Simulation and Solving Substitution Codes

Stephen Connor

Abstract

In his Hardy Lecture at Warwick University in May 2001, Persi Diaconis described some work he had undertaken for the Californian Prison Service. This entailed applying the theory of Markov Chain Monte Carlo (MCMC) simulation to the problem of decoding substitution codes written between prisoners. In this project I have attempted to tackle this issue myself. In it, I give an account of MCMC algorithms and their application to decoding, producing simulation programs of my own. I then introduce the theory of random walks on groups, and use this to estimate bounds on the convergence rate of such algorithms.

1 Introduction

In this section, the principle of Markov Chain Monte Carlo simulation is introduced, with a brief insight into its history and applications to date. Some consideration is also given to the issues involved in using such techniques in the context of decoding.

1.1 Monte Carlo Techniques

Statistical models have, over the years, become more and more complex, partly as a result of the increasing amount of data available. This increasing complexity has made it much harder to compute characteristics of associated probability distributions. For example, suppose we wish to calculate the expected value $\mathbb{E}[\psi(Y)]$ of a random vector Y in k dimensions with density function $f(x)$, where $\psi : \mathbb{R}^k \to \mathbb{R}$ and $\mathbb{E}[|\psi(Y)|] < \infty$. The usual method of course is to analytically calculate

$$\mathbb{E}[\psi(Y)] = \int_{\mathbb{R}^k} \psi(x) f(x) \mathrm{d}x \ ,$$

though in high dimensions this cannot often be accomplished. One alternative is to turn to numerical integration, but another is to use Monte Carlo estimation. This involves generating a sequence of i.i.d. random vectors $\{Y_i\}_{i \geq 1}$ with the same distribution as Y. The strong law of large numbers tells us that

$$\mathbb{E}[\psi(Y)] = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} \psi(Y_i)$$

and so we proceed to estimate the value in question by $\frac{1}{n}\sum_{i=1}^{n}\psi(Y_i)$. The accuracy of the estimate is clearly dependent upon the sample size, and the theory of confidence intervals tells us how large to choose n to satisfy a given level of accuracy. The Monte Carlo method so described is known to compete well with numerical integration and, in some circumstances, is the only method available.

The drawback, however, is that Monte Carlo methods require us to sample from high dimensional distributions, and this may be extremely difficult or expensive in computer and analysis time. In such cases, there is another method available to us: Markov Chain Monte Carlo. One of the main areas in which this has proved extremely useful is that of computing normalizing factors for probability distributions. Suppose we have some unknown variables $x \in X$ and data $y \in Y$, then to obtain the posterior $p(y|x)$ given the prior $p(x)$ and likelihood $p(y|x)$, we can use Bayes' theorem:

$$p(x|y) = \frac{p(y|x)p(x)}{\int_X p(y|x')p(x')\mathrm{d}x'}.$$

In high dimensions though, it can prove extremely difficult to calculate the integral (normalizing factor) above. Using MCMC, we can still sample from the distribution of interest, π, by constructing an aperiodic and irreducible Markov chain which has π as a limiting distribution. We can then sample from the chain when it is close to equilibrium, and treat this as a dependent sample from π.

Of course, this assumes that we can easily construct a suitable Markov Chain with limiting distribution π. It turns out that it is in fact not only relatively simple to do this, but that there are many suitable schemes available to us. In this project I will concentrate upon the *Metropolis-Hastings algorithm* (Section 2.1). This was introduced in the 1950s, and was originally motivated by the desire to solve problems in combinatorics and physics. It has been used extensively since, and, two years ago, was placed in a survey among the top ten algorithms that had the greatest influence on the development and practice of science and engineering in the 20th century [3]. Its use in decoding (a new application) is effective, but unfortunately limited as we shall see.

1.2 Modelling Language

The original application of Markov chains was to analyze the Russian poem *Evgeny Onedin*. Markov analyzed the alternation of consonants and vowels, which he modelled as a simple two-state Markov chain [24]. This, of course, means assuming that the probability of observing a vowel/consonant depends solely upon the last letter observed. Although this simplified approach to modelling language may seem a bit crude at first, the use of Markov chains in this area is widely regarded as acceptable.

We do not need to restrict ourselves to a two-state chain however. A simple generalization of Markov's work is to consider transition probabilities between an alphabet of letters. Further, Shannon and Weaver [26] discuss the modelling of language at a range of different levels: first using N-grams of letters

(sequences of length N), and progressing to the level of considering transition probabilities between sequences of words. At each increase in the amount of information being used, one sees a vast improvement in the language produced. The downside, however, is that the computational effort involved increases exponentially at the same time: if our alphabet is of size n, then a model using N-grams involves $n^{2(N-1)}$ transition probabilities. This clearly makes high-order models impractical.

This discussion does, however, give us some reason to consider using Markov chains to decipher substitution codes. By a substitution code I formally mean the following.

Definition 1.1 (Substitution Code). Suppose *Alph* is a character set of size n, and that we have a portion of text $\widetilde{T}$ (expressed as a vector of characters), that we wish to encode, made up entirely of characters in *Alph*. Suppose that *Alph'* is another character set, also of size n. If we encode $\widetilde{T}$ by choosing some bijection $f : Alph \to Alph'$ and applying this map to $\widetilde{T}$, then $T = f(\widetilde{T})$ is a substitution code.

This is obviously one of the simplest forms of code possible. It is also worth pointing out that there exist far more accurate and efficient methods of deciphering much harder codes than this, so this is not a particularly practical application of Markov Chain Monte Carlo. This project does, however, provide a beautiful example of the power of MCMC techniques.

2 Markov Chain Monte Carlo

I now move on to introduce the Markov Chain Monte Carlo algorithms in more detail, in particular the Metropolis-Hastings algorithm. This will lead on to a discussion of how best to apply these techniques to decoding substitution codes.

2.1 The Metropolis-Hastings algorithm

The first example of a method to sample from a distribution using Markov chains was proposed in 1953 by Metropolis *et al.* [25]. Known as the Metropolis algorithm, it had a huge impact, and many more papers on Monte Carlo simulation appeared, particularly in the physics literature, following this. The next really significant contribution came from Hastings in 1970. He, along with his student Peskun, showed that the Metropolis algorithm is a particular example of a large family of algorithms [17]. They studied the optimality of these algorithms, and introduced the Metropolis-Hastings algorithm, which is what I shall use as the basis of my decoding algorithm.

I first start by considering the general case, as proposed by Hastings. The main principle behind the Markov Chain Monte Carlo method is that any Markov chain on a finite state space which is *aperiodic* and *irreducible* has a unique stationary distribution, and that the k-step transition matrix $\mathcal{P}^k$ will

converge to this stationary distribution as $k \to \infty$. This means that, given our 'target' distribution π, we simply need to find a transition kernel $\mathcal{P}$ that satisfies the above conditions, and has π as a stationary distribution, i.e. satisfies $\pi\mathcal{P} = \pi$.

A simple way to do this is to consider a certain type of Markov chain, called a *(time) reversible* chain:

Definition 2.1. A stationary Markov chain with initial distribution π is called *reversible* if for all $x, y \in \mathcal{S}$,

$$\pi(x)p(x,y) = \pi(y)p(y,x). \tag{2.1}$$

where $p(x,y)$ is the one-step transition probability that the chain moves from state x to state y.

Equations (2.1) are called the *detailed balance* equations.

Proposition 2.1. *Let $\mathcal{P} = \{p(i,j)\}_{i,j\in\mathcal{S}}$ be a transition matrix on a countable state space $\mathcal{S}$, and let π be some probability distribution on $\mathcal{S}$. If for all $x, y \in \mathcal{S}$, the detailed balance equations (2.1) are satisfied, then π is a stationary distribution of $\mathcal{P}$.*

Proof. Fix $x \in \mathcal{S}$, and sum equalities (2.1) with respect to $y \in \mathcal{S}$ to obtain

$$\sum_{y\in S} \pi(x)p(x,y) = \sum_{y\in S} \pi(y)p(y,x) \ .$$

The left-hand side is equal to $\pi(x)\sum_{y\in\mathcal{S}} p(x,y) = \pi(x)$, and therefore, for all $x \in \mathcal{S}$,

$$\pi(x) = \sum_{y\in\mathcal{S}} \pi(y)p(y,x) \ .$$

□

This is the motivation for considering reversible chains: if a Markov chain is designed so as to be reversible, then it automatically follows from Proposition 2.1 that it will have π as its unique stationary distribution. It is usually much easier to verify that the detailed balance equations hold than it is to check the *general balance* equations: $\pi\mathcal{P} = \pi$.

Now consider the transition kernel $\mathcal{P}$ in the original form proposed by Hastings in [17]. This is designed to have the following form:

$$\begin{aligned} & p(x,y) &=& \ q(x,y)\alpha(x,y) \qquad (x \neq y) \\ \text{with} \qquad & p(x,x) &=& \ 1 - \sum_{y\neq x} p(x,y) \ . \end{aligned}$$

Here $\mathcal{Q} = \{q(i,j)\}$ is the transition matrix of an arbitrary Markov chain on $\mathcal{S}$, and $\alpha(i,j)$ is given by

$$\alpha(i,j) = \frac{s(i,j)}{1+t(i,j)} \tag{2.2}$$

where $s(i,j)$ is a symmetric function of i and j chosen so that, for all i, j, $0 \leq \alpha(i,j) \leq 1$, and

$$t(i,j) = \frac{\pi(i)q(i,j)}{\pi(j)q(j,i)}.$$

$q(x,y)$ is commonly referred to as the *proposal* probability, and $\alpha(x,y)$ as the *acceptance* probability.

Lemma 2.2. *Let $\{X_n\}$ be a Markov chain with transition matrix $\mathcal{P}$ as described above. Then $\{X_n\}$ satisfies detailed balance with respect to π.*

Proof.

$$\begin{aligned}
\pi(x)p(x,y) &= \pi(x)q(x,y)\alpha(x,y) \\
&= \frac{\pi(x)q(x,y)s(x,y)}{1 + \frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}} \\
&= \frac{\pi(x)q(x,y)s(y,x)\pi(y)q(y,x)}{\pi(y)q(y,x) + \pi(x)q(x,y)} \qquad (\text{since } s(x,y) = s(y,x)) \\
&= \frac{\pi(y)q(y,x)s(y,x)}{1 + \frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}} \\
&= \pi(y)q(y,x)\alpha(y,x) \\
&= \pi(y)p(y,x)\ .
\end{aligned}$$

□

The algorithm for producing a sample from the target distribution π, using a chain defined by the transitions as above, is as follows:

1. Set an (arbitrary) starting state X_0, and initialize the iteration counter $j = 1$.
2. Propose a new state y using the proposal density $q(X_{j-1}, \cdot)$.
3. Evaluate the acceptance probability $\alpha(X_{j-1}, y)$.
   Uniformly generate a random number $U \in [0,1]$.
4. If $U \leq \alpha(X_{j-1}, y)$, then the move is accepted, and we set $X_j = y$. If not, the chain does not move, i.e. $X_j = X_{j-1}$.
5. Increase the counter from j to $j+1$, and return to step 2 until convergence is reached.

Since this algorithm allows for rejection, it is clearly aperiodic. By Lemma 2.2 the chain satisfies detailed balance, so we need only check irreducibility in each specific case to ensure that it will converge to the required target distribution.

Now, in order to satisfy the constraint in (2.2) that $\alpha(x,y) \in [0,1]$,we must have

$$s(x,y) \leq 1 + min\{t(x,y),\ t(y,x)\}\ ,$$

since s is a symmetric function. There are clearly many possible functions $s(i,j)$ that satisfy the above constraint, and these lead to different MCMC algorithms. I will concentrate upon one of these only: that produced by taking equality in the above expression. This leads to

$$\alpha(x,y) = min\left\{1, \frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right\}\ . \tag{2.3}$$

Henceforth I shall refer to the algorithm produced by taking α as in (2.3) as the Metropolis-Hastings algorithm. It is this form of MCMC algorithm that will be used in the application to substitution codes henceforth.

2.2 Optimality of the Metropolis-Hastings algorithm

I now take a moment to justify the decision to use the Metropolis-Hastings algorithm. This entails the introduction of a result of Peskun that shows that this algorithm is optimal (in some sense) of all MCMC algorithms of the form considered by Hastings.

Let $\{X_n\}$ be an ergodic homogeneous Markov chain on some finite state space E, with transition matrix $\mathcal{P}$ and stationary distribution π. Let $f : E \to \mathbb{R}$. Define the *asymptotic variance* of X_n to be

$$v(f,\mathcal{P},\pi) = \lim_{n\to\infty} \frac{1}{n}\ Var\left(\sum_{k=1}^{n} f(X_k)\right).$$

When designing a simulation algorithm for our chain, we would clearly like to minimize $v(f,\mathcal{P},\pi)$ with respect to $\mathcal{P}$ and for fixed π. The following result of Peskun advises how to do this [4].

Theorem 2.3. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be reversible ergodic transition matrices on a finite state space E, with the same stationary distribution π. If $\mathcal{P}_1$ has all its off-diagonal terms greater than or equal to the corresponding off-diagonal terms of $\mathcal{P}_2$, then*

$$v(f,\mathcal{P}_1,\pi) \leq v(f,\mathcal{P}_2,\pi)$$

for all $f : E \to \mathbb{R}$.

Recall that in the Hastings algorithm, we have

$$p(x,y) = q(x,y)\frac{s(x,y)}{1+t(x,y)}$$

where $t(x,y)$ depends only on $q(x,y)$ and π. We aim to find the best MCMC algorithm in the Hastings class given a fixed $\mathcal{Q}$. It was observed above that to satisfy $\alpha(x,y) \in [0,1]$, with $s(x,y)$ symmetric, we must have

$$s(x,y) \leq 1 + min\{t(x,y),\ t(y,x)\}.$$

In the Metropolis-Hastings algorithm, we have equality in the above. As $\mathcal{Q}$ is fixed, $p(x, y)$ is dependent only on $s(x, y)$. It follows that the off-diagonal terms of $\mathcal{P}$, when using this algorithm, are greater than or equal to the corresponding off-diagonal terms of any other possible transition matrix. It hence follows from Theorem 2.3 that the Metropolis-Hastings algorithm is optimal with respect to asymptotic variance for fixed $\mathcal{Q}$.

2.3 Decoding Substitution Codes

I now introduce the idea of a random walk on a group.

Definition 2.2. A *group* G is a set with an associative multiplication $s, t \to st$, an identity id, and inverses s^{-1}. In particular, the symmetric group S_n is the group of permutations of a set containing n elements.

Note that if *Alph'* = *Alph* in Definition 1.1 then $f \in S_n$. From now on it will be assumed that this is the case, and I talk about 'applying a permutation to a character set'. This assumption is purely for simplicity: the deciphering method I shall develop can easily be applied to the more general case. Hence to encode a message all that need be considered is the application of a permutation σ to the relevant character set, and then the act of decoding reduces to the problem of finding the inverse permutation σ^{-1}.

This suggests the following idea. Whatever our original text, it can be viewed as a sequence of characters generated by some Markov chain. If we can assign a probability to each possible character transition (i.e. working with *digrams*), then we can calculate the likelihood of any permutation by applying the permutation to the text, and then multiply up the resulting sequence of transition probabilities. We then decide upon the 'most likely' decoding, $\hat{\sigma}^{-1}$, by choosing the permutation that has the greatest calculated likelihood.

For a simple example, suppose we are working with the (encoded) text $T = (t, a, c)$. We can thus restrict ourselves to working with the character set $Alph = \{a, c, t\}$. Suppose further that we have the following matrix of estimated transition probabilities:

$$\begin{array}{c} \\ a \\ c \\ t \end{array} \begin{array}{c} \begin{array}{ccc} a & c & t \end{array} \\ \begin{pmatrix} 0 & 0.3 & 0.7 \\ 0.5 & 0.05 & 0.45 \\ 0.3 & 0 & 0.7 \end{pmatrix} \end{array}.$$

There are six possible permutations of this character set ($|S_3| = 6$). These permutations are given in Figure 2.1, with the result of applying each to T, and the resulting likelihood of each permutation.

In this case we would say that the most likely permutation is $\{a, c, t\} \mapsto \{a, t, c\}$, since this has the greatest likelihood (0.35). This then, is $\hat{\sigma}^{-1}$, our best guess at the inverse permutation needed to decode the text T. (In this case, the decoding produces the word 'cat'.)

Permutation	Applied to T	Likelihood
$\{a, c, t\}$	(t, a, c)	$0.3 \times 0.3 = 0.09$
$\{a, t, c\}$	(c, a, t)	$0.5 \times 0.7 = 0.35$
$\{c, a, t\}$	(t, c, a)	$0 \times 0.5 = 0$
$\{c, t, a\}$	(a, c, t)	$0.3 \times 0.45 = 0.135$
$\{t, a, c\}$	(c, t, a)	$0.45 \times 0.3 = 0.135$
$\{t, c, a\}$	(a, t, c)	$0.7 \times 0 = 0$

Figure 2.1: Decoding some simple text

When we are dealing with a larger character set however, the above method of calculating the likelihood for each permutation becomes infeasible (since $|S_n| = n!$). The problem is that we have a (very large, but finite) set of permutations $S = S_n$ (for some fixed n), and we can calculate the unnormalized probability of any given permutation $\tilde{p}(s) : s \in S$ (by multiplying up transition probabilities, as above). What is required, however, is to be able to draw approximately from the normalized distribution:

$$p(\cdot) = \frac{\tilde{p}(\cdot)}{\sum_{s \in S} \tilde{p}(s)}.$$

Clearly the sum in the denominator is far too large to calculate, due to the size of the state space S. This suggests turning to MCMC techniques for an answer, since it has already been mentioned that they are extremely useful in problems involving the calculation of normalizing constants. The idea is then as follows: construct a Markov chain on the state space S (a 'random walk on the permutation group'), which has as its equilibrium distribution the posterior distribution for the true substitution code given the encoded message. This chain can then be run for a long time, until it is believed to be close to equilibrium, and then sampling from it can occur.

