A Markov Chain Case Study: Card Shuffling

Yongwhan Lim

December 20, 2012

Abstract

In this thesis, I highlight important results in the mixing time of card shufflings. Then, I pose a number of interesting problems and provide simulation and proof to address the problems. I conclude by suggesting future work.

Contents

1 Overview

There are number of interesting mathematical problems about shuffling cards. Of those, in [1], the authors settled the mixing time for the riffle shuffle, concluding that under the Gilbert-Shannon-Reeds (GSR) model, it is sufficient to use riffle shuffle only about 7 times to sufficiently randomly shuffle a deck of 52 cards.

I am hoping to expand the statistical literature and analyze their applications to this topic by proposing more interesting questions and providing results on them. I will do this partially by running a computer simulation and more fully by proving a closely related statement. In the case of the former, I provide conjectures that seem to hold based on the simulation. In particular, here is the highlight of the problems that I am considering in this thesis.

1. Find an optimal shuffling sequence given a finite set of shuffling methods.
2. Compare a customized shuffling method to other benchmark methods by simulation using off-the-shelf test statistics.
3. Use a comparison technique [2, 3] and recent result on k-cycle shuffles [8] to determine an upperbound on the mixing time of a special class of card shuffling.

This thesis is organized as follows. First, I will provide a brief overview of the related literature on the mixing time study of card shuffling. Then, I will visit the highlighted topics one by one. I will conclude by the future work. The interested readers could download the source code by following the link provided in the last section.

2 Related Literature

There are few lines of work related to card shuffling that are directly related to the problems we are considering in this thesis. One is the analysis of mixing time of Markov chain arising from special classes of card shuffling. The other is the comparison technique, which is the technique we used to prove the upperbound in Section 5. The problems we consider here are inspired from careful reading of the card shuffling

literature. There are number of standard textbooks that one should peruse to get familiar with the topic first such as [4, 6, 7].

2.1 Mixing Time

There several lines of work here. We will focus on following key results: the mixing time bound on a riffle shuffle and an overhand shuffle [1, 10, 11]. A reader should also find [5] interesting, for a number of other results can be proven using the finite Fourier methods.

2.1.1 Riffle Shuffle : $\frac{3}{2}\log n + \theta$

First, as I briefly mentioned in the introduction, the most celebrated result is perhaps the one on the riffle shuffle [1] where the author settles that the mixing time is $\Theta(\log n)$ for riffle shuffle. To be more precise, a riffle shuffling under the Gilbert, Shannon, and Reeds model is the following: a deck of n cards is cut into two portions according to a binomial distribution. The two packets are then riffled together such that cards drop from the left and right packets with probability proportional to the number of cards in the packet.

Let S_n be the symmetric group, U the uniform probability, and Q^m the Gilbert-Shannon-Reeds probability after m shuffles. Then, the total variance distance that we would like to approximate is:

$$||Q^m - U|| = \max_{A \in S_n} |Q^n(A) - U(A)|.$$

The main theorem in [1] is the following theorem:

Theorem 1. *If n cards are shuffled m times with $m = 3/2 \log_2 n + \theta$, then for large n,*

$$||Q^m - U|| = 1 - 2\Phi\left(\frac{-2^{-\theta}}{4\sqrt{3}}\right) + O\left(\frac{1}{n^{1/4}}\right)$$

with

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

Thus, the variation distance tends to 1 with θ small and to 0 with θ large.

2.1.2 Overhand Shuffle : $\Theta(n^2 \log n)$

There are two works on the overhand shuffle. The upperbound is first proven in [10]. Subsequently, in [11], the author showed that the bound $\Theta(n^2 \log n)$ is tight by proving the lowerbound using an extension of a Wilson's lemma.

To be more precise, the model Pemantle used for the overhand shuffle is parameterized by a probability $p \in (0, 1)$. Each of the $n - 1$ slots between adjacent cards is, independently of the other slots, declared a cutpoint with probability p. A model is obtained by reversing each of these packets without changing its position relative to

the other packets. We assume a *circular deck convention*, which means the top card and the bottom card regarded as being adjacent to each other.

