps are determined randomly, a run could perform poorly if it is "unlucky" in a sense, where the few accepted swaps are simply not enough to get to the desired mapping in the time, or steps, allotted. However, this can be remedied if the method is allowed more time.

The method used here could be referred to as simulated quenching. SQ is guided by concepts similar to simulated annealing. Both methods begin by freely exploring the state space at a high temperature, SA decreases the temperature very slowly over many steps while SQ rapidly drops the temperature from very high to very low in a single step. SA allows for more thorough exploration of the entire space, which is why it works well for *global* optimization. SQ is good at searching for a neighborhood with a local optima and then approaching said local optima quickly. As stated earlier, for optimization to be successful the location of the optima *must* correspond to the best mapping. For this particular problem there 24 "best" mappings, so if a local maximum exists at or very near each of those 24 mappings in the state space, the message can be decoded by finding a mapping that corresponds to just one of those 24 local maximums. I suspect that since the

likelihood for most mappings will be equal to 0, selecting/finding an initial mapping with a nonzero likelihood starts the MCMC chain in the neighborhood of a local optima and given the right swaps, a successful mapping can be found quickly.

There are several cases where the optimization approach would certainly fail. The transition frequencies provided were estimated using classic literature, including *Oliver Twist*. It just so happens that the true decoded message is actually an excerpt from *Oliver Twist*, so it's really no surprise that the estimated transition frequencies provided were a good guide for decoding the excerpt. However, if the true decoded message was from an excerpt of contemporary literature that contained more modern language or from a scientific journal that contained a lot of scientific or mathematical jargon, the same M used for this problem will likely be a terrible guide for finding the most likely mapping. The transition frequencies in M are like the training data and the transitions of the encrypted message are the test data. If the matrix M is not "trained" properly with relevant instances of text/literature, it will perform poorly when testing.