2.4 Constructing the Markov Chain

Before beginning to consider the application of the Metropolis-Hastings algorithm to the problem at hand, the character set needs to be decided upon, and the matrix of character transition probabilities obtained. The easiest way to do this empirically is to analyze a very large piece of text: I chose the novel *War and Peace* [18]. I made this decision based upon the constraints that it had to be a large piece of text to obtain 'good' probabilities (*War and Peace* contains approximately 3 million characters), and it had to be available on-line (ruling out anything too recent). Of course, this project could easily be carried out again using a different basis for the transition probabilities, although this will be further discussed later. *War and Peace* has a character set of size 79 (26 lower case letters, 26 upper case, 10 numbers and 17 punctuation). From now

on this character set will be referred to as *Alph*, and S_{79} will be denoted simply by S.It now becomes clear just how large the state space is (79!), and why the method used in our earlier example is so infeasible.

Now that the language model has been finalized, a suitable algorithm for updating the Markov chain must be designed. I have already shown the appropriateness of the Metropolis-Hastings algorithm, and so it simply remains to decide upon suitable proposal and acceptance probabilities ($q(x, y)$ and $\alpha(x, y)$ respectively).

I identified (in the earlier simple example) a method of evaluating the likelihood of any specific permutation. I now develop this slightly, and introduce the notation that will be used henceforth.

Suppose T is our encoded text. Then we can write

$$T = (t_0, t_1, \ldots, t_N) \quad \text{for some } N \in \mathbb{N}.$$

Then, if σ is some permutation in S, the 'likelihood of σ' is calculated by:

$$\tilde{p}(\sigma) = \beta(\sigma_{t_0}) \prod_{k=1}^{N} \gamma(\sigma_{t_{(k-1)}},\ \sigma_{t_k}) \tag{2.4}$$

where

- $\beta(\sigma_{t_0})$ is the prior probability that the initial letter of the decoded text is σ_{t_0} (the image of t_0 under the permutation σ).
- $\gamma(\sigma_{t_{(k-1)}},\ \sigma_{t_k})$ is the prior probability that the k^{th} letter is σ_{t_k} given that the $(k-1)^{th}$ letter is $\sigma_{t_{(k-1)}}$.

Under the assumption that the English language can be well modelled by a Markov chain, the target distribution for our chain is simply the normalized form of equation (2.4).

This just leaves the task of choosing the way in which to propose the moves that the chain can make. This requires a bit of consideration: we do not want to jump too wildly around the state space. We hence want to propose only permutations that are similar to the present permutation. This suggests proposing random *transpositions*: the two proposed characters in the current permutation are then transposed to get the new (proposed) permutation. If $x, y \in S$ differ by a single transposition, write $x \leftrightarrow y$. With a character set of size n, there are $\binom{n}{2}$ possible transpositions, and so if we choose these uniformly at random, we have that

$$q(\sigma,\ \phi) = \begin{cases} \frac{1}{\binom{n}{2}} & \text{if } \sigma \leftrightarrow \phi \\ \\ 0 & \text{otherwise.} \end{cases}$$

So in equation (2.3), we have $q(x, y) = q(y, x)$ if $x \leftrightarrow y$. This reduces the acceptance probability to the ratio $\pi(y)/\pi(x)$, where π is our target distribution. We have already seen that we only know our target distribution up to

a normalizing constant, but this constant now cancels out in the above ratio! This demonstrates why MCMC techniques are particularly useful when the normalizing constant is unknown: it does not feature in the Metropolis-Hastings algorithm.

I have now completely specified the nature of the Markov chain. Random transpositions of characters will be proposed, and a move between permutations σ and ϕ will be accepted with probability

$$\begin{aligned} \frac{\pi(\phi)q(\phi,\sigma)}{\pi(\sigma)q(\sigma,\phi)} &= \frac{\pi(\phi)}{\pi(\sigma)} \\ &= \frac{\beta(\phi_{t_0})}{\beta(\sigma_{t_0})} \prod_{k=1}^{N} \frac{\gamma(\phi_{t_{(k-1)}},\phi_{t_k})}{\gamma(\sigma_{t_{(k-1)}},\sigma_{t_k})}. \end{aligned} \tag{2.5}$$

The state space for this chain is clearly irreducible, since the chain can move between any two given permutations by a sequence of transpositions. Thus the chain should, since it satisfies the detailed balance equations, converge to its stationary distribution. The next task is to consider the best way of implementing this algorithm on a computer, and this is what I proceed to do in the next section.

3 Computer Implementation

When it comes to running a Markov Chain Monte Carlo simulation, the only practical method available is of course to use a computer. For the purpose of this project I chose to work in `Mathematica`, as I have some experience of working with this package in the past. It must be pointed out, however, that the programs produced would run much faster in a language such as `Java`: I unfortunately did not have time to attempt such an implementation. All the relevant `Mathematica` code produced for this project is included in Appendix A.

3.1 The Acceptance Probability

In the breakdown of the Metropolis-Hastings algorithm given in Section 2.1, it is clear that the only step requiring some consideration is Step 3: calculation of the acceptance probability (since the proposal probabilities have been designed to be uniform). The formula in equation (2.5) involves at least $2(N+1)$ calculations in its present form, where N is the length of our text T. This is awfully large, and would considerably decrease the speed of the program as the length of the text increases. However, as the chain is only considering moves between permutations that differ by a single transposition, only a relatively small number of characters will change when the new proposed permutation is applied to T. For example, if it proposes swapping characters i and j, then the new proposed

decoding is identical to the present one, apart from the fact that all occurrences of character i have been replaced with j, and vice versa.

This suggests rewriting equation 2.5 in the following way.

$$\begin{aligned}\frac{\pi(\phi)}{\pi(\sigma)} &= \frac{\beta(\phi_{t_0})}{\beta(\sigma_{t_0})} \prod_{k=1}^{N} \frac{\gamma(\phi_{t_{(k-1)}},\ \phi_{t_k})}{\gamma(\sigma_{t_{(k-1)}},\ \sigma_{t_k})} \\ &= \frac{\beta(\phi_{t_0})}{\beta(\sigma_{t_0})} \prod_{x,y \in S} \frac{\gamma(x,\ y)^{\,\#_\phi(x,\,y)}}{\gamma(x,\ y)^{\,\#_\sigma(x,\,y)}}\ , \end{aligned} \tag{3.1}$$

where $\#_\phi(x, y)$ is the number of times character y follows character x when permutation ϕ is applied to T. But if the chain is currently at permutation σ and we propose to transpose characters i, j (producing permutation ϕ), we have that $\#_\phi(x, y) = \#_\sigma(x, y)$ if $x, y \notin \{i, j\}$, so most of the above terms cancel. Equation (3.1) then reduces to

$$\frac{\pi(\phi)}{\pi(\sigma)} = \frac{\beta(\phi_{t_0})}{\beta(\sigma_{t_0})} \prod_{x \in S} \frac{\gamma(x,\ \{i,j\})^{\,\#_\phi(x,\,\{i,j\})}\ \gamma(\{i,j\},\ x)^{\,\#_\phi(\{i,j\},\,x)}}{\gamma(x,\ \{i,j\})^{\,\#_\sigma(x,\,\{i,j\})}\ \gamma(\{i,j\},\ x)^{\,\#_\sigma(\{i,j\},\,x)}} \tag{3.2}$$

It is also worth pointing out here that, in most cases, $\beta(\phi_{t_0}) = \beta(\sigma_{t_0})$, unless the first letter of T under the current permutation is one of those in the proposed transposition.

The number of calculations required at each iteration to compute the acceptance probability α has thus been substantially reduced. Equation (3.2) still involves the product of quite a large number of probabilities though, and if I were to use this version when writing my program, I would be very likely to incur errors due to rounding. For the final form of α, the above equation is therefore rewritten using logarithms to provide more numerical stability:

$$\begin{aligned} log\ \pi(\phi) - log\ \pi(\sigma) &= \big[log\ \beta(\phi_{t_0}) - log\ \beta(\sigma_{t_0})\big] \\ &\quad + \sum_{x \in S} \big[\#_\phi(x, \{i,j\}) - \#_\sigma(x, \{i,j\})\big]\ log\ \gamma(x,\ \{i,j\}) \\ &\quad + \sum_{x \in S} \big[\#_\phi(\{i,j\}, x) - \#_\sigma(\{i,j\}, x)\big]\ log\ \gamma(\{i,j\},\ x)\ . \end{aligned} \tag{3.3}$$

The decision to use this form of acceptance probability means that the program now does not need to apply each permutation to T in order to calculate α. Instead, it just needs to keep track of the number of each type of character transition, $\#_\phi(x, y)\ \forall x, y \in S$, for any given permutation ϕ. The obvious way to do this is to use matrices, which will also help to increase the speed of the programs, as will be explained shortly.

From now on, I will refer to the 'transition matrix of a permutation σ':

$$\mathcal{M}_\sigma = \begin{array}{c} \\ Alph_1 \\ Alph_2 \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccc} Alph_1 & Alph_2 & \dots \end{array} \\ \begin{pmatrix} m_{11} & m_{12} & \dots \\ m_{21} & m_{22} & \dots \\ \vdots & \vdots & \ddots \end{pmatrix} \end{array}$$

where $m_{ij} = \#_\sigma(Alph_i, Alph_j)$, and I will usually just write

$$\mathcal{M}_\sigma = \begin{pmatrix} m_{11} & m_{12} & \dots \\ m_{21} & m_{22} & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}.$$

This means that in matrix form, equation (3.3) reads:

$$log\ \pi(\phi) - log\ \pi(\sigma) = \left[log\ \beta(\phi_{t_0}) - log\ \beta(\sigma_{t_0})\right] + (\mathcal{M}_\phi - \mathcal{M}_\sigma)(log\ \mathcal{P})^T \quad (3.4)$$

where $\mathcal{P}$ is the matrix of empirical transition probabilities obtained from *War and Peace*, and $(log\mathcal{P})$ is the matrix whose $(i,j)^{th}$ entry is the log of the $(i,j)^{th}$ entry of $\mathcal{P}$. So the chain accepts a move from σ to ϕ with probability α, where

$$log\ \alpha = min\left\{0,\ log\ \pi(\phi) - log\ \pi(\sigma)\right\}.$$

The algorithm for the chain is then as follows:

1. calculate the transition matrix for the current state σ
2. propose new permutation, ϕ
3. calculate new transition matrix for ϕ
4. calculate the acceptance probability using equation (3.4).

Again the task of calculating α is simplified by the fact that when we transpose two characters, the rest are unaffected. This means that if characters i and j are transposed, only the entries in the rows and columns of the transition matrix corresponding to these characters will change. So most of the entries of the matrix $(\mathcal{M}_\phi - \mathcal{M}_\sigma)$ in equation (3.4) are zero. This suggests that the speed of computation can be increased by looking for simple matrix operations to apply to the current $\mathcal{M}_\sigma$ in order to obtain the matrix for the newly proposed permutation.

A little experimentation shows that swapping characters i and j corresponds to first swapping rows $Alph_i$ and $Alph_j$ in $\mathcal{M}_\sigma$, and then swapping columns $Alph_i$ and $Alph_j$. For example, suppose we are working with a character set of size 5, and the chain is currently at permutation σ, with

$$\mathcal{M}_\sigma = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} & m_{15} \\ m_{21} & m_{22} & m_{23} & m_{24} & m_{25} \\ m_{31} & m_{32} & m_{33} & m_{34} & m_{35} \\ m_{41} & m_{42} & m_{43} & m_{44} & m_{45} \\ m_{51} & m_{52} & m_{53} & m_{54} & m_{55} \end{pmatrix}.$$

Now suppose we propose to transpose characters 1 and 4, resulting in permutation ϕ. To obtain $\mathcal{M}_\phi$, we simply swap rows 1 and 4 of $\mathcal{M}_\sigma \ldots$

$$\begin{pmatrix} m_{41} & m_{42} & m_{43} & m_{44} & m_{45} \\ m_{21} & m_{22} & m_{23} & m_{24} & m_{25} \\ m_{31} & m_{32} & m_{33} & m_{34} & m_{35} \\ m_{11} & m_{12} & m_{13} & m_{14} & m_{15} \\ m_{51} & m_{52} & m_{53} & m_{54} & m_{55} \end{pmatrix}$$

$\ldots$ and then swap columns 1 and 4 to get:

$$\mathcal{M}_\phi = \begin{pmatrix} m_{44} & m_{42} & m_{43} & m_{41} & m_{45} \\ m_{24} & m_{22} & m_{23} & m_{21} & m_{25} \\ m_{34} & m_{32} & m_{33} & m_{31} & m_{35} \\ m_{14} & m_{12} & m_{13} & m_{11} & m_{15} \\ m_{54} & m_{52} & m_{53} & m_{51} & m_{55} \end{pmatrix}$$

Now, when we subtract $\mathcal{M}_\sigma$ from $\mathcal{M}_\phi$, we are left with:

$$\begin{pmatrix} m_{44}-m_{11} & m_{42}-m_{12} & m_{43}-m_{13} & m_{41}-m_{14} & m_{45}-m_{15} \\ m_{24}-m_{21} & 0 & 0 & m_{21}-m_{24} & 0 \\ m_{34}-m_{31} & 0 & 0 & m_{31}-m_{34} & 0 \\ m_{14}-m_{41} & m_{12}-m_{42} & m_{13}-m_{43} & m_{11}-m_{44} & m_{15}-m_{45} \\ m_{54}-m_{51} & 0 & 0 & m_{51}-m_{54} & 0 \end{pmatrix}.$$

So the number of calculations required to compute the acceptance probability has been reduced to just a few: row and column operations are simple calculations for a computer. The algorithm can even be sped up slightly more by observing the pattern of entries in the final matrix, $(\mathcal{M}_\phi - \mathcal{M}_\sigma)$. It is possible to force the computer to calculate each non-zero entry of this matrix individually: this involves the calculation of $4(n-1)$ non-zero entries (where $|Alph| = n$), compared to the $2(N+1)$ calculations in equation 2.5. Since in nearly all situations, $n \ll N$, this is a significant improvement. With a large character set this method is marginally faster than using matrix operations, but the gain is minimal.

3.2 Program-writing

Now that a relatively fast method of computing the acceptance probability has been developed, the construction of the programs that will be needed to decode

a portion of text can be considered. All relevant code is included in Appendix A with some commenting. In this subsection I do not go into much detail: the aim is merely to present some of the methods used, explain why they were used, and describe some of the problems encountered.

The first issue that requires some consideration is that of obtaining the empirical character transition probability matrix, $\mathcal{P}$. I have already described my reasons for using *War and Peace* as the basis for this matrix, and the program that was used to convert the novel into transition matrix form is named `prob`. Entries in this matrix then need to be converted into probabilities (just dividing each entry by the relevant row-sum) and logarithms taken. The program that does is this is `logprob`. Although this stage seems rather trivial, it is mentioned because some thought is needed when we have a zero entry in the transition matrix: we clearly do not want to be summing terms involving $log(0)$ when it comes to calculating α. On the other hand, if these entries of $(log\ \mathcal{P})$ are simply set to zero, then the chain will always accept a move to a permutation that has likelihood zero. A little trial and error work resulted in my defining $log(0) := -12$: this seems to be small enough to prevent the chain from accepting such a move. The matrix $(log\ \mathcal{P})$ is called `matwrpc` whenever it appears in a program.

The other matrix obtained from *War and Peace* is `freq`. This is a column vector of length 79, whose i^{th} entry is the relative frequency of character $Alph_i$ in the novel. This is used in the calculation of the acceptance probability, since

$$\beta(\phi_{t_0}) = \texttt{freq}\,(\phi_{t_0})$$

in equation (3.4).

The two programs that are involved in calculating the acceptance probability are named `swap3` and `swap`. The former of these carries out the row swap followed by the column swap to produce the transition matrix for the proposed permutation. This matrix is accepted as the starting matrix for the next iteration if the chain moves to this permutation. The other program calculates the matrix $(\mathcal{M}_\phi - \mathcal{M}_\sigma)$ by explicitly determining each non-zero element, as described on the previous page.

It has already been mentioned that the program does not need to apply each permutation to the text at each iteration in order to calculate it's likelihood. This means that all it needs to do is store the original permutation, and then to update this by applying any accepted transposition to the current permutation. The program stores the current permutation under the name `alph`, and applies transpositions using the program `tranAlph`. When the chain is terminated, the program `alphify` applies the final permutation to the text, so producing the actual decoding.

The final program to run the algorithm is entitled `solvitbound`, and runs according to the following algorithm:

1. calculate the transition matrix of T under the identity permutation (i.e. take `alph` = `Alph`)

2. propose a transposition of two characters at random

3. calculate the transition matrix of the permutation arising from the above transposition

4. calculate the prior probability of the first letter of T under the new permutation

5. calculate the acceptance probability

6. if the move is accepted, set the current transition matrix to be that found in step 3

7. return to step 2 until the chain converges[1] or the maximum number of iterations is reached

8. produce appropriate diagnostic plots

9. decode the text using the final permutation.

The main problem that I encountered whilst programming was that of tracking the effects of the chain accepting a move. Not only does the transition matrix change, but the first letter of the decoded text must also be tracked, and this produced quite a few indexing problems. The main issue is that of logically following through the indexation. For example, to find the prior probability of the initial letter under the current permutation `alph`, three steps are involved:

- the first letter under `alph` must be calculated
- the position of this letter in `Alph` must be found
- the relevant entry in `freq` must be referenced.

The above method works well, but there may well be a much more efficient way of doing this.

This concludes the discussion of the decoding program that I developed. Further commenting is provided with the code, and this should help to provide more of an insight into the methods that were used. I now proceed to discuss some diagnostic methods that I used to assess the efficacy of my program.

3.3 Convergence Diagnostics

The convergence of Markov chains produced under Markov Chain Monte Carlo simulation can be notoriously difficult to assess. There is usually no definitive way of deciding that the chain has converged, and so convergence diagnostics are used as a heuristic method of assessing this. These methods normally warn if convergence has not been reached, but can not generally be used to guarantee that the chain is close to equilibrium.