The author of [11] proves the following main theorem of the paper using an extension of Wilson's lemma:

Theorem 2 ([2])**.** *A lower bound for the overhand shuffle with with circular deck convension and with parameter $p \in (0,1)$ is given by*

$$\frac{p^2(2-p)}{8\pi^2(1-p^2)} n^2 \log n.$$

2.2 Comparison Technique

There are two versions of comparison techniques [2, 3]: one on the random walk on finite groups and the other on the reversible Markov chains. They are powerful techniques that can often be applied to provide an upperbound on mixing time of Markov chains.

To be more precise, note that, given real-valued function ϕ on G, the eigenvalues of symmetric probabilities can be characterized by quadratic forms:

$$E_p(\phi,\phi) = E(\phi,\phi) = \langle (I-P)\phi, \phi\rangle,$$
$$F_p(\phi,\phi) = F(\phi,\phi) = \langle (I+P)\phi, \phi\rangle.$$

We call E_p a *Dirichilet form*. We develop a bound of type $\tilde{E} \le AE$ for Dirichlet forms associated with symmetric probabilities $\tilde{p}$ and p on a finite group G. Let E be a symmetric set of generators of the finite group G. For $y \in G$, write $y = z_1 \cdots z_k$ with $z_i \in E$. The smallest such k is called the length of y, denoted by $|y|$. Also, $|y|_*$ is defined to be the length of the shortest representation of y as a product of an odd number of generators. Let $N(z,y) = N_*(z,y)$ be the number of times $z \in E$ occurs in the chosen representation of y. Then we have:

Theorem 3. *Let $\tilde{p}$ and p be symmetric probabilities in a finite group G. Let E be a symmetric set of generators. Suppose that the support of p contains E. Then the Dirichlet forms satisfy*

$$\tilde{E} \le AE,$$
$$\tilde{F} \le A_* F$$

where

$$A = \max_{z\in E} \frac{1}{p(z)} \sum_{y\in G} |y| N(z,y) \tilde{p}(y),$$
$$A = \max_{z\in E} \frac{1}{p(z)} \sum_{y\in G} |y|_* N_*(z,y) \tilde{p}(y),$$

In this thesis, we are going to use the above version of the comparison technique on the proof of the upperbound on the k-uniform shuffle.

3 Optimal Shuffling Technique

3.1 Introduction

Fix a permutation group S_N on N elements. Typically, the measure of the randomness of a particular shuffle Q on S_N is calculated by comparing the transition probability to the uniform measure U using the total variation distance; that is, $||Q - U||_{TV}$. We have already seen the example above where we chose Q to be riffle shuffle or overhand shuffle. By bounding the total variance distance between the shuffle method Q and the uniform measure, we obtained $n \log n$ bound on the riffle shuffle and $n^2 \log n$ bound on the overhand shuffle.

Here, we are interested in explicitly characterizing the best shuffling methods given a finite set of shuffling methods (e.g., riffle, overhand, cut, and etc.).

Say $\mathcal{M} = \{M_1, \cdots, M_n\}$ are a finite set of n shuffling methods where M_i is the ith shuffle method. Then, corresponding to each M_i, we have a transition matrix Q_i. Also, write $\mathcal{I}_k$ to denote the set of all index sequences of length k: explicitly, $(i_1, ..., i_k)$ where $i_j \in \{1, \cdots, n\}$ for each $j = 1, \cdots, k$. Now, for $I \in \mathcal{I}_k$, write Q_I to mean a convolution sequence of Q's corresponding to I: explicitly, $Q_{(i_1,\cdots,i_k)} = Q_{i_1} * \cdots * Q_{i_k}$. Then, here is a short list of selected questions that we are interested in resolving:

Q1. Fix $\epsilon > 0$. What is the minimum k such that there exists some $I \in \mathcal{I}_k$ satisfying $||Q_I - U||_{TV} < \epsilon$? Find asymptotic formula for k with respect to ϵ; also, characterize such I's; in particular, find out subset of $\mathcal{S}$ that are not useful at all. Also, for each shuffle method, typically, how many times do we need to use them to meet the requirement? Generally, what is the underlying distribution for such k's (not only the minimum)? In practice, this answers the question of if we would like the deck of cards to be sufficientlly random using the available shuffling techniques, how many shufflings would be enough and what the required underlying sequence of methods is. How would the answer change if for each index i, we have additional restriction that it must be used at least p_i times but at most q_i times. What if each i's have a probability of p_i of being selected where $\sum_{i=1}^{n} p_i = 1$?