A common diagnostic is to plot a summary statistic of the chain, especially if the chain lives in a space of high-dimension. It is usually the case that these

[1] This will be discussed in detail in Section 3.3

plots display atypical behaviour early on, and this can often be attributed to initialization bias caused by the choice of starting state for the chain. Once the chain has reached equilibrium, its distribution becomes more constant over time, and so a more stable graph is to be expected. The reverse of this is not necessarily true however: if the path of the chain appears stable, but equilibrium has not been reached, then the chain is said to be experiencing *meta-stability*. This can be caused by the stationary distribution being multi-modal. In the decoding application this is equivalent to the chain finding a pretty good permutation (but not the target one) and 'getting stuck' there for a time before moving on again. If we were to sample during this time, believing that convergence had been reached, we would clearly not find the target permutation. It is therefore common to choose a range of starting states for the chain, which are widely dispersed in the sample space. A Markov chain is then simulated from each of these starting states, with the idea being that we can detect meta-stability if it occurs since the different chains will get stuck in different modes of the stationary distribution.

In this project I decided upon three methods of assessing convergence. Using the three together improves the chance of avoiding sampling when the chain is in meta-stability, as described above.

The three diagnostics used are:

- Proportion of Transpositions Accepted
- Negative Log-Likelihood
- Percentage Accuracy

The first of these simply counts how often the chain accepts a new permutation. This is displayed as a percentage every few hundred iterations (dependent upon the maximum allowable number of iterations). When the chain is started, it is likely to be visiting states of low likelihood and so we would expect to see quite a high acceptance rate at first (approximately 50%). As the chain approaches equilibrium, it becomes less likely that a better permutation will be proposed and accepted, and so the percentage of transpositions accepted should drop significantly. Hastings defines the *rejection rate* of a chain as the proportion of times t for which $X_{t+1} = X_t$ [17]. He recommends recording the rejection rate (effectively what my diagnostic is doing), since a high rejection rate "may be indicative of a poor choice of initial state or transition matrix, or of a 'bug' in the computer program". A consistently high rejection rate is clearly undesirable, as this would lead to long periods of meta-stability: this would not only increase the number of iterations needed to reach equilibrium, but could also fool us into believing that the chain had converged.

Before explaining the second of these diagnostics, I first need to explain exactly what I mean by negative log-likelihood. If the chain is currently at permutation σ, then the negative log-likelihood of σ is defined to be

$$- \mathcal{M}_\sigma (log\ \mathcal{P})^T \ .$$

This is calculated and printed by the program `likeprint`. The negative log-likelihood of a permutation is a positive number, rounded by the program to the nearest integer (the exact version is used by the main program when calculating the acceptance probability). Since the log-likelihood of any permutation is negative, and we expect it to increase as the chain approaches equilibrium, we would clearly expect the negative log-likelihood to *decrease* as the chain converges. If the required permutation (σ^{-1}) is known, then the negative log-likelihood of any permutation can be compared with that of σ^{-1} as a measure of how close the chain is to equilibrium. This permutation is, of course, unknown in all practical cases, but when testing the programme I started by encoding some text and then trying to decode it. This meant that I knew σ, and hence σ^{-1}, and so this diagnostic proved particularly useful when checking that the program actually worked.

The third diagnostic that I used again requires σ^{-1} to be known in advance, making it useless when faced with some encoded text. The percentage accuracy of any permutation ϕ is the proportion of letters correctly decoded when ϕ is applied to T. This is not simply the number of characters in ϕ which agree with σ^{-1}: it also takes into account the frequency with which each character occurs in the decoded text. This diagnostic is calculated by the program `percentage`, and is rounded to the nearest integer. It is not calculated by applying each permutation to T and comparing the two decodings, but instead compares the two permutations and uses the transition matrix of ϕ to take into account the character frequencies in the text.

When presenting my results I will concentrate upon the second two of these diagnostics. Although tracking the rejection rate was very helpful when developing the programs, it proved not to be as useful as the other two when trying to identify convergence.

4 Preliminary Language-Analysis

Before presenting the main findings of this project, I first take a moment to discuss the type of text that I used my program to decode. Since I used the novel *War and Peace* as the basis of the matrix ($log\ \mathcal{P}$), it seemed intuitive that the program would perform best on similar sorts of text. This is because the program works by assuming that the text it is trying to decode has similar character transition probabilities when decoded. Hence, if the text T is too dissimilar to *War and Peace*, the program will be quite useless. This led me to my decision to use the program to decode scrambled versions of English novels.

A little more justification is needed here though: just how similar are these novels? In an attempt to answer this question, I selected the first chapters of nine novels and looked at their letter frequencies. These texts were of varying lengths, by different authors, and from a range of eras. They are: *Oliver Twist* (Dickens), *Treasure Island* (Stevenson), *Robinson Crusoe* (Defoe), *Pride and Prejudice* (Austen), *Jane Eyre* (Brontë), *War and Peace* (Tolstoy), *Far from*

the Madding Crowd (Hardy), *Framley Parsonage* (Trollope) and an unpublished children's story by W. Kendall. (With the exception of the last, these are all available from the Project Gutenberg website [18].) Henceforth I shall refer to them by author.

Of course it is the transition probabilities which are really important to the program, but it is a much bigger job to compare 79^2 frequencies between the texts, as opposed to just 79. The results are displayed in Figure 4.1: the numbers on the x-axis refer to the position of the characters in `Alph` (see Appendix A).

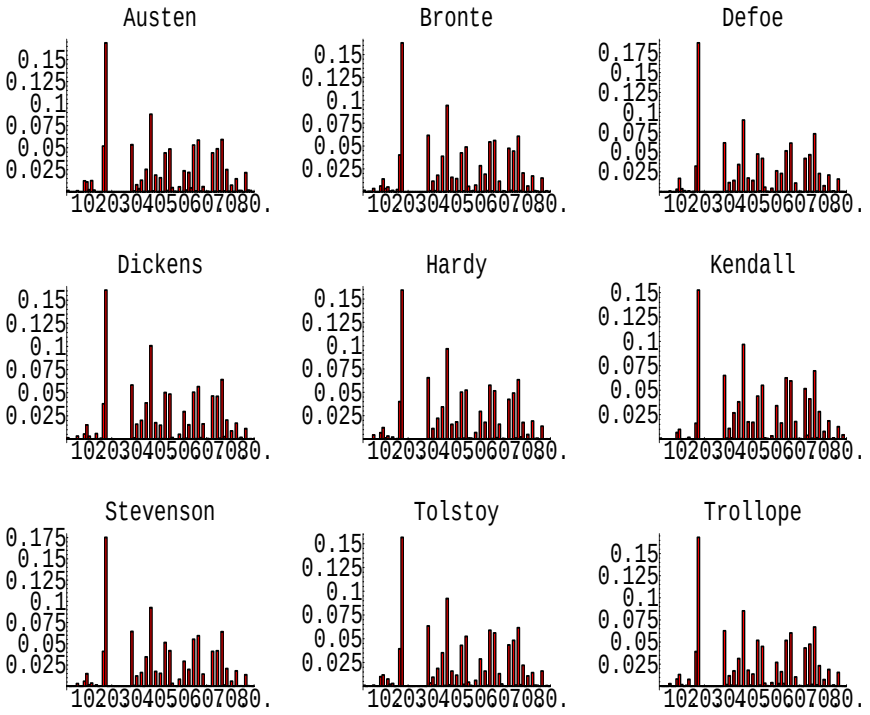

Figure 4.1: Character frequencies for the nine novels

The mode at position 17 corresponds to 'space', and the next two peaks (at 28 and 36) correspond to the letters 'a' and 'e' respectively, as might be expected.

It would appear then that character frequencies are fairly consistent between novels: this is encouraging, as it indicates that these texts may decode well when using *War and Peace* as a basis. The above graph does not, however, tell us anything about the character transitions. I have already mentioned that producing histograms with 79^2 bars is infeasible, but I have already discussed another method of assessing character transitions: the negative log-likelihood.

I hence calculated the negative log-likelihood of the first chapters of the nine novels, and plotted these against their lengths (since negative log-likelihood almost certainly increases with length of text). The graph of this is presented in Figure 4.2, with the simple linear regression line drawn in.

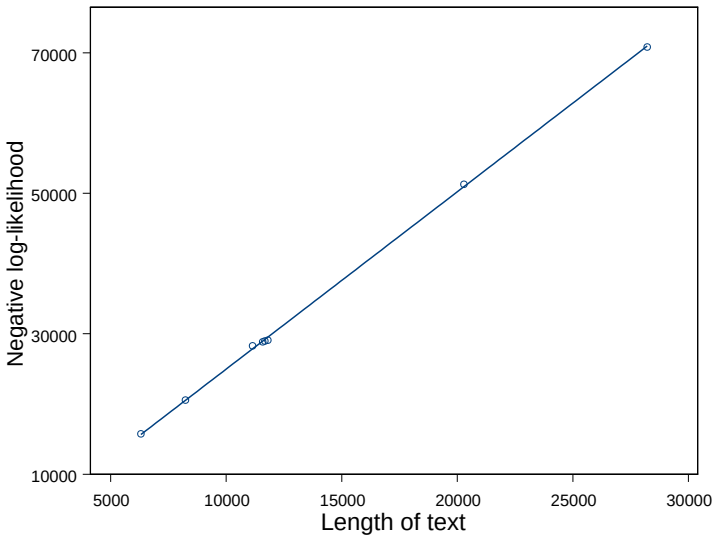

Figure 4.2: Linear regression of negative log-likelihood against length.

This best-fit line has a y-intercept of -281, and gradient 2.5. We would expect a y-intercept of zero of course, but the value obtained is reasonable in light of the scale of the data points. The fact that these points lie so close to a straight line suggests that, if negative log-likelihood turns out to be a good indicator of convergence, we can use this line to predict to what level the negative log-likelihood should drop when the chain reaches equilibrium. This would be extremely useful, for two reasons:

- we do not need to know σ^{-1} to use the negative log-likelihood diagnostic - i.e. this can be used in 'real' applications of the program
- we will not be caught out by meta-stability, since we can predict the approximate value at which the diagnostic should stabilize, just by considering the length of the text that we are decoding.

It remains to be seen whether or not the negative log-likelihood is actually that good at assessing convergence, but if it is then we have found an extremely useful diagnostic measure.

It hence appears that the program should be quite effective at decoding text which is 'similar enough' to the basis of the matrix ($log\ \mathcal{P}$). To see how dissimilar other sorts of text would be, I selected six texts of various kinds and performed the above analysis of them. These texts were: novels in German, Dutch and Welsh; some Latin prose; an article from a computer magazine; an extract from a biochemical paper.

I chose these as I would expect them to be different from *War and Peace*, but on different scales. For instance, the computer magazine and biochemistry paper contain mainly English, but with more abbreviations and acronyms than used by Tolstoy. I would also expect the Latin text to have similar character transitions to English, but the Welsh and Dutch texts to be completely dissimilar.

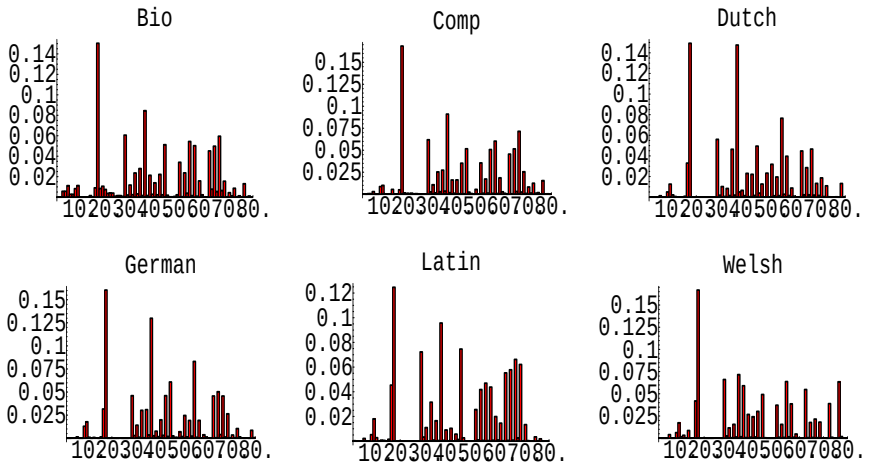

Figure 4.3: Character frequencies for the other texts

The character frequencies for these texts are presented in Figure 4.3 . The histograms for the computer magazine article and the Latin text are similar to that of the novels, but the others display some differences. The biochemical article has many more capital letters than the novels (the bars in the histograms for the novels are not separated - the 'spaces' are the bars for the capital letters). This is almost certainly due to more acronyms, descriptions of DNA chains etc. The Dutch text has a very high frequency of the letter 'e', whilst the Welsh text favours the letter 'y'.

The most significant difference between these texts and the novels is only really identified by considering the negative log-likelihood for these texts. Figure 4.4 shows how these compare to the best-fit line found in Figure 4.2 (the new points are represented as triangles).

The negative log-likelihood for these texts is clearly quite different from that of the novels. Although this is not conclusive proof that the program will not decode these texts, it does indicate that their character transition frequencies

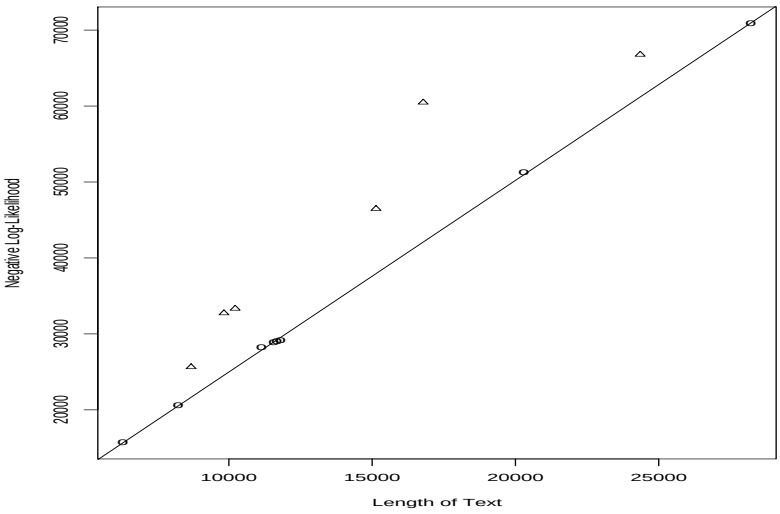

Figure 4.4: Negative log-likelihood vs length for the other texts.

are different from those in *War and Peace*, and so these probably would not decode so well. It also means that, if we were to try to decode texts of this kind using *War and Peace* as a basis, the negative log-likelihood diagnostic would not be as useful as it will hopefully prove to be with the novels.

As a preliminary form of analysis therefore, it appears that English novels do display similar characteristics concerning character frequencies and transitions. This is encouraging, as it indicates that the program should be able to decode any text of this kind. Other kinds of text are quite different as we have seen. and so it is unlikely that the program as it stands would decode these. This problem is easily resolved of course: if we wanted to decode something in Welsh for example, then we could simply create a new ($log\ \mathcal{P}$) matrix using some Welsh text. This should then work just as well as the English version: any substitution code can theoretically be deciphered, as long as a suitable basis for the character transition matrix can be found. As I have chosen to use *War and Peace* for my project however, I will henceforth consider principally English novels, and proceed to carry out my analysis on just four of these: *Kendall*, *Brontë*, *Dickens* and *Stevenson*.

5 Results

The program works! The final working version took me quite a long time to develop, but is indeed capable of producing quite an accurate decoding of the sort of text I am considering. The first point that must be emphasized, however, is that the program does not always decode the text to a satisfactory degree, as we shall see. It must be remembered that the Markov chain evolves according to numbers produced randomly by the computer: although the theory states that the chain should eventually converge, practical constraints (such as imposing a maximum number of iterations) often prevent equilibrium being reached. One of the areas that I investigated was how many iterations it takes for the chain to converge, and these results are presented in section 5.3.

The results presented in this section come from a number of versions of the final program `solvitbound` (the names of which, for reference, all start '`solv`'). These all use the same algorithm to construct the Markov chain, but the arguments and output change according to exactly what is desired. An example of the program `solveplot2` working is provided in Appendix **??**: the output comprises of the negative log-likelihood and percentage correct at the start and finish, a graph of the negative log-likelihood against the number of iterations, and the first few lines of the decoded text.

5.1 Length of Text

Once it was established that the program worked, I used it to investigate ways of improving the efficiency and accuracy of the decoding. The first thing that I chose to consider was the length of the text T that the program was being asked to decode. Of course, the running time of the program is fairly independent of this length: once the initial transition matrix is constructed (which *is* length-dependent), the program runs completely on matrix operations. Since changing the length of the text cannot therefore significantly improve the efficiency of the program, it may well be asked why it is worthy of any consideration. The reason is that the program works by comparing character transition frequencies in T with those in *War and Peace*. Hence, if the text is too short, there may not be enough information about these transitions available. This suggests that the program will decode longer text to a better degree of accuracy.

To investigate this, I selected two of the novels (*Kendall* and *Dickens*) and used the program to decode different lengths of these. For each run of the program, the encoding of the text and the random seed (defining the proposed transpositions and the uniform random variable contributing to the acceptance probability) were fixed. Four lengths were used for each text. This was implemented using the program `solveplot2`: this was an early version of `solvitbound`, and ran very slowly in comparison. Consequently, I imposed an upper bound of $25,000$ iterations for each run: this is relatively small and the chain is unlikely to be in equilibrium by this stage as we shall see later, but was sufficient to gain some results in this area. Each run of four decodings was repeated 20 times for each piece of text. An example of the result of one run of four decodings of

Dickens is included in Appendix **??**.

In this instance, the first $2,000$ and first $4,000$ characters both decoded to 99% accuracy, whilst the other two lengths (first $1,000$ and first $6,000$ characters) did not perform well at all after $25,000$ iterations. (It is worth pointing out here that the length of time the program takes to run is indeed independent of the length of text: each decoding in this example took approximately 990 seconds.)

This was not the case with every run, however. Figure 5.1 is a box-plot of the percentage error of decoding these four lengths of *Dickens* at $25,000$ iterations. Although the results from the $2,000$ and $4,000$ character runs do not tell us much, the marked difference between the error rates at $1,000$ and $6,000$ characters supports the proposal that longer text should decode better.

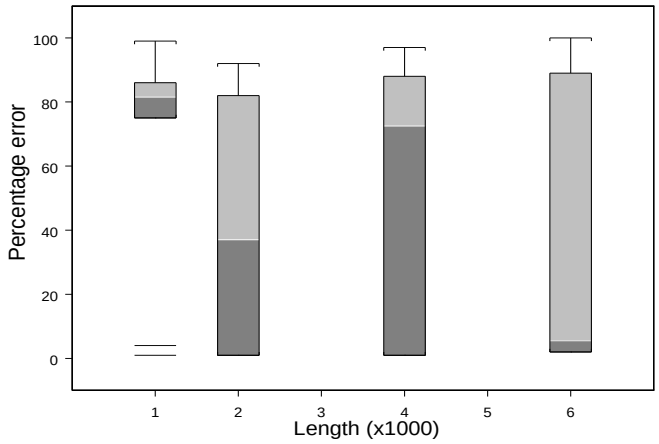

Figure 5.1: Percentage error vs length of text for *Dickens*

This was not the trend obtained from the results of decoding *Kendall* in the above manner though. Figure 5.2 is the corresponding box-plot for this text. This shows a trend that is almost completely the opposite to that in Figure 5.1: the percentage error appears to be increasing with text length. This is far from conclusive, however, since the sample size is unfortunately quite small. The range of values at each length is very large, due to the fact that at least one extremely accurate decoding was produced for each length of text, but in many cases equilibrium was not reached. It is almost certainly the case that many of the decodings with high percentage errors would actually decode very well if allowed to run for longer, as will be investigated in subsection 5.3. Unfortunately these results were obtained, as mentioned, before I had developed the algorithm for my final program, and time constraints meant that I was not able to repeat

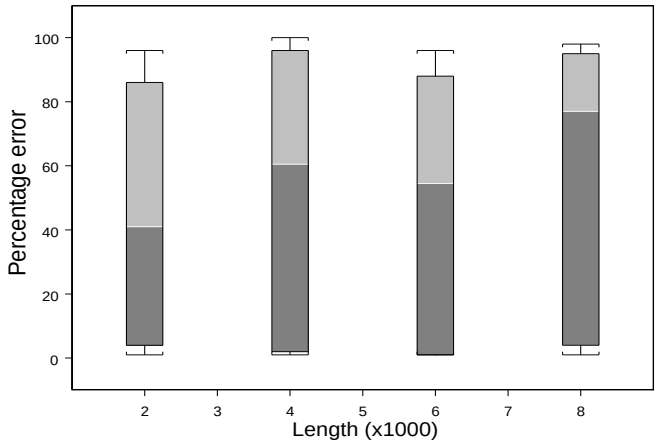

Figure 5.2: Percentage error vs length of text for *Kendall*

this part of my investigation with `solvitbound`.

These results seem pretty inconclusive therefore: they certainly do not support the idea that more text should produce a better decoding. Going back to the reasoning behind this idea though, there is another effect introduced by increasing text length that was not considered. In my earlier argument, I stated that increasing text length provides the program with more information regarding character transition frequencies. This assumes, however, that these frequencies become more consistent with those in *War and Peace* as more text is included. But this may not always be the case: for example, the character frequencies for the first half of the correctly decoded text may be similar to those in *War and Peace*, whilst those for the second half of the text are markedly different. This may arise for any number of reasons. For example, a new proper noun may be introduced suddenly: this may significantly increase the number of transitions between two characters that are not commonly seen together in everyday English, e.g. the '*nz*' in the name '*Rapunzel*' (*Kendall*).

5.2 Splitting the Text

It would appear then that we cannot expect to significantly increase the accuracy of a decoding simply by introducing more text, unless we can be sure that the character transitions become more consistent with our basis text as we do this. This suggested that I should next try splitting up the encoded text, and then using the program to decipher each part separately.

This was carried out in a manner similar to the above investigation, but I did not simply add more text at each run. I instead partitioned each novel into

four overlapping parts. The text partitions are labelled 1-4 in each case, and the sequence of characters in T to which each corresponds is as follows:

Partition	*Dickens*	*Kendall*
1	1-4,000	1-3,000
2	2,000-6,000	1,500-4,500
3	4,000-8,000	3,000-6,000
4	1-8,000	1-6,000

The aim of this was to see if certain sections of text decode better than others: if so, then this may help to explain the results obtained in Section 5.1. For example, if we were to find that the first n characters of T decode well, but that the following N characters do not decode so well, then it may well be the case that the first n would decode better than the first $n+N$ (a longer piece of text).

Each run of four decodings was carried out 20 times for each piece of text again (for 25,000 iterations), keeping the encoding and seed constant for each set of four. The results of this are displayed in Figures 5.3 and 5.4.

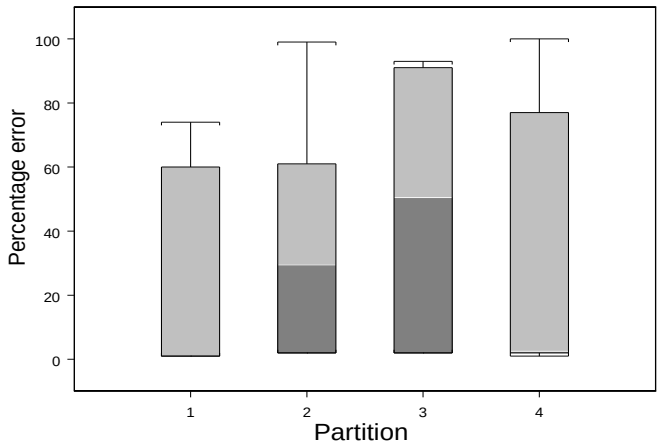

Figure 5.3: Percentage error vs partition of text for *Dickens*

These plots clearly show that there is a lot of variation within a piece of text. For example, in *Dickens* the first 3,000 characters decoded much better than the final 3,000. Together, however, they decode very well (partition 4). Figure 5.4 is again different though. For *Kendall* it appears that both the first and second halves (partitions 1 and 3) decode well, but that combining them

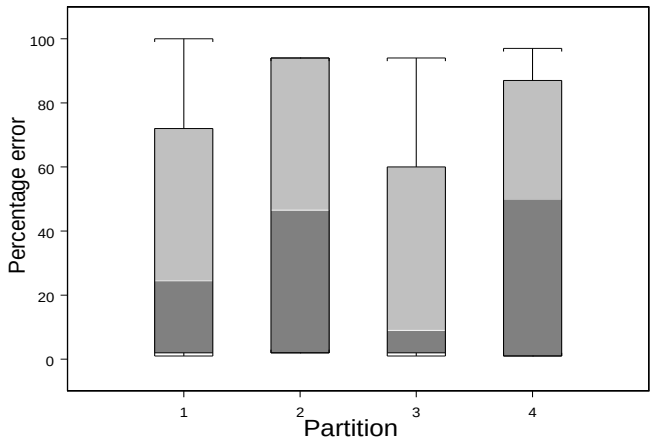

Figure 5.4: Percentage error vs partition of text for *Kendall*

actually increases the percentage error. This is in keeping with Figure 5.2, where increasing the length of the text worsens the deciphering accuracy.

It appears then, that there are some quite complex interactions occurring within the text. It is worth emphasizing that the above results were obtained by keeping the random seed constant for each run of four decodings, so the results of different partitions are directly comparable. This variation within text could have severe implications for our aim to develop the best method (within the MCMC framework) of deciphering substitution codes. The fact that the observed variation does not seem to follow any obvious pattern (evident from the direct comparison of *Dickens* and *Kendall* above) does not suggest any ways of dealing with it.

One method, however, may be to partition the text and decode each portion as above. This would then lead to multiple suggestions for the required permutation σ^{-1}. Out of these, the permutation with the lowest negative log-likelihood could be accepted as $\hat{\sigma}^{-1}$, and then this could be applied to T to achieve the final decoding.

Interested to see if this would actually improve the frequency with which a 'good' decoding was achieved, I again considered the results obtained from partitioning the text. In doing this I decided to define a 'good' decoding as one that achieves an error rate of less than 10%: once the program gets this close, it is usually quite easy to identify and rearrange the incorrect characters. The outcome of this was quite interesting: it appears that the frequency with which a good decoding is achieved can be significantly increased by splitting the text into two halves and using the program separately on each. With *Dickens*,

decoding the whole text produced a good result 60% of the time, whilst doing the two halves separately increased this to 80%. An even better result was observed with *Kendall*: here the frequency was increased from 40% to 80%. Although the sample size is again quite small, the intuition behind this idea makes sense and is supported by the results obtained. My first recommendation for increasing the accuracy of the decoding is therefore to consider splitting the text into two parts (or possibly more) and attempting to decipher each separately.

5.3 Number of Iterations

One of the most important issues associated with this project is, of course, how many iterations it takes for the chain to converge. This is investigated in this section, using a faster program than `solveplot2`, which I had thankfully developed in time to do this.

Having this final program available meant that I was able to produce a larger sample size, and also to run each chain for far more than the $25,000$ iterations to which the program was restricted above. In order to investigate the number of runs to reach equilibrium, I first had to decide how to define equilibrium. From the earlier results, it appeared that the chain could reach at least 99% accuracy in some cases, and so I decided to use this as my definition of convergence. I selected four of the nine novels (*Kendall, Dickens, Brontë* and *Stevenson*) and ran the program 60 times on each. The program was set to terminate if 99% accuracy was attained, or if the maximum allowable number of iterations was reached ($200,000$).

In addition to considering those that 'reached equilibrium', I also looked at the number of unfinished runs which had reached at least 90% accuracy by $200,000$ iterations: it was highlighted above that at this level of accuracy it is often easy to 'fill in the blanks'. Although the program was not set up in a way to tell us how many iterations it took to reach this stage, this could easily be implemented. With some text, as we shall see, the program often managed to obtain an error rate of less than 10%, but 'got stuck' at this point, meaning that it did not satisfy the strict criterion for convergence.

The results are displayed below. The following table identifies the proportion of times 99% or 90% accuracy was reached for each novel. It also gives the mean number of iterations the chain took to reach convergence, given that it did indeed reach the 99% level:

%'ge accuracy	*Kendall*	*Dickens*	*Brontë*	*Stevenson*
99%	63%	67%	28%	68%
90%	63%	72%	68%	68%
Mean no. of iterations	20580	42666	31095	23211
Standard deviation	9195	34718	28491	11652

Leaving the results for *Brontë* aside for a second, the program seems to have a success rate of approximately 66% at reaching equilibrium within $200,000$

iterations. It would also appear that in the cases when it exceeds the 90% accuracy mark, it nearly always converges. Moreover, the mean number of iterations taken for each novel appears surprisingly low compared to the bound imposed: to investigate this further, I produced a box-plot of the number of iterations the chains took to converge for each novel (Figure 5.5).

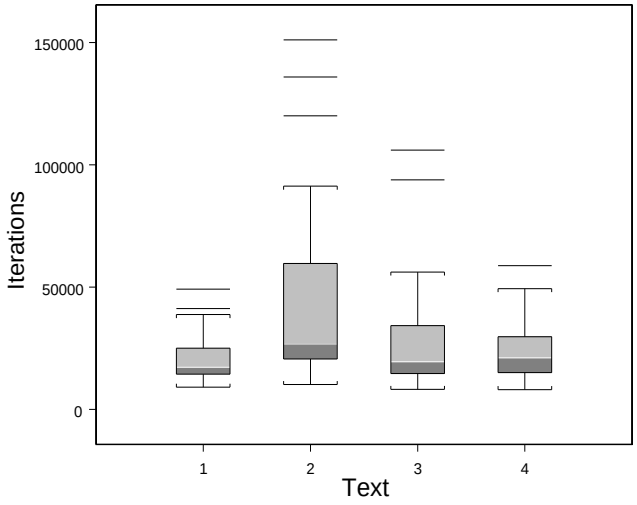

Figure 5.5: Number of iterations to reach convergence for:
1.*Kendall* 2.*Dickens* 3.*Brontë* 4.*Stevenson*

From this it is evident that most of the 66% of chains that converge before $200,000$ iterations actually do so before about $60,000$. This is clearly quite surprising: it appears that the chains either converge quite quickly or do not manage this in the time allowed. The theory states that any chain should eventually converge, and so we would probably expect to see more chains converging between 100 and 200 thousand iterations. If this is really the case, and the number of chains that reach equilibrium after $60,00$ iterations is so small, then this is both good and bad news. It is helpful to us in that it indicates that it is not worth running any chain for more than, say, $70,000$ iterations: if convergence has not been reached then we are more likely to obtain a good decoding by running the program again, rather than continuing with the current chain.

Of course, it is also bad news as it poses the question of why such a small number of chains converge after this cut-off point (assuming that the program works). This is quite difficult to answer, since the permutation space that we are using is so vast and complicated. Ideally, it would be helpful if we could produce a 3-D model of the 'likelihood-surface': imagine the permutations lying

on a plane, in a way such that permutations that differ by a single transposition are 'next to each other'. Now add a third dimension corresponding to the log-likelihood of the permutations. So the permutations with the highest likelihood (i.e. the lowest negative log-likelihood) appear as peaks, whilst those of low likelihood are troughs. This is the 'surface' that the Markov chain traverses in its search for the correct peak: meta-stability occurs when it finds a certain local maximum/minimum on the surface and gets stuck there. Of course, it is also possible for a permutation to have a higher likelihood than the target permutation! Remember that, although the chain will always accept a move that increases its likelihood, it can also accept moves in the opposite direction by construction of the acceptance probability in the Metropolis-Hastings algorithm. It is clearly desirable for the chain to be able to move in this direction, else it could easily reach a local maximum on the likelihood surface from which there would be no chance of progressing to the required permutation. If we could do this, then to understand why the chains do not converge after $60,000$ iterations, we would simply need to plot the paths of such chains on this surface and see what prevents them from converging. Of course, such a model is impossible to construct, but the mental image is useful when trying to understand the behaviour of the chain.

From the graphs produced in the output of the program, it would appear that the chains that do not converge get stuck at certain permutations extremely early on, and then move very little, if at all, from these points. This then is a description of what happens, but it does not really explain why it occurs: the chain should always converge, and so should be able to dig itself out of any troughs encountered. In an attempt to understand why a chain may experience this lengthy period of meta-stability, I modified the program to produce a histogram of the character frequencies of the final permutation if the chain did not converge. The results were not conclusive, but the histograms produced were quite similar to those in Figure 4.1. The only difference was that the peaks corresponding to the lower-case vowels and the 'space' character were shuffled. Hence the program had in almost all cases probably found a permutation which had similar character transition frequencies to those in *War and Peace*: this did not class as convergence, however, because the percentage accuracy of any permutation that does not correctly decode the most frequent characters is extremely low by design. So even if the chain had reached a permutation that decoded the text perfectly apart from the characters 'space' and 'e', then it may experience meta-stability there, since nearly all transpositions proposed would increase the negative log-likelihood. But the program would not term this as convergence, even though it would be quite easy to finish the decoding by hand.

Indeed, if we now look back at the results obtained from deciphering *Brontë*, we see that 40% of the runs reached at least 90% accuracy without reaching the 99% target. Upon closer inspection of this 40%, it was observed that *all* of these had reached a percentage accuracy of 98%. This is strong evidence for other local maxima on the likelihood surface to which these chains converged. Indeed, it would appear that there are at least two different areas on the surface in

which these chains stabilized. This observation arises from consideration of the negative log-likelihood of the final states of the unfinished chains that reached 90%. Half of these readings are between $28,257$ and $28,259$, and the other half lie between $28,140$ and $28,145$. It is not the case, of course, that if two permutations have equal negative log-likelihoods then they are the same permutation. However, the fact that the readings from so many chains lie so close together is, I believe, evidence for two local maxima on the likelihood surface. The values of the negative log-likelihood for those chains that *did* converge lie in the range $15,598$ to $28,366$, with nearly all of them being clustered about the value $28,200$. Hence the final negative log-likelihood values of the chains that reached 98% accuracy are actually lower than some of those that converged. They are also lower than the true value of the first chapter of *Brontë*: $28,281$.

This is a good example therefore of the complexity of the likelihood surface, and of the problems associated with using percentage accuracy to diagnose convergence. We have seen that it is possible for a chain to get stuck at a permutation that has a low negative log-likelihood but whose percentage error is slightly too high to diagnose equilibrium. Of course, we could simply relax the percentage accuracy bound, but problems of this kind will always be found near to the designated cut-off value. The above results also cast some doubt on the usefulness of the negative log-likelihood as a measure of convergence, since it may be possible to find a permutation with a high likelihood (indeed, higher than that of the target permutation) but lower than 99% accuracy. This will be discussed further at the end of the next subsection, at which point there will be more information available to us.

5.4 Starting State

In the discussion about meta-stability in Section 3.3 it was mentioned that one way of trying to avoid such a phenomenon is to simulate a few Markov chains with different starting states. This should help to identify any initialization bias caused by the choice of the initial state. With this in mind I next decided to investigate the effect of the starting state on the time taken to reach convergence.

The final program, `solvitbound`, starts from the identity permutation: this means that, in a real substitution code, it is likely to have a high negative log-likelihood. An obvious way to try to speed up the rate of convergence is to start the chain from a better starting state (although this will not necessarily ensure convergence, since this state may be one of the local maxima described above). This also assumes that a method is available to choose such a starting state. There is, however, one easy way of doing this: simply match up the character frequencies in T with those in *War and Peace* (i.e. replace the most frequent character in T with 'space', the next most frequent with 'e' etc). The program `starter` returns the permutation that, when applied to T, achieves just this. Of course, there are many states that we could choose from when searching for a good starting permutation. The advantage of this choice is, of course, that it can be used in real situations - knowledge of the decoded text is not needed.

I hence proceeded to decode the same four novels as earlier, starting from

two different states. The first of these was the 'intelligent' permutation described above, and the second was simply the identity permutation as normal. Having fixed the random seed, I used `solvitbound` to decode the two versions 20 times for each novel (with an upper bound of 50,000 iterations). An example of the output from this program is included in Appendix **??** (the graphs are: negative log-likelihood vs iterations; percentage accuracy vs iterations; negative log-likelihood vs percentage accuracy).