Q2. Fix k. What are $I \in \mathcal{I}_k$ that minimize $||Q_I - U||_{TV}$? In general, what is the underlying distribution of $||Q_I - U||_{TV}$? Concretely, this translates to the best way to shuffle the deck of cards, given that the total number of shuffles is limited.

3.2 Explicit Computation

We use computer simulation to explore the first question. We restrict ourselves to the case where $N = 5$ and $n = 5$; we allow top-to-random, random transpositions, overhand, riffle, and cut. We would like to find the minimum k for a given $\epsilon > 0$. Now, let's describe each of the shuffling methods and provide the corresponding result. I include the complete implementation in the appendix as a reference.

3.3 Shuffling Method Types

There are a lot of shuffling methods; of those, the following five methods are most prominent in the mathematical literature. Say that there are cards with label $0, \cdots, N-1$ in a deck. In **cut** [6], there is equal chance of picking a position before $0, \cdots, N-1$ and after $N-1$ to cut. In **overhand** [10] shuffle, there is equal chance of choosing positions to cut the deck. Once these positions are determined, the new deck is formed by stacking from the back, parsed by the chosen positions. In **riffle** shuffle [1], there is equal chance of choosing the place to cut. Then, for each remaining k and $N-k$ cards, all the possible arrangement of these cards are counted with unit weight. In **transposition** [6], each label i and j have equal chance of being chosen, each with probability $1/N^2$. In **top-to-random** [6] shuffle, each label i has equal chance of being chosen, with probability $1/N$.

3.4 Result

When $\epsilon = 10^{-5}$, we have the following results for minimum k. Since getting an exact number may be infeasible, as there are too many sequences to consider, we give an upperbound obtained by running each case at most `niter` times where each method is randomly chosen with an equal probability. For increasing value of `niter`, if we see that the entry in the table is not changing (i.e., converged) then it gives a bit more confidence that the entry may be tight. The table provided below is obtained when the number of iteration is set to 500; highlighted ones are the ones that have an upperbound for a mini mum k smaller than the one for the riffle shuffle only:

#	type	minimum k
1	cut	∞
2	overhand	≤ 56
3	cut, overhand	≤ 87
4	riffle	≤ 15
5	cut, riffle	≤ 18
6	overhand, riffle	≤ 16
7	cut, overhand, riffle	≤ 19
8	trans	≤ 24
9	cut, trans	≤ 26
10	overhand, trans	≤ 23
11	cut, overhand, trans	≤ 27
12	riffle, trans	≤ 16
13	**cut, riffle, trans**	≤ 14
14	**overhand, riffle, trans**	≤ 13
15	**cut, overhand, riffle, trans**	≤ 13
16	t-b	≤ 26
17	cut, t-b	≤ 31
18	overhand, t-b	≤ 28
19	cut, overhand, t-b	≤ 35
20	riffle, t-b	≤ 16
21	cut, riffle, t-b	≤ 19
22	overhand, riffle, t-b	≤ 18
23	cut, overhand, riffle, t-b	≤ 22
24	trans, t-b	≤ 25
25	cut, trans, t-b	≤ 17
26	overhand, trans, t-b	≤ 17
27	cut, overhand, trans, t-b	≤ 16
28	riffle, trans, t-b	≤ 16
29	**cut, riffle, trans, t-b**	≤ 14
30	**overhand, riffle, trans, t-b**	≤ 13
31	**cut, overhand, riffle, trans, t-b**	≤ 14

The above table is created by obtaining a following table for the total variation distance over iterations corresponding to each case. In particular, we obtain the following table for case #4:

0	0.7750000000
1	0.2892038297
2	0.1198392702
3	0.0518480347
4	0.0227610018
5	0.0100659162
6	0.0044644191
7	0.0019817814
8	0.0008802168
9	0.0003910758
10	0.0001737822
11	0.0000772301
12	0.0000343231
13	0.0000152544
14	0.0000067797

Hence, for the above table, we get k to be 15; hence, the upperbound for a minimum k is 15. This bound gets refined as we do more iterations.