The first observation to report is that using the intelligent permutation dramatically increases the percentage accuracy of the starting state: an average of 54% for these four novels. Figure 5.6 shows the percentage of runs attaining the 90% and 99% levels of accuracy for each starting state ('both' means that both of the chains produced a decoding to the relevant level of accuracy):

	Kendall		*Dickens*		*Brontë*		*Stevenson*	
	90%	99%	90%	99%	90%	99%	90%	99%
Intelligent	100	100	100	90	100	40	95	95
Identity	60	60	65	35	60	20	75	75
Both	60	60	65	35	60	10	75	75

Figure 5.6: Percentage of runs that reached 90% or 99% accuracy with different starting states

From these results it appears that using the intelligent permutation increases the chance of the chain reaching equilibrium. An improvement was seen at both the 90 and 99% levels for all the texts, and some of these were very significant (for example, the number of runs of *Dickens* reaching 99% accuracy). In total, 98.75% of the intelligent chains reached the 90% level, and 81.25% the 99% level, compared to just 65% and 47.5% of the identity chains respectively. It is also the case that, with the exception of *Brontë*, there were no instances where the intelligent chain did not perform as well as its opponent. With *Brontë* there were two instances where this was not the case, but we have already seen that the likelihood surface for this text is quite complicated: in all cases the intelligent chain reached the 90% level, and in fact all of those that did not reach 99% finished at the 98% level again.

It therefore looks as though choosing a good starting state can increase the frequency of runs that produce good decodings. It remains to be seen, however, whether or not this method actually decreases the number of iterations required for the chain to reach equilibrium. Figure 5.7 is a box-plot of the number of iterations taken by the two types of chain to reach equilibrium for the four texts. It can be seen that the mean number of iterations for each novel, with the exception of *Brontë* as usual, decreased when using the intelligent chain. However, since only four of the identity chains for *Brontë* actually decoded, no real weight can be given to this result. (The other plots were all produced from samples of a much better size.) The weighted mean number of iterations for

the four texts were as follows: 21,790 for the identity chain, and 15,462 for the intelligent chain. This is a reduction of almost 30%, and so it would appear that using an intelligent starting permutation does indeed decrease the time taken to equilibrium.

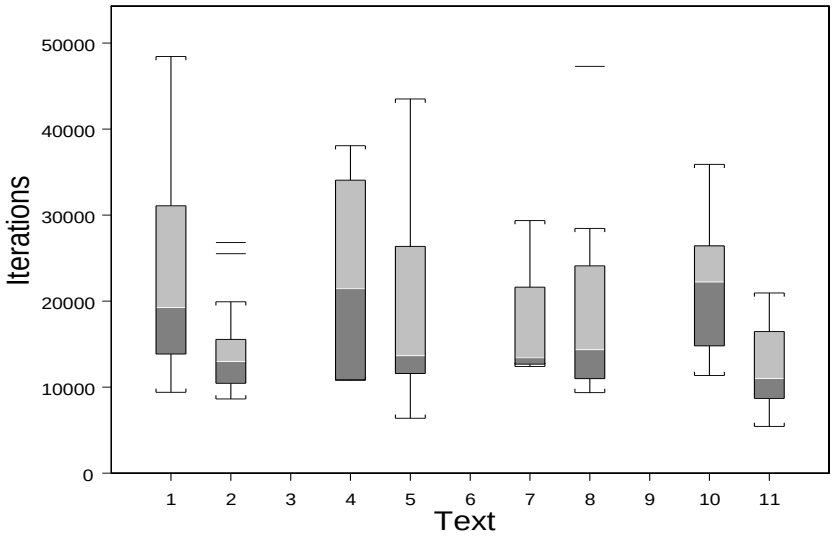

Figure 5.7: Number of iterations to reach convergence for:
1.*Kendall* (identity) 2.*Kendall* (intelligent) 4.*Dickens* (identity)
5.*Dickens* (intelligent) 7.*Brontë* (identity) 8.*Brontë* (intelligent)
10.*Stevenson* (identity) 11.*Stevenson* (intelligent).

With the output from `solvitbound` in this section, it is now appropriate to draw to a conclusion the ongoing debate about how best to diagnose convergence. So far we have seen that measuring the percentage error is an accurate way of predicting convergence. The downsides are that, firstly, the Markov chain can often get stuck at a level below this: although this may be good enough to enable us to finish the decoding by hand, the program will not terminate. Secondly, and more importantly, this diagnostic measure is absolutely useless if we are unaware of the correctly deciphered text from the start! Its use is therefore limited to helping to debug programs.

On the other hand we have the negative log-likelihood method. It was first observed that, due to the consistency of character transitions in English novels, the target negative log-likelihood (of the deciphered text) can be calculated before running the program: all that is required is the length of T. This is clearly an advantage over the percentage accuracy method. However, so far we have

been unconvinced about its efficiency: we have seen examples of permutations which have a lower negative log-likelihood than the target permutation, but a lower percentage accuracy. However, the output of the program `solvitbound` produces a plot of the negative log-likelihood vs the percentage accuracy of the chain. Every such graph produced by this program (given that the chain converged) shows a linear relationship: see the example in Appendix **??** for an example. This is not a perfect relationship, but then we do not need it to be: the important point is that the negative log-likelihood decreases steadily as the percentage accuracy increases. As this is the case, all that is needed is to make the program terminate when the negative log-likelihood of the chain is close enough to that of the target permutation. If the chain stops at a permutation with lower negative log-likelihood than is required this does not matter: experience shows that these permutations are usually incorrect only in a few entries (typically some upper-case letters and punctuation). Using the negative log-likelihood method also means that the program will diagnose convergence in those cases where the chain stops at a level slightly below 99% (as was so often the case when decoding *Brontë*). This should decrease run-time in many cases, and will increase the frequency of runs that are classified as having reached equilibrium.

5.5 Other Text

In Section 4 I presented my reasons for using my program to decode English novels only. In this, I made a brief comparison between such texts and some other literature whose character frequencies and negative log-likelihood values marked them out as being quite different. In this section I briefly present the outcome of attempting to use my program to decode some of these texts, in order to see if my original assertions about its applicability in such circumstances were justified.

I hence used the program to decode the *Welsh*, *Dutch* and *Biochemistry* texts. This was done without the use of an intelligent starting state, and with an upper bound of $50,000$ iterations for each run: 20 runs were carried out for each text.

The results are quite conclusive: not one run achieved the 99% accuracy level. This is not too surprising, since we would hardly expect any of these texts to be that similar to *War and Peace*. Figure 5.8 shows the range of percentage errors produced for these texts. From this we can see a high level of error across the board. As we could have predicted, the text that was best-decoded was the biochemical journal: here the hardest thing that the program was faced with was DNA strings and a few acronyms. This was the only text to have been deciphered to more than 70% accuracy: the Dutch and Welsh texts performed very poorly. A better result may have been achieved by using the 'intelligent' chains discussed earlier, but there was unfortunately not time to implement this. It is uncertain how much of a difference it would have made however: the mean percentage accuracy of the intelligent starting state for these texts was just 34%, compared to an average of 50% for the nine English novels.

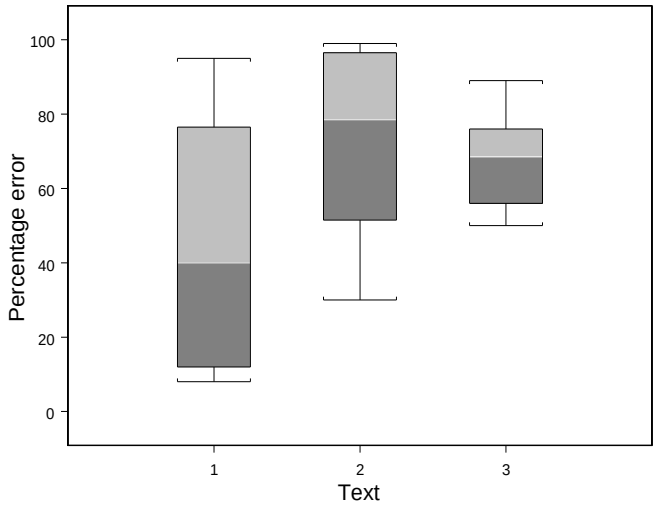

Figure 5.8: Percentage error for: 1.*Biochemistry* 2.*Dutch* 3.*Welsh*

It must also be remembered that the program is not designed to decode such texts, since it uses *War and Peace* as its basis. These results are therefore in keeping with my preliminary analysis, and emphasize the necessity of choosing an appropriate basis for the matrix $(log\, \mathcal{P})$.

5.6 Further Issues

There are clearly many issues remaining in this area: I have only managed to scratch the surface in the permitted time. Below I outline a few directions in which the work in this project could be continued.

Increasing the speed of convergence

I have already described some methods of decreasing the number of iterations taken to reach equilibrium: splitting the text into two and using the intelligent starting state to name but two. There are undoubtedly more methods available, but to research any of these in depth (and to produce good sample sizes) it would be necessary to implement my programs in a faster language than `Mathematica`. I originally intended to re-write my programs in `Java`, but found the fine-tuning of the code in Appendix A too time-consuming to enable me to do this.

Second-order chains

The algorithm employed by my program is based upon character transition frequencies, where I assumed that the English language could be modelled as a simple Markov chain. A better approximation to English may be obtained by extending this model using N-grams. I have already mentioned in Section 1.2 the work of Shannon and Weaver [26]: this shows just how much difference a progression to a higher order model may make.

Inspired by this, I wrote adapted versions of my programs which were designed to use trigrams (i.e. a second-order chain). Unfortunately, this involves creating a new matrix ($log\ \mathcal{P}$) from *War and Peace*. This matrix now has 79^4 entries, which produces severe implementational issues: `Mathematica` was very reluctant to work with such large matrices! Again, this would be more practical if working with a faster programming language.

Reducing the character set *Alph*

Another method of improving the decoding may be to work with a reduced character set. The reasoning behind this is the fact that many of the entries in the original character transition frequency matrix obtained from *War and Peace* are very small, or even zero. Hence there is not much information regarding transitions between these characters. For example, the sequence 'Az' may not be used at all by Tolstoy, but it may form part of the text we are trying to decode. If the entry in the aforementioned matrix corresponding to this sequence is zero, then the program will have trouble accurately decoding this part of the text.

One way of reducing this problem is to combine certain entries in the matrix obtained from the basis text, in such a way that the amount of information available is increased. For example, with the exception of the characters 'space', 'comma' and 'full-stop', punctuation tends to be fairly infrequent in most texts, compared to the number of other characters. This suggests combining little-used punctuation into a 'wild character'. The transition probabilities for this character could then be arranged so as to incorporate all of the information arising from each punctuation mark being mapped to it. An extension of this would be to combine the upper and lower-case letters into one. Upper-case letters are also quite rare, and so the program has real difficulty in decoding them. If the two cases were combined more information would be available to the program, provided by the lower-case letters.

A program of this sort would run according to the following algorithm:

1. propose a permutation (via transpositions as usual)
2. introduce the wild character, and combine upper/lower cases
3. calculate the acceptance probability using the reduced character set matrix
4. move the chain if appropriate, and return to stage 1.

I attempted to implement such an algorithm, but was quite unsuccessful. The main problem was that, after I had combined the transition frequencies

for the punctuation marks into one matrix entry for the wild character, this character became too popular! The relatively high frequency with which one punctuation mark follows another in English (that is, *any* mark follows *any* other) meant that the output from the program consisted of a string of almost entirely wild characters. This idea has merit, however, since in theory it should help to produce a more accurate decoding. In fact, I have recently heard from Diaconis himself that one of his Ph.D students has just tackled this problem in his thesis. Apparently quite a lot of thoughtful work is needed to adjust the transition frequency of the wild character to avoid the problems that I encountered.

6 Coupling

Having considered the number of iterations my program takes to converge, I now present two methods for calculating a theoretical bound on this figure. In this section, I use coupling theory and strong uniform times, and in the following section an approach involving representation theory is given. Both methods are based upon the analysis of shuffling a pack of n cards. The shuffling construction is different in each section, but both use random transpositions of cards and consider the time taken until the pack is 'well shuffled'. The ways in which transpositions of cards are proposed and accepted is much simpler than in the Metropolis-Hastings algorithm: in fact, no information about the current state of the cards is used in determining which cards are swapped. Nevertheless, this theoretical approach is extremely interesting, and provides some very useful information about the convergence rate of the Markov chains produced by the program.

Before considering the shuffling construction for the coupling method, it is first necessary to introduce the language of random walks and probabilities on groups.

Definition 6.1. A *probability* Q on G assigns a real number to each element of G, such that $Q(g) \geq 0$ for all g, and $\sum_g Q(g) = 1$.

The *Uniform distribution* on the finite group G is the probability that assigns the value $\frac{1}{|G|}$ to each element g in G.

Suppose now that we wish to construct a random walk on G, where Q is a probability on G. If independent choices are made from G, according to Q, then this yields elements $\zeta_1, \zeta_2, \ldots \in G$. We can then proceed to define the products

$$\begin{aligned} X_0 &= id \\ X_1 &= \zeta_1 \\ X_2 &= \zeta_2\zeta_1 \\ \vdots &\quad \vdots \\ X_k &= \zeta_k X_{k-1} = \zeta_k\zeta_{k-1}\ldots\zeta_1. \end{aligned}$$

It follows that $\{X_k\}_{k\geq 0}$ is a random walk on G with step distribution Q. Now, note that:

$$\mathbb{P}(X_2 = g) = Q^*Q(g) = \sum_{h\in G} Q(h)Q(gh^{-1}),$$

since $Q(h)Q(gh^{-1})$ is the probability that element h is picked first, and gh^{-1} picked second, resulting in the product g. Induction on this argument leads to:

$$\mathbb{P}(X_k = g) = Q^{*k}(g) = Q * Q^{*(k-1)}(g) = \sum_{h\in G} Q(h)Q^{*(k-1)}(gh^{-1}).$$

Definition 6.2. If P and Q are two probabilities on G, then the *convolution* $P * Q$ is the probability $P * Q(s) = \sum_t P(st^{-1})Q(t)$.

Now, in these card-shuffling setups, we are interested in the number of transpositions needed to get the pack 'close to random'. Before we can progress any further, it is hence necessary to decide exactly what we mean by this. To do this we first require the following definition:

Definition 6.3. Let G be a finite group, and let P and Q be probability distributions on G. Define the *variation distance* between P and Q as

$$\|P - Q\| = \max_{A\in G} |P(A) - Q(A)|$$

This definition provides us with a metric structure that enables us to compare the distance between probabilities on G, and U, the uniform distribution. It makes sense to use the uniform distribution as the probability on a well-shuffled pack, since this assigns an equal weight to each permutation of the n cards. Say that the smaller the variation distance from uniform, the 'closer the pack is to random'.

I said at the start of this section that the first method of finding an upper bound on the number of iterations would involve coupling and strong uniform times, so I had better explain what I mean by this. Coupling is an extremely useful method with a wide range of applications in Markov processes. First proposed by Doeblin in 1938, it enables us to assess the convergence of a Markov chain by comparing it with another chain. For our purpose, we wish to assess the convergence of P^{*k} (for some distribution P describing the shuffling method), and have decided to do this by comparing it with the uniform distribution, U. The exact method of doing this is presented shortly, but first we need the notion of a strong uniform time:

Definition 6.4. A *stopping time* is formally a function $T : G^\infty \to \{1, 2, \ldots, \infty\}$ such that if $T(\underline{s}) = j$, then $T(\underline{s}') = j$ for all $\underline{s}'$ with $s'_i = s_i$ for $1 \leq i \leq j$. At a more intuitive level, a stopping time considers a sequence of elements in G and says 'stop at the j^{th} one': this rule is allowed to depend on what happens up to time j, but not on any events further into the future.

A *strong uniform time* is a stopping time T such that for each $k < \infty$,

$$\mathbb{P}(T = k,\ X_k = s) \text{ is constant in s}$$

where $\{X_j\}_{j\geq 0}$ is the observed Markov chain.

Note that, if T is a strong uniform time, the above definition implies that

$$\mathbb{P}(X_k = s \mid k \geq T) = \mathbb{P}(X_k = s \mid k = T) = \frac{1}{|G|}.$$

The following lemma will be instrumental in this section in the attempt to bound the number of shuffles needed to approximate a uniform distribution. It provides a useful connection between strong uniform times and the distance in variation of two distributions.

Lemma 6.1. *Let Q be a probability on the finite group G. Let T be a strong uniform time for Q. Then, for all $k \geq 0$,*

$$\left\|Q^{*k} - U\right\| \leq \mathbb{P}(T > k)$$

where U is the uniform distribution on G.

Proof. For any $A \subset G$:

$$\begin{aligned} Q^{*k}(A) &= \mathbb{P}(X_k \in A) \\ &= \sum_{j \leq k} \mathbb{P}(X_k \in A,\ T = j) + \mathbb{P}(X_k \in A,\ T > k) \\ &= \sum_{j \leq k} \mathbb{P}(X_k \in A \mid T = j)\,\mathbb{P}(T = j) + \mathbb{P}(X_k \in A \mid T > k)\,\mathbb{P}(T > k). \end{aligned}$$

Now, $\mathbb{P}(X_k \in A \mid T = j) = U(A)$, for $j \leq k$, since T is a strong uniform time. So we have,

$$\begin{aligned} Q^{*k}(A) &= \sum_{j \leq k} U(A)\mathbb{P}(T = j) + \mathbb{P}(X_k \in A \mid T > k)\,\mathbb{P}(T > k) \\ &= U(A) + [\mathbb{P}(X_k \in A \mid T > k) - U(A)]\,\mathbb{P}(T > k) \\ \\ &\Rightarrow \left|Q^{*k}(A) - U(A)\right| \leq \mathbb{P}(T > k). \end{aligned}$$

□

I now describe in detail the shuffling method that will be used to bound the convergence rate. It uses random transpositions and the notion of 'checking' certain cards as they are selected and swapped. The particular construction upon which I shall concentrate is due to Andre Broder.

6.1 Shuffling Construction

Picture laying out n cards in a row on a table. At each stage we randomly and independently select a card with each hand, giving rise to a pair (L_i, R_i), which are swapped.

If either

1. both hands touch the same unchecked card, or

2. L_i is unchecked, but R_i is checked,

then the card touched by the left hand, L_i, is checked. The shuffling process stops at time T when all the cards are checked.

The aim is now to prove that T is a strong uniform time, so that it can be used in Lemma 6.1 to bound the time taken to get close to the uniform distribution. First note that T is clearly a stopping time, since it depends purely on the card transpositions taken for all the cards to get checked, and not upon what occurs after this point. This reduces the problem to showing that at time T the ordering of the cards is completely random.

Let us first consider the process on an informal basis, before turning to a formal proof. The process of checking begins when both hands touch the same unchecked card. This card (call it card 1) is checked. Nothing happens then until one of two things happens: either both hands hit a different card to card 1 (say card 2), or the left hand hits an unchecked card (say 2) and the right hand hits card 1 - these are then swapped. At this stage, conditional upon the positions of the two checked cards in the line, a_1 and a_2 say, and the labels 1 and 2, the positions are clearly equally likely to correspond to $(a_1, 1)(a_2, 2)$ or $(a_1, 2)(a_2, 1)$. This follows since the chance of choosing card 2 is $\frac{1}{n^2}$ in either case.

Before attempting a formal proof of the uniformity of the cards once all have been checked, I first take a moment to establish notation. In general, the position we are in at any time may be described as follows:

$$\{L, \{a_1, \ldots, a_L\}, \{c_1, \ldots, c_L\}, \Pi_L\}.$$

Where,

$$\begin{aligned} L &= \text{number of checked cards} \\ \{a_1, \ldots, a_L\} &= \text{set of positions of the checked cards} \\ \{c_1, \ldots, c_L\} &= \text{labels of the checked cards} \\ \Pi_L : \{a_1, \ldots, a_L\} &\to \{c_1, \ldots, c_L\} \text{ records the card at each position.} \end{aligned}$$

It is extremely important to ensure full comprehension of this notation before reading on. The main point to appreciate is that once L cards have been checked, the sets $\underline{a}_L = \{a_1, \ldots, a_L\}$ and $\underline{c}_L = \{c_1, \ldots, c_L\}$ are fixed: Π_L is a permutation, telling us how these checked cards are arranged amongst themselves. So if I write $\Pi_L(\{a_1, \ldots, a_L\}) = (\gamma_1, \ldots, \gamma_L)$, then I mean that card γ_1 is in position a_1, γ_2 in a_2 etc, where $\underline{\gamma}_L = (\gamma_1, \ldots, \gamma_L)$ is a distinct ordering of the cards $\{c_1, \ldots, c_L\}$.

The fact that T is a strong uniform time will follow from the following proposition.

Proposition 6.2. *At each time, conditional on L, $\{a_1, \ldots, a_L\}$, and $\{c_1, \ldots, c_L\}$, the permutation Π_L is uniform.*

Proof. This is proved by induction on L. For $k \geq 0$, define B_k to be the event $\{L = k$, positions are $\{a_1, \ldots, a_k\}$, labels are $\{c_1, \ldots, c_k\}\}$. We need to show that, for fixed $L = k$,

$$\mathbb{P}\left(\Pi_k\left(\{a_1, \ldots, a_k\}\right) = \underline{\gamma}_k \mid B_k\right) = \frac{1}{k!} \tag{6.1}$$

We have:

- $L = 0,\ \Pi_0 : \{\phi\} \to \{\phi\}$
- $L = 1,\ \Pi_1 : \{a_1\} \to \{c_1\}$

and in these cases it is clear that 6.1 is satisfied, since there is only one possible permutation in each case.

Assume now that the equation (6.1) holds true for $L = k$. Consider again the shuffling setup: if we are at a time when B_k holds, then the permutation remains uniform until a new card (c_{k+1}) is checked, and event B_{k+1} occurs. This happens when the left hand (LH) touches c_{k+1} and the right hand (RH) touches one of $\{c_1, \ldots, c_{k+1}\}$. The right hand is equally likely to touch any one of this set of $k+1$ cards, and so we have

$$\begin{aligned}\mathbb{P}\left(\text{new card checked} \mid B_k\right) &= \mathbb{P}\left(\text{LH} = c_{k+1},\ \text{RH} \in \{c_1, \ldots, c_{k+1}\} \mid B_k\right) \\ &= \frac{k+1}{n^2}. \end{aligned} \tag{6.2}$$

Now,

$$\begin{aligned}&\mathbb{P}\left(\Pi_{k+1}\left(\underline{a}_{k+1}\right) = \underline{\gamma}_{k+1} \mid B_{k+1}\right) \\ &= \mathbb{P}\left(\Pi_{k+1}(\{a_j\}) = c_{k+1},\ \Pi_{k+1}\left(\underline{a}_{k+1} \setminus \{a_j\}\right) = \underline{\gamma}_{k+1} \setminus \{c_{k+1}\} \mid B_{k+1}\right) \\ &\qquad\qquad \text{for some } j \in \{1, \ldots, k+1\} \\ &\text{(simply looking at where } \Pi_{k+1} \text{ maps card } c_{k+1}) \\ &= \mathbb{P}\left(\Pi_{k+1}\left(\underline{a}_{k+1} \setminus \{a_j\}\right) = \underline{\gamma}_{k+1} \setminus \{c_{k+1}\} \mid \Pi_{k+1}(a_j) = c_{k+1},\ B_{k+1}\right) \\ &\qquad\qquad \times \mathbb{P}\left(\Pi_{k+1}(\{a_j\}) = c_{k+1} \mid B_{k+1}\right) \\ &\text{(conditioning on the position of the new card, } c_{k+1}) \\ &= \mathbb{P}\left(\Pi_{k+1}\left(\underline{a}_{k+1} \setminus \{a_j\}\right) = \underline{\gamma}_{k+1} \setminus \{c_{k+1}\} \mid B_k\right) \\ &\qquad\qquad \times \mathbb{P}\left(\Pi_{k+1}(\{a_j\}) = c_{k+1} \mid B_{k+1}\right) \\ &\text{(once we know where } c_{k+1} \text{ is, we just need to consider the permutation} \\ &\quad \text{of the other } k \text{ cards}) \\ &= \frac{1}{k!}\,\mathbb{P}\left(\Pi_{k+1}(\{a_j\}) = c_{k+1} \mid B_{k+1}\right) \quad \text{(by the inductive hypothesis)} \\ &= \frac{1}{k!}\,\frac{\mathbb{P}\left(\text{RH} = \Pi_k(\{a_j\}),\ \text{LH} = c_{k+1} \mid B_k\right)}{\mathbb{P}\left(\text{a new card is checked} \mid B_k\right)}, \quad \text{where } \Pi_k(\{a_{k+1}\}) := c_{k+1} \end{aligned}$$

(This step follows from the shuffling setup. Imagine that we are at a stage where B_k holds. The probability of the event $\{c_{k+1}$ is checked and Π_{k+1} maps position a_j to card $c_{k+1}\}$, equals the probability that LH touches c_{k+1} and RH touches the card that was in position a_j, divided by the probability that we actually check a new card, given B_k holds.)

$$\begin{aligned} &= \frac{1}{k!}\left(\frac{1}{n^2}\right)\left(\frac{k+1}{n^2}\right)^{-1} \quad \text{(from equation (6.2))} \\ &= \frac{1}{(k+1)\,!} \quad \text{as required.} \end{aligned}$$

□

We have therefore shown that T is indeed a strong uniform time, meaning that we can use it in Lemma 6.1.

6.2 Convergence Rates

Now, the strong uniform time T can be written in the following way:

$$T = \sum_{i=0}^{n-1} \left(T_{i+1,n} - T_{i,n}\right),$$

where $T_{i,n}$ is the number of transpositions until i cards are checked, for a fixed total of n cards. Clearly the random variables $(T_{i+1,n} - T_{i,n})$ are independent. They also each follow a Geometric$(p_{i,n})$ distribution, by the independence of transpositions, where

$$\begin{aligned} p_{i,n} &= \mathbb{P}\left(\text{new card checked} \mid i \text{ cards checked already}\right) \\ &= \mathbb{P}\left(\text{LH touches one of } (n-i) \text{ unchecked cards}\right) \\ &\qquad \times \mathbb{P}\left(\text{RH touches one of } i \text{ checked, or RH=LH}\right) \\ &= \frac{(n-i)(i+1)}{n^2}. \end{aligned}$$

It follows that the mean of $(T_{i+1,n} - T_{i,n})$ is $n^2/[(n-i)(i+1)]$.

Thus

$$\begin{aligned}
\mathbb{E}[T] &= \sum_{i=0}^{n-1} \frac{n^2}{(i+1)(n-i)} \qquad (6.3)\\
&= \left(\frac{n^2}{n+1}\right) \sum_{i=0}^{n-1} \left(\frac{1}{i+1} + \frac{1}{n-i}\right) \quad \text{(partial fractions)}\\
&= \left(\frac{2n^2}{n+1}\right) \left(1 + \frac{1}{2} + \ldots + \frac{1}{n}\right)\\
&= \left(\frac{2n}{n+1}\right) (n\ log\ n - n\ r_n) \quad \text{where } \lim_{n\to\infty} r_n = \gamma \text{ (Euler's constant [6].)}\\
&= n\ log\ n \left(2 - \frac{2}{n+1} - \frac{2n}{(n+1)}\ \frac{r_n}{log\ n}\right)\\
&= n\ log\ n \left(2 - \frac{1}{log\ n} \left(\frac{2n}{n+1} r_n - \frac{2\ log\ n}{n+1}\right)\right).
\end{aligned}$$

Now, setting $k_n = \frac{1}{log\ n} \left(\frac{2n}{n+1} r_n - \frac{2\ log\ n}{n+1}\right)$, we have:

$$\begin{aligned}
&|k_n\ log\ n| \leq c \quad as\ n \to \infty \quad \text{for some constant c}\\
&\Rightarrow k_n = O\left(\frac{1}{log\ n}\right)
\end{aligned}$$

Hence,

$$\mathbb{E}[T] = \left(2 + O\left(\frac{1}{log\ n}\right)\right) n\ log\ n.$$

Note that this varies from the account given by Diaconis in [8]: this gives $O\left(\frac{1}{n}\right)$ where in the above I have $O\left(\frac{1}{log\ n}\right)$. This does not affect the convergence rate calculations explicitly: it is merely pointed out here for completeness.

We also have:

$$\begin{aligned}
Var(T) &= \sum_{i=0}^{n-1} Var\left(T_{i+1,n} - T_{i,n}\right) \quad \text{(by independence)}\\
&= n^2 \sum_{i=0}^{n-1} \frac{n^2 - (i+1)(n-i)}{(i+1)^2(n-i)^2} \quad \text{since variance of a Geometric}(p_{i,n}) = \frac{1-p_{i,n}}{p_{i,n}^2}\\
&= \frac{n^4}{(n+1)^2} \sum_{i=0}^{n-1} \frac{2}{(i+1)^2} \ + \ \frac{2n^2(n^2-2n-1)}{(n+1)^3} \sum_{i=0}^{n-1} \frac{1}{(i+1)}\\
&= n^2 \left[\left(\frac{n}{n+1}\right)^2 \sum_{k=1}^{n} \frac{2}{k^2} \ + \ \left(\frac{n^2-2n+1}{(n+1)^3}\right) \sum_{k=1}^{n} \frac{2}{k}\right]\\
&= O(n^2) \quad \text{since both sums in the brackets converge as } n \to \infty.
\end{aligned}$$

In order to determine the long-run behaviour of this method of shuffling, it is necessary to consider the sum of the independent random variables $X_{i,n} := (T_{i+1,n} - T_{i,n})$. An obvious method of doing this is to look to the central limit theorem (CLT) for help. The two versions of the CLT that I attempted to use are presented below in Theorems 6.3 and 6.4.

Theorem 6.3 (Central Limit Theorem - Lyapunov). *Let $X_1, X_2, \ldots$ be independent variables satisfying*

$$\mathbb{E}[X_j] = 0, \quad Var(X_j) = \sigma_j^2, \quad \mathbb{E}[X_j^3] < \infty$$

and such that

$$\frac{1}{\sigma(n)^3} \sum_{j=1}^{n} \mathbb{E}[X_j^3] \to 0 \quad as \ n \to \infty \tag{6.4}$$

where

$$\sigma(n)^2 = Var\left(\sum_{1}^{n} X_j\right) = \sum_{1}^{n} \sigma_j^2.$$

Then

$$\frac{1}{\sigma(n)} \sum_{1}^{n} X_j \to N(0,1).$$

Theorem 6.4 (Central Limit Theorem - Lindeberg). *Let $X_1, X_2, \ldots$ be mutually independent one-dimensional random variables with distributions $F_1, F_2, \ldots$ such that*

$$\mathbb{E}[X_k] = 0, \quad Var(X_k) = \sigma_k^2,$$

and put

$$s_n^2 = \sigma_1^2 + \ldots + \sigma_n^2.$$

Assume that for each $t > 0$

$$s_n^{-2} \sum_{k=1}^{n} \int_{|y| \geq t s_n} y^2 F_k\{dy\} \to 0. \tag{6.5}$$

Then the distribution of the normalized sum

$$S_n^* = (X_1 + \ldots + X_n)/s_n$$

tends to the standard normal distribution.

However, I was unable to make the centred random variables, $Y_i := X_i - \mathbb{E}[X_i]$, satisfy either of equations (6.4) or (6.5). It then came to light that, despite Diaconis [8], the central limit theorem can *not* be applied in this situation.

To show why this is true I turn to the theory of epidemics without removals. Suppose we have a constant population of size n, with one individual infected at time $t = 0$. At time t let X_t be the number of susceptible individuals, and

Y_t the number of individuals infected (so $X_t + Y_t = n$ for all t). Assume that infection is by contact and that the population mixes homogeneously. Assume also that if there are y infected then the rate at which $y \to y+1$ is given by $\alpha y(n+1-y)$ for some $\alpha > 0$.

We wish to study the time taken for all the population to become infected, S. As before, we can write

$$S = \sum_{i=0}^{n-1} (S_{i+1} - S_i),$$

where S_i is the length of time until i individuals are infected. $(S_{i+1} - S_i)$ is exponentially distributed with mean $(\alpha i(n-i))^{-1}$. A calculation similar to that in (6.3) gives

$$\mathbb{E}[S] \approx \frac{2log\ n}{\alpha\ n} \tag{6.6}$$

This is clearly very similar to the card shuffling setup. Here, we are looking for the time at which all individuals have become infected, whereas before we were interested in the time taken for all the cards to get checked. We can hence model the card checking as an epidemic without removals, and this gives $\alpha = n^{-2}$ for the shuffling method described above (by comparison of equations (6.3) and (6.6), setting $S = T$).

Now, it can be shown that the limiting distribution of $(Sn^{-1} - 2\ log\ n)$ exists ([7] pp200, ex.13). We do not need to concern ourselves with what this distribution is (call it X), simply that it exists and is not Normal. This shows that the CLT cannot apply in the card checking situation: I have already shown that

$$\mathbb{E}[T] \approx 2n\ log\ n \quad \text{and} \quad Var(T) = O(n^2).$$

It therefore follows that if the CLT applied, it would be the case that

$$\frac{T - 2n\ log\ n}{n} \sim N(0, \sigma^2) \quad \text{for some } \sigma.$$

This is not the case though, since X is not Normal.

Although I have now significantly wandered from the discussion of Diaconis, it is possible to reach his result of an upper bound. This can be achieved simply by using the fact that the distribution X exists, and that

$$\mathbb{P}(T - 2n\ log\ n > nx) \to \mathbb{P}(X > x).$$

Now, if $k = 2n\ log\ n + c(n)n$, with $c(n) \to \infty$,

$$\left\| P^{*k} - U \right\| \leq c\mathbb{P}(T > k),$$

since T is a strong uniform time.

But,

$$\mathbb{P}(T > k) = \mathbb{P}(T - 2n\ log\ n > c(n)n).$$

It follows that

$$\lim_{n\to\infty} \sup \mathbb{P}\left(T - 2n\ log\ n > c(n)n\right) \leq \mathbb{P}\left(X > x\right) \quad \text{if } c(n) > x \text{ eventually.}$$

Since $c(n) \to \infty$, this is true for all x, and so the limit superior above must be equal to 0.

Therefore, with $k = 2n\ log\ n + c(n)n$, where $c(n) \to \infty$,

$$\left\|P^{*k} - U\right\| \to 0 \qquad \text{as } n \to \infty.$$

These sorts of calculations have been used to solve many similar card-shuffling problems, and have generally proved more useful than the more traditional methods involving the second largest eigenvalue of a chain's transition matrix [4]. The main difference is that these methods enable us to answer questions such as "are k shuffles enough to get close to uniform?", as opposed to the more traditional "how close to uniform is the deck after n shuffles?".

Other shuffling schemes have been studied as well, and similar bounds obtained. For example, the pack could be shuffled by repeatedly removing the top card and inserting it at a random position. Diaconis [1, 8] has shown that, for this setup, $k = \frac{1}{2}n\ log\ n$ shuffles are enough. In all shuffling models that have been studied, there appears to be what Diaconis refers to as a *cut-off phenomenon*: there is some critical number of shuffles, k_0 such that

$$\begin{aligned} &\left\|Q^{*(k_0+o(k_0))} - U\right\| \approx 0 \\ \text{but} \quad &\left\|Q^{*(k_0-o(k_0))} - U\right\| \approx 1. \end{aligned}$$

Thus the pack suddenly becomes well-shuffled at this cut-off point: the variation distance is practically 1 until this number of shuffles, and then becomes very small, before tending to zero exponentially after k_0. This phenomenon is what enables us to produce such sharp bounds as the $2n\ log\ n$ result above.

7 Representation Theory

In this section I present a completely different area of theory: that of group representations. This area has had a huge impact upon calculating convergence rates for card shuffling schemes over the past 20 years. There are many texts devoted to its study, but here I only give a brief introduction to the theory, and work towards another result of Diaconis in [8].

I begin with a few definitions:

Definition 7.1. A *representation* of a group G of degree n is a homomorphism of G into the group $GL(V)$ of linear maps on a vector space V. So to each element $g \in G$ there corresponds a matrix $\rho(g)$, and if $x, y \in G$ we have $\rho(xy) =$

$\rho(x)\rho(y)$.

The *dimension* of ρ is d_ρ, which equals the dimension of V.

The *trivial representation* is 1-dimensional: it maps every group element to the identity.