3.5 Analysis

Note that cases $\#1, 2, 4, 8, 16$ all match the theoretical guarantee. Also, comparing each consecutive even and odd pairs reveal that cut does not help with the following exceptions: cases $\#12, 28$. Mixing cut, riffle, and transposition decreases the upperbound for minimum k. An example sequence that achieves $k = 13$ is the following:

$$\{2, 2, 2, 2, 3, 1, 3, 3, 2, 3, 1, 3, 2\}$$

where 1, 2, and 3 denote, respectively, overhand, riffle, and transposition.

It seems overhand shuffle is essentially the least effective shuffle because it requires a lot more iterations to achieve the same total variation distance.

Also, we can obtain the following kind of typical count table corresponding to each case. In particular, for case #31, we have:

$$\{10, 28, 40, 44, 53, 49, 57, 44, 47, 38, 38, 17, 11, 8, 6, 1, 4, 4, 1\}$$

This means, out of 500 iterations, we have $k = 14$ for 10 iterations, $k = 15$ for 28 iterations, and so on.

As we can observe from the table, I conjecture that combining all the shuffling methods helps in decreasing the minimum number of shufflings required to reach the uniform measure - in particular, the optimal shuffling is not the one obtained from using riffle shuffle only.

Lastly, when we vary ϵ in log scale, we see that the minimum k changes approximately linearly. So, I conjecture that there should be a similar type of result on the mixing time on the mixed type shuffles.

4 Explicit Shuffling Simulation

4.1 Introduction

In this simulation, we want to see if we can gather some statistical evidence to see if the shuffle used in the World Series of Poker (WSOP) tournament is too "weak" to provide the required randomness - in particular, we compare the test statistics of WSOP to those of riffle-only and strip-only, to get a rough estimation on the number of shuffles required to get an "equivalent" shuffle, if we are to use only a single shuffle method; following page 1 of the 2011 World Series of Poker Official Dealer Guide [9], we define the WSOP shuffle to be a shuffle method that has (riffle, riffle, strip, riffle, cut) as a shuffle sequence. Since we already know some asymptotic results of riffle / strip, the idea is to reduce down the complexity of analyzing the complex shuffle method to that of analyzing a repeated application of a single method - for strip shuffle, we follow the method used in the classic papers on it [10, 11], where the authors showed that it takes $\Theta(n^2 \log n)$ to mix the deck of n cards. As before, we provide the code in the appendix.

4.2 Setup

Let's make our experiment setup explicit. Suppose there are N cards to be shuffled (of course, N=52 in a standard deck). Fix the test statistics - here, we fix it to be the sum of the absolute difference between the next card values from 1 to $N-1$; to be completely precise, for each index i, we are considering $|v(i+1) - v(i)|$. Though I tried some variants and found this particular statistic to be the best, it might be possible to find a better test statistics to distinguish the shuffle methods. The idea is to compare the statistics as we change the shuffle methods, to reveal how powerful the shuffle method is. We fix K, the number of trials (i.e., the number of times we perform the set of shuffles per method). We repeat this experiment (a set of K trials) T times (the fixed number of iterations). Then, we take the average of the p-values, which we get from the standard χ^2 goodness of fit test.

4.3 Result

In the table below, we provide a summary for the average of p-values as we change K and T. For each pair (K, T), we provide the average p-values for (7 riffle, 3 riffle, 4 riffle, 50 strip, 15 strip, 30 strip, WSOP) over T experiments. We find:

(K, T)	7 riffle	3 riffle	4 riffle	50 strip	