Definition 7.2. If W is a subspace of V stable under G (i.e. $\rho(g)W \subset W$, for all $g \in G$), then $\rho|_W$ gives a subrepresentation. If ρ has no non-trivial subrepresentation, then ρ is called *irreducible*.

Irreducible representations are the building blocks of the theory, since it can be shown that every representation is a direct sum of irreducible representations. They will prove important later on, as will the notion of the character of a representation:

Definition 7.3. If ρ is a representation, the *character* of ρ, χ_ρ, is defined by

$$\chi_\rho(s) = Tr\left(\rho(s)\right)$$

where Tr is the trace of the matrix. (This is independent of the choice of basis of V, since the trace function is basis-free.)

Throughout the following discussion, the Fourier transform will play an important role, and I now take a moment to explore the link between this and the convolution defined in Definition 6.2.

Definition 7.4. Let P be a probability on G. The *Fourier transform* of P at the representation ρ is the matrix

$$\hat{P}(\rho) = \sum_{s \in G} P(s)\rho(s). \tag{7.1}$$

The Fourier transform and the convolution interact as follows. Let ρ be any representation. We have

$$\widehat{P * Q}(\rho) = \hat{P}(\rho)\hat{Q}(\rho).$$

Proof.

$$\begin{aligned}
\widehat{P * Q}(\rho) &= \sum_s P * Q(s)\rho(s) \\
&= \sum_s \sum_t P(st^{-1})Q(t)\rho(s) \\
&= \sum_s \sum_t P(st^{-1})\rho(st^{-1})Q(t)\rho(t) \quad \text{since } \rho \text{ is a homomorphism} \\
&= \sum_t \left(\sum_s P(st^{-1})\rho(st^{-1}) \right) Q(t)\rho(t) \\
&= \left(\sum_s P(s)\rho(s) \right) \left(\sum_t Q(t)\rho(t) \right) \\
&= \hat{P}(\rho)\hat{Q}(\rho)
\end{aligned}$$

□

I now move onto one of the most important results in the theory of group representations, Schur's Lemma. The proof is not hard [8, 23], but is omitted here for brevity.

Lemma 7.1. *(Schur's Lemma) Let $\rho_1 : G \to GL(V_1)$ and $\rho_2 : G \to GL(V_2)$ be two irreducible representations of G, and let f be a linear map of V_1 into V_2, such that*

$$\rho_2(s) \circ f = f \circ \rho_1(s) \qquad \forall s \in G. \tag{7.2}$$

Then,
1) if ρ_1 and ρ_2 are not equivalent, $f = 0$
2) if $V_1 = V_2$ and $\rho_1 = \rho_2$, $f = \lambda I$, λ a constant.

Corollary 7.2. *Let $V_1 = V_2 = V$, $\rho : G \to GL(V)$ be a representation, and U be the uniform probability on G. Define $f : V \to V$ by $f = \hat{U}(\rho)$. Then*

$$\rho(s) \circ f = f \circ \imath(s) \quad \forall s \in G$$

where $\imath$ is the trivial representation.

Proof.

$$\begin{aligned}
\rho(s) \circ f &= \rho(s) \sum_{t \in G} U(t)\rho(t) \\
&= \sum_{t \in G} \frac{1}{|G|} \rho(s)\rho(t) \\
&= \sum_{t \in G} \frac{1}{|G|} \rho(st) \\
&= \sum_{t \in G} U(t)\rho(t) \quad \text{since } G \text{ is a finite group} \\
&= f \circ \imath(s).
\end{aligned}$$

□

Before proceeding to the main results of this section, I first present a couple of equalities that will be of significant use later on. The first is the observation that $\hat{Q}(\imath) = 1$, when Q is any probability on G. This follows simply from the fact that $\imath(t) = 1$ for all $t \in G$, and that Q is a probability, hence $\sum_t Q(t) = 1$.

Furthermore, the above corollary shows that, when ρ is the trivial representation, the conditions for Schur's Lemma are satisfied by U. So, in the notation of the corollary, $\rho \neq \imath$ implies that $\hat{U}(\rho) = 0$. We hence have:

$$\hat{U}(\rho) = \begin{cases} 1 & \text{for } \rho \text{ trivial} \\ 0 & \text{for } \rho \text{ non-trivial.} \end{cases} \tag{7.3}$$

Recall from 6.3 the definition of $\|Q - P\|$. This will again be the method used to consider how far the cards are from random. However, for its employment in this section it has a much more useful form:

Proposition 7.3.

$$\|Q - P\| = \frac{1}{2} \sum_s |Q(s) - P(s)| .$$

Proof.

$$\begin{aligned} \|Q - P\| &= \max_{A \subset G} |Q(A) - P(A)| \\ &= \sum_{g \in A'} |Q(g) - P(g)| \quad \text{where } A' = \{g \in G : Q(g) > P(g)\}. \end{aligned}$$

But,

$$\sum_{g \in G} |Q(g) - P(g)| = \sum_{g \in A'} |Q(g) - P(g)| + \sum_{g \notin A'} |Q(g) - P(g)| .$$

Clearly, by symmetry, the two sums on the right hand side are equal, hence the result. □

The following theorem, presented here without proof [8] is the last that is needed to be able to really start on the quest to find an upper bound.

Theorem 7.4 (Plancherel Theorem). *Let f and h be functions on G, then*

$$\sum_s f(s^{-1}) h(s) = \frac{1}{|G|} \sum_i d_i Tr\left(\hat{f}(\rho_i)\hat{h}(\rho_i)\right).$$

Note that when f and h are real functions, and the ρ_i are unitary representations, the above can be rewritten as:

$$\sum_s f(s) h(s) = \frac{1}{|G|} \sum_i d_i Tr\left(\hat{h}(\rho_i)\hat{f}(\rho_i)^*\right),$$

where x^* is the conjugate of x.

This theorem, along with Proposition 7.3 enable us to bound the variation distance $\left\|P^{*k} - U\right\|$. This is courtesy of the Upper Bound Lemma, the main result of this section. This inequality was first used in Diaconis and Shahshahani [9]:

Lemma 7.5 (Upper Bound Lemma). *Let Q be a probability on the finite group G. Then*

$$\|Q - U\|^2 \leq \frac{1}{4} \sum\nolimits^* d_\rho Tr\left(\hat{Q}(\rho)\hat{Q}(\rho)^*\right),$$

where the sum ($\sum^*$) is over all non-trivial representations.

Proof. From 7.3,

$$\begin{aligned} 4\left\|Q-U\right\|^2 &= \left\{\sum_s |Q(s)-U(s)|\right\}^2 \\ &\leq |G|\sum |Q(s)-U(s)|^2 \quad \text{by Cauchy-Schwarz} \\ &= |G|\sum f(s)^2 \quad \text{where } f(s)=|Q(s)-U(s)| \\ &= \sum d_\rho Tr\left(\hat{f}(\rho)\hat{f}(\rho)^*\right) \quad \text{by the Plancherel Theorem.} \end{aligned}$$

Now,

$$\begin{aligned} \hat{f}(\rho) &= \left|\hat{Q}(\rho)-\hat{U}(\rho)\right| \\ &= \begin{cases} 0 & \text{for } \rho \text{ trivial} \\ \hat{Q}(\rho) & \text{for } \rho \text{ non-trivial, by equation (7.3).} \end{cases} \end{aligned}$$

Therefore,

$$4\left\|Q-U\right\|^2 \leq \sum^* d_\rho Tr\left(\hat{Q}(\rho)\hat{Q}(\rho)^*\right).$$

□

Now that we have a method of using representations to bound the distance a distribution is from uniform, we need to consider the method of card shuffling to be used. This will then lead to determination of the associated probability distribution, which can then be used in the Upper Bound Lemma to produce the required bound.

In this section a simpler method of shuffling is available to us. Suppose we have n cards laid in a row on a table. Shuffle these in the following manner: the left hand touches a random card, as does the right hand, and the two cards are simply interchanged. This can be modelled this by the following probability distribution on the symmetric group of n elements, S_n:

$$P(\rho) = \begin{cases} \frac{1}{n} & \text{for } \rho = id \\ \frac{2}{n^2} & \text{for } \rho \text{ a transposition (denoted } \tau) \\ 0 & \text{otherwise.} \end{cases}$$

Repeated random transpositions of the n cards can then be modelled as repeatedly convolving the above probability. From this setup it is clear that P is constant on conjugacy classes: $P(\eta\pi\eta^{-1}) = P(\pi)$. Hence, looking to satisfy

the conditions for Schur's Lemma, we see that:

$$\begin{aligned}\rho(s) \circ \hat{P}(\rho) &= \rho(s) \sum P(t)\rho(t) \\ &= \sum P(sts^{-1})\rho(st) \\ &= \left(\sum P(sts^{-1})\rho(sts^{-1})\right)\rho(s) \\ &= \hat{P}(\rho) \circ \rho(s).\end{aligned}$$

Thus Schur's Lemma applies here, and so $\hat{P}(\rho) = \lambda I$ for some constant λ. Taking traces of both sides gives:

$$\begin{aligned}\lambda d_\rho &= \sum_t P(t)\chi_\rho(t) \\ &= \frac{1}{n}d_\rho + \sum_\tau \frac{2}{n^2}\chi_\rho(\tau) \quad \text{since } \chi_\rho(\imath) = d_\rho \\ &= \frac{1}{n}d_\rho + \frac{n(n-1)}{2}\frac{2}{n^2}\chi_\rho(\tau) \quad \text{since there are } \tbinom{n}{2} \text{ possible transpositions.}\end{aligned}$$

Therefore, $\quad \lambda = \dfrac{1}{n} + \dfrac{(n-1)}{n}\dfrac{\chi_\rho(\tau)}{d_\rho}.$

To ease notation in the sequel, and keeping with that of Diaconis [8], I will write $r(\rho) = \frac{\chi_\rho(r)}{d_\rho}$ for any transposition r. This gives:

$$\lambda = \frac{1}{n} + \frac{(n-1)}{n}r(\rho).$$

By the Upper Bound Lemma, we have:

$$\begin{aligned}\left\|P^{*k} - U\right\|^2 &\leq \frac{1}{4}\sum_\rho^* d_\rho Tr\left(\widehat{P^{*k}}(\rho)\widehat{P^{*k}}(\rho)^*\right) \\ &= \frac{1}{4}\sum_\rho^* d_\rho\left(d_\rho\lambda^{2k}\right) \quad \text{since } \widehat{P^{*k}}(\rho)^* = \widehat{P^{*k}}(\rho) = \left(\hat{P}(\rho)\right)^k \\ &= \frac{1}{4}\sum_\rho^* d_\rho^2\left(\frac{1}{n} + \frac{(n-1)}{n}r(\rho)\right)^{2k}.\end{aligned}$$

Now, there exist methods of calculating the characters and dimensions of the irreducible representations of S_n, but this is quite time-consuming (for example, there are 42 of them to calculate for S_{10}). Such calculations can be avoided, however, by using certain properties of the representations of the symmetric group and the theory of *Young tableaux* [12]. This is another area of theory completely, and there is not space to give it justice here. Instead, I move straight to the main theorem of Diaconis [8], which uses the above bound as the foundation of its proof.

Theorem 7.6. *Let $k = \frac{1}{2}n\ log\ n + cn$. For $c > 0$,*

$$\left\|P^{*k} - U\right\| \leq ae^{-2c}$$

for a universal constant a.

I do not give the proof here: it is quite approachable, but very combinatorial in nature. The following discussion gives the idea of where this result comes from. Upon consideration of the values of $|r(\rho)|$ for fixed n, it turns out that these values are usually very small, with a few exceptions: these arise from representations of low dimensions. If it could be shown that $|r(\rho)| \leq \frac{1}{2}$ for most ρ, then for large n, $\left|\frac{1}{n} + \frac{n-1}{n}r(\rho)\right| \leq \frac{1}{2}$ approximately, and we would have

$$\left\|P^{*k} - U\right\|^2 \leq \frac{1}{4}\left(\frac{1}{2}\right)^{2k} \sum d_\rho^2.$$

Now, it can easily be shown that $\sum d_\rho^2 = |S_n|$ (this holds for *all* finite groups). The above bound then becomes $\frac{1}{4}\left(\frac{1}{2}\right)^{2k} n!$. We can next employ Stirling's formula:

$$n! \approx n^n e^{-n} \left(2\pi n\right)^{\frac{1}{2}},$$

giving, for $k = n\ log\ n$,

$$\begin{aligned}\left(\frac{1}{2}\right)^{2k} n! &\approx exp\left(log\ n\left[n\left(1 - 2\ log\ 2\right) + \frac{1}{2}\right] - n + \frac{1}{2}\ log\ (2\pi)\right) \\ &\to 0 \quad \text{as } n \to \infty\end{aligned}$$

(since the term in the square brackets is negative for $n \geq 2$).

In a similar manner it is possible to consider the term arising from the $n-1$ dimensional representation: it turns out that this is the slowest term, with most others being less than $\left(\frac{1}{2}\right)^{2k}$. This representation has character $n-3$, and so

$$\left(\frac{1}{n} + \frac{n-1}{n}r(\rho)\right)^{2k} = \left(1 - \frac{2}{n}\right)^{2k}.$$

This is a long way off from $\left(\frac{1}{2}\right)^{2k}$: we therefore need to adjust our value for k in order to kill this term in Lemma 7.5, i.e. we need to be able to kill

$$(n-1)^2\left(1 - \frac{2}{n}\right)^{2k}.$$

It can quite easily be shown that this term is equal to

$$exp\left(2k\ log\ \left(1 - \frac{2}{n}\right) + 2\ log\ (n-1)\right) = exp\left(-\frac{4k}{n} + 2\ log\ n + O\left(\frac{k}{n^2}\right)\right).$$

Hence, if we take $k = \frac{1}{2}n\ log\ n + cn$, this is asymptotic to e^{-4c}. Taking square roots of this leads to the e^{-2c} in Theorem 7.6. Although this argument is not particularly rigorous, it does shed some light on the origins of the upper bound produced in this theorem. This upper bound is therefore an improvement on that found via the coupling method, although only by a factor of four. A proper comparison of the bounds produced and the actual number of iterations taken by my program to reach convergence will be given in Section 9.

8 Application: Internet Search Engines

The method developed and put to use in this project to decode substitution codes may be adaptable to an internet search engine algorithm. Markov chains are currently used in this field: indeed, statistical language modelling is often crucial to the process of information retrieval. One such method used in information retrieval systems is that of Natural Language Processing (NLP) [20].

NLP involves two mathematical procedures: Markov chains and scoring. The Markov chain element is similar to that used in my decoding programs. The idea is to record prior *word* transition probabilities in a language, just as I used a matrix of character transition probabilities. With this in mind, the British National Corpus has been developed [15]: this is a collection of more than $4,000$ text files, on a wide variety of subjects, totaling about 100 million words (slightly more than in *War and Peace*)! Suitable prior transition matrices can then be formed using text from this database.

The retrieval system works as follows. Each component in a user's query is given a weighting according to the scoring scheme. Suitable documents containing these components are found, and the text adjacent to these words considered: these documents are then assigned a probability of relevance. The method for calculating this probability involves multiplying-up the appropriate word transition probabilities for the surrounding text, using the prior transition matrix, aiming to maximize this value. The documents retrieved are then ranked by this relevance probability before being suggested to the user.

This is clearly similar to the method used above to find the best decoding. There, the program aims to find the best possible permutation $\hat{\sigma}^{-1}$ using character transition frequencies. With the web, the most useful document is targeted with the use of word transitions. To the best of my knowledge, MCMC is not actually used at the moment in this process, but its applicability is obvious. Such methods may, however, prove far too time-consuming to be practical, especially since the growth of the web means a rapidly-evolving link structure. This means that large-scale ranking computations, such as those used in NLP, are becoming too expensive.

Other methods are currently being developed though, and in many cases these are still based upon the N-gram structure of language. One idea is to categorize texts by their letter frequencies. This assumes that similar texts have similar character frequencies, just as we saw with the English novels in

Section 4. The aim of this is to encourage search engines to return only a list of documents which are of a suitable nature.

It is extremely difficult to model natural language effectively - Markovian language models using N-grams are widely acknowledged to be too crude, but it is hard to find anything better at present. Grammatical constructions and long-distance correlations between words, for example, are ignored under this framework. Many other methods are currently being proposed and developed (e.g. a whole-sentence exponential model [19]), but there will always be room, and indeed need, for improvement in this important field.

9 Conclusions

This project has shown that Markov Chain Monte Carlo techniques can be successfully employed in the deciphering of simple substitution codes. Of course, there are many much more efficient methods available for solving such easy codes, but it does serve to emphasize the wide applicability of this area of theory. I have shown, through the development of my own program to implement a version of the Metropolis-Hastings algorithm, that this method can frequently decode text to 99% accuracy. Using this program I then investigated certain methods of improving the accuracy of the decoding. My final recommendation for this is to firstly split the text into two halves, and then to decode each part using the intelligent starting permutation scheme explained in Section 5.4: the final decoding is formed by using the resulting permutation with the lowest negative log-likelihood. This should significantly improve the frequency with which a good deciphering is produced, and also reduce the mean number of iterations taken for the Markov chain to converge.

The expected number of iterations when the chain is run without the help of this starting state turned out to be $29,459$, with a standard deviation of $24,766$ (these figures come from the 66% of runs that reached 99% accuracy within $200,000$ iterations). In assessing these figures, I attempted to describe the 'likelihood surface' traversed by the Markov chain. This turned out to be extremely complex, with peaks and troughs producing periods of meta-stability: often these stopped the program from terminating when using the percentage accuracy diagnostic. I also investigated the use of the negative log-likelihood as a diagnostic measure: it turned out to be very effective in this capacity. Indeed, it is such a consistent measure across English novels that the negative log-likelihood of the target permutation can be calculated in advance simply from knowledge of the length of the text. This means that meta-stability can be avoided and convergence easily diagnosed.

I then presented two very different theoretical methods for calculating the number of shuffles it takes to randomize a pack of n cards. The shuffling setup for each of these was much simpler than the method of character transpositions used by my program, since no information about the current state of the pack was involved. From these I obtained two upper bounds on the required number

of transpositions: $2n\ log\ n$ for the coupling method, and $\frac{1}{2}n\ log\ n$ for that using representation theory. Since, for my program, I was working with the group S_{79}, these upper bounds become 690 and 173 respectively when $n = 79$. These are clearly much lower than the actual number of iterations taken by my decoding chains to reach equilibrium. It hence appears that the speed of convergence is greatly reduced when the chain takes account of its present position, i.e. when it uses the data available to it. It is currently an open problem to find a way of integrating the unconditioned information from the coupling or group representation methods into the algorithm. The hope is that this may be able to significantly increase the convergence speed, since the theory shows that it is definitely not the huge size of the state space ($n!$) that slows down the algorithm. There may, however, be no way of solving this problem, since it may be the case that the data completely breaks the symmetry of the unconditional chain.

Now, since $|Alph| = 79$, there are $\binom{79}{2} = 3,081$ possible transpositions that the program can propose. Define T_i to be the number of transpositions proposed before a new one is proposed, given that i have already been considered. Then if T is the time until *all* possible transpositions have been suggested, we can write

$$T = \sum_{i=0}^{k-1} (T_{i+1} - T_i) \quad \text{where } k = \binom{n}{2}.$$

If i transpositions have already been proposed, then a new one is proposed with probability $\left(\frac{k-i}{k}\right)$, and hence $(T_{i+1} - T_i)$ is Geometric with mean $\left(\frac{k}{k-i}\right)$. Therefore,

$$\begin{aligned}
\mathbb{E}[T] &= 1 + \frac{k}{k-1} + \frac{k}{k-2} + \ldots + \frac{k}{1} \\
&= k\left[1 + \ldots + \frac{1}{k-1} + \frac{1}{k}\right] \\
&\approx k\ log\ k.
\end{aligned}$$

Since $k = 3,081$, this gives the expected number of transpositions needed for all possibilities to have been suggested as $24,750$. This is much closer than the theoretical upper bounds to the actual mean number of iterations taken by the program: $29,459$. It hence looks as though in practice it is necessary to run the chain for a length of time that allows all possible permutations to be suggested. This was certainly the case in my investigations, where I recommended that the program should be allowed to run for at least 60-70 thousand iterations.

Therefore I have demonstrated the use of MCMC in decoding, and have implemented my own program to this effect. I have compared the convergence rates in reality with those produced by the theory, and have attempted to explain their difference in magnitude. I also gave a brief insight into possible uses of this kind of algorithm in internet search engines, although it would appear that such uses are limited in practice. There are clearly many further issues to be investigated in extension of this work. There are many implementational issues, for example, that I was unable to address due to lack of time and want of a

faster program, as mentioned in Section 5.6. The two areas of theory also have much potential, especially that of representation theory, although this is limited to random walks generated by probabilities that are constant on conjugacy classes.

A Computer Code

All relevant code produced during this project is included below. It must be emphasized that, to the best of my knowledge, these programs implement the Metropolis-Hastings algorithm with the results presented in Section 5. I do not claim that these are the most efficient way of producing such an algorithm; they are included purely for completeness.

```
Alph = {"!", "*", "(", ")", "-", "[", "]", ".", ",", ";", "\"",
        "?", "'", "/", ":", "\n", " ",
        "0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
        "a", "A", "b", "B", "c", "C", "d", "D", "e", "E", "f", "F",
        "g", "G", "h", "H", "i", "I", "j", "J", "k", "K", "l", "L",
        "m", "M", "n", "N", "o", "O", "p", "P", "q", "Q", "r", "R",
        "s", "S", "t", "T", "u", "U", "v", "V", "w", "W", "x", "X",
        "y", "Y", "z", "Z"}
```

→ This is the character set ("\n" corresponds to a carriage return).

```
prob[code_/;VectorQ[code],alph_/;VectorQ[alph]]:=
    Module[{matrix, xx, yy},
        matrix = Table[Table[0, {Length[alph]}], {Length[alph]}];
        For[ii = 1,
            ii < Length[code],
            ii = ii + 1,
            xx = Flatten[Position[alph, code[[ii]]]];
            yy = Flatten[Position[alph, code[[ii + 1]]]];
            matrix[[xx, yy]] = matrix[[xx, yy]] + 1
        ];
    matrix
    ]
```

→ Returns the transition matrix for `code`, whose rows/columns follow the ordering of `alph`.

```
makefreq[mat_/;MatrixQ[mat]]:=
    Module[{temp, yy},
    temp = Table[0, {Length[mat]}];
    yy = Apply[Plus, Apply[Plus, mat]];
    For[ii = 1,
        ii <= Length[temp],
        ii = ii + 1,
        temp[[ii]] = Apply[Plus, mat[[ii]]]/yy
        ];
    temp
    ]
```

→ Converts matrix `mat` into a vector of relative frequencies.

```
freq = Log[makefreq[pr]]
```

```
compare[xx_,yy_]:=
    If[xx < yy, 1, 0]
SetAttributes[compare, Listable]
```

→ Allows comparison of `xx` with `y`, irrespective of the level at which each is nested within a list.

```
alphify[lst1_,lst2_,lst3_]:=
    Module[{temp},
        temp = lst1;
        For[ii = 1,
            ii <= Length[lst2],
            ii = ii + 1,
            temp[[Flatten[Position[lst1,lst2[[ii]]]]]]=lst3[[ii]]
        ];
        temp
    ]
```

→ Decodes `lst1` using permutation `lst3`, assuming `lst2` to be the identity permutation.

```
swap3[mat_/;(MatrixQ[mat]&&Apply[Equal,Dimensions[mat]]),ii_,jj_]:=
    Module[{temp, temp2},
        temp2 = mat;
        temp2[[ii]] = mat[[jj]];
        temp2[[jj]] = mat[[ii]];
        temp = temp2;
        temp2[[All, ii]] = temp[[All, jj]];
        temp2[[All, jj]] = temp[[All, ii]];
        temp2
    ]
```

→ Returns the new transition matrix obtained when `mat` is the current transition matrix and characters `ii` and `jj` are proposed.

```
swap[mat_/;(MatrixQ[mat]&&Apply[Equal,Dimensions[m]]),ii_,jj_]:=
    Module[{temp, mat2},
            temp = Table[Table[0, {Length[mat]}], {Length[mat]}];
            temp[[ii]] = mat[[jj]] - mat[[ii]];
            temp[[jj]] = -temp[[ii]];
            temp = Transpose[temp];
            mat2 = Transpose[mat];
            temp[[ii]] = mat2[[jj]] - mat2[[ii]];
            temp[[jj]] = -temp[[ii]];
            temp = Transpose[temp];
            temp[[ii, ii]] = mat[[jj, jj]] - mat[[ii, ii]];
            temp[[jj, jj]] = -temp[[ii, ii]];
            temp[[ii, jj]] = mat[[jj, ii]] - mat[[ii, jj]];
            temp[[jj, ii]] = -temp[[ii, jj]];
            temp
    ]
```

→ Returns the difference between two transition matrices, calculated explicitly entry-by-entry.

```
 tranAlph[lst_/;VectorQ[lst],no2_,no3_]:=
    Module[{temp},
            temp = lst;
            temp[[Flatten[Position[lst,Alph[[no2]]]]]]=Alph[[no3]];
            temp[[Flatten[Position[lst,Alph[[no3]]]]]]=Alph[[no2]];
            temp
    ]
```

→ Exchanges characters `no2` and `no3` in permutation `lst`.

```
logprob[mat_/;MatrixQ[mat]]:=
    Module[{prmat, help, logprmat},
        prmat = DiagonalMatrix[Table[0, {Length[mat]}]];
        logprmat = DiagonalMatrix[Table[0, {Length[mat]}]];
        help = Table[0, {Length[mat]}];
        For[ii = 1,
            ii <= Length[mat],
            ii = ii + 1,
            {help[[ii]] = (Apply[Plus, mat[[ii]]]),
            For[jj = 1,
            jj <= Length[mat],
            jj = jj + 1,

            prmat[[ii, jj]]=If[help[[ii]]==0,0,mat[[ii,jj]]/help[[ii]]];

            logprmat[[ii, jj]] =
            If[prmat[[ii, jj]] == 0, -12, Log[prmat[[ii, jj]]]]
            ]
            }
        ];
        logprmat
    ]
```

→ Returns matrix of log probabilities produced from `mat`.
log(0) is defined to be −12.

```
joiner[lst_/;VectorQ[lst]]:=
    Module[{start},
        start = lst[[1]];
        For[ii = 1,
            ii < Length[lst],
            ii = ii + 1,
            start = StringInsert[start, lst[[ii + 1]], -1]
        ];
        start
    ]
```

→ Converts a list into a string.

```
unpermute[lst_/;VectorQ[lst],alph_/;VectorQ[alph]]:=
    Module[{temp, xx},
        temp = Table[0, {Length[lst]}];
        For[ii = 1,
            ii <= Length[lst],
            ii = ii + 1,
            xx = Flatten[Position[lst, alph[[ii]]]];
            temp[[ii]] = alph[[xx]]
        ];
        Flatten[temp]
    ]
```

→ If permutation `lst` is applied to some text, then this returns the inverse permutation when using character set `alph`.

```
percentage[vec1_/;VectorQ[vec1],vec2_/;VectorQ[vec2],
            mat_/;MatrixQ[mat]]:=
    Module[{total, help},
        total = Table[0, {Length[vec1]}];
        For[vv = 1,
            vv <= Length[vec1],
            vv = vv + 1,
            If[vec1[[vv]] != vec2[[vv]], total[[vv]] = 1]
        ];
        help = makefreq[mat];
        newhelp = help;
        For[zz = 1,
            zz <= Length[help],
            zz = zz + 1,
            newhelp[[zz]]=help[[Flatten[Position[Alph,vec1[[zz]]]]]]
        ];
        Round[100*(1 - total.newhelp)]
    ]
```

→ Returns the percentage to which permutations `vec1` and `vec2` agree when applied to the text. Used when the target permutation (σ^{-1}) is known, to help test for convergence.

```
matwrpc = logprob[pr]; matwr = Flatten[matwrpc];
```

→ Produces the matrix ($log\mathcal{P}$), as well as its flattened (vector) representation.

```
likeprint[mat_/;MatrixQ[mat]]:=
     Round[(Flatten[mat]).matwr]
```

→ Calculates the negative log-likelihood of the matrix `mat`.

```
starter[code_/;VectorQ[code],freq1_/;VectorQ[freq1],
            freq2_/;VectorQ[freq2]]:=
        Module[{temp, temp1, temp2, help1, help2, final, xxx},
            temp1 = Table[{0}, {Length[freq2]}];
            temp2 = temp1;
            For[tt = 1,
                tt <= Length[freq2],
                tt = tt + 1,
                temp1[[tt]]=Insert[{freq1[[tt]]},Alph[[tt]],{2}];
                temp2[[tt]]=Insert[{freq2[[tt]]},Alph[[tt]],{2}]
            ];
            help1 = Sort[temp1][[All, 2]];
            help2 = Sort[temp2][[All, 2]];
            final = Table[0, {Length[help1]}];
            For[hh = 1,
                hh <= Length[final],
                hh = hh + 1,
                xxx = Flatten[Position[help1, Alph[[hh]]]];
                final[[hh]] = help2[[xxx]]
            ];
            {help1, help2, Flatten[final]}
        ]
```

→ Calculates the 'intelligent starting permutation' as described in Section 5.4.

```
solvitbound[code_/;VectorQ[code],number_/;IntegerQ[number],
               bound_/;IntegerQ[bound]]:=
    Module[{matp, len, ratio, ran, ran1, ran2, newmat, alph,
            extra1, liker, confound, extra2, no1, kk, nn,
            plotter1, plotter2, plotter3},

      nn = number*100;
      matp = prob[code, Alph];
      len = Length[matp];
      alph = Alph;
      liker = -likeprint[matp];
      confound = percentage[unperm, alph, matp];
      plotter1 = {{0, liker}};
      plotter2 = {Flatten[{0, confound}, 1]};
      plotter3 = {Flatten[{confound, liker}, 1]};
      no1 = Flatten[Position[Alph, First[code]]];
      kk = Ceiling[nn/5];

      For[jj = 1,
          jj <= 5,
          jj = jj + 1,

          Do[
            extra1 = 0;
            extra2 = 0;
            ran1 = Random[Integer, (len - 1)] + 1;
            ran2 = Random[Integer, (len - 1)] + 1;
            newmat = Flatten[swap[matp, ran1, ran2]];
            If[{ran1}==no1,
               {extra1=freq[[Flatten[Position[Alph,alph[[ran2]]]]]],
               extra2=freq[[Flatten[Position[Alph,alph[[ran1]]]]]]}
            ];
            If[{ran2}==no1,
               {extra1=freq[[Flatten[Position[Alph,alph[[ran1]]]]]],
               extra2=freq[[Flatten[Position[Alph,alph[[ran2]]]]]]}
            ];

            ratio = extra1 - extra2 + newmat.matwr;
            ran = Log[Random[]];
            If[compare[ran, ratio] == 1 || compare[ran, ratio] == {1},
                {alph = tranAlph[alph, ran1, ran2],
                matp = swap3[matp, ran1, ran2],
                confound = percentage[unperm, alph, matp],

                If[compare[bound, confound] == {1},
                    {liker = -likeprint[matp],
                    Print["Runs     Neg.Log.Like.    %'ge Correct"],
                    Print[N[(jj - 1)*kk + pp], "        ", liker,
                           "              ", confound],
                    AppendTo[plotter1,{(jj-1)*kk+pp,liker}],
                    AppendTo[plotter2,Flatten[{(jj-1)*kk+pp,confound},1]],
                    AppendTo[plotter3,Flatten[{confound,liker},1]],
                    Goto[overhere]}],
```

```
                no1 = Switch[no1,
                       {ran1}, {ran2},
                       {ran2}, {ran1},
                         _, no1]
            }
        ],
      {pp, kk}
      ];

      AppendTo[plotter1, {(jj - 1)*kk, -likeprint[matp]}];
      AppendTo[plotter2,
      Flatten[{(jj-1)*kk,percentage[unperm,alph,matp]},1]];
      AppendTo[plotter3,
      Flatten[{percentage[unperm,alph,matp],-likeprint[matp]},1]]
  ];

  Print["Unfinished     ", -likeprint[matp], "    ",
          percentage[unperm, alph, matp]];

  Label[overhere];

  ListPlot[plotter1, PlotJoined -> True];
  ListPlot[plotter2, PlotJoined -> True];
  ListPlot[plotter3, PlotJoined -> False];
  joiner[alphify[code[[Range[1, 200]]], Alph, alph]]
]
```

→ This is the main decoding program. The arguments are the encoded text, the upper bound on the number of iterations (in hundreds) and the percentage accuracy required for the chain to stop, respectively. When the algorithm stops, the following are displayed:

- the number of iterations completed ('Unfinished' if convergence is not reached)
- the negative log-likelihood of the final permutation
- the percentage agreement between the final permutation and the correct inverse permutation.

Three graphs are then produced:

- negative log-likelihood vs no. of iterations
- percentage accuracy vs no. of iterations
- negative log-likelihood vs percentage accuracy.

The first 200 characters of the final decoding are then displayed in string-format.

References

[1] D. Aldous and P. Diaconis, *Shuffling cards and stopping times*, American Mathematical Monthly **93** (1986), 333–348.

[2] C. Andrieu, N. de Freitas, A. Doucet, and M.I. Jordan, *An introduction to mcmc for machine learning*, Kluwer Academic Publishers, 2001.

[3] I. Beichl and F. Sullivan, *The metropolis algorithm*, Computing in Science and Engineering **2(1)** (2000), 65–69.

[4] P. Brémaud, *Markov chains, gibbs fields, monte carlo simulation, and queues*, Springer-Verlag New York Inc, 1999.

[5] S.P. Brookes, *Mcmc & its applications*, The Statistician **47** (1998), 69–100.

[6] V. Bryant, *Yet another introduction to analysis*, Cambridge University Press, 1990.

[7] D.R. Cox and H.D. Miller, *The theory of stochastic processes*, Methuen & Co. Ltd, 1965.

[8] P. Diaconis, *Group representations in probability and statistics*, Lecture Notes - Monograph Series, vol. 11, Institute of Mathematical Statistics, 1988.

[9] P. Diaconis and M. Shahshahani, *Generating a random permutation with random transpositions*, Z. Wahrscheinlichkeitstheorie Verw. Gebiete **57** (1981), 159–179.

[10] W. Feller, *An introduction to probability theory and its applications*, second ed., vol. II, John Wiley & Sons, 1971.

[11] L. Flatto, A.M. Odlyzko, and D.B. Wales, *Random shuffles and group representations*, Ann. Prob. **13** (1985), 154–178.

[12] W. Fulton, *Young tableaux*, London Mathematical Society Student Texts, vol. 35, Cambridge University Press, 1997.

[13] D. Gamerman, *Stochastic simulation for bayesian inference*, Texts in Statistical Science, Chapman and Hall, 1997.

[14] W.R. Gilks, S. Richardson, and D.J. Spiegelhalter, *Markov chain monte carlo in practice*, Chapman and Hall, 1996.

[15] Y. Gotoh and S. Renals, *Topic-based mixture language modelling*, University of Sheffield.

[16] P. Green, *Tutorial lectures on markov chain monte carlo*, University of Bristol, 1998.

[17] W. K. Hastings, *Monte carlo sampling methods using markov chains and their applications*, Biometrika **57** (1970), no. 1, 97.

[18] http://promo.net/pg/index.html.

[19] http://www.cs.cmu.edu/∼roni.

[20] http://www.tgpconsulting.com/articles/mind.html.

[21] http://www.theregister.co.uk/content/1/12200.html.

[22] M. H. Kalos and P. A. Whitlock, *Monte carlo methods*, vol. 1, John Wiley & Sons Inc, 1986.

[23] R. Keown, *An introduction to group representation theory*, Mathematics in Science and Engineering, vol. 116, Academic Press Inc., 1975.

[24] A. A. Markov, *Appendix to the calculus of probability*, 4 ed., Moscow, 1924.

[25] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, *Equations of state calculations by fast computing machines*, J. Chem. Phys. **21** (1953), 1087.

[26] C. E. Shannon and W. Weaver, *The mathematical theory of communication*, University of Illinois Press, 1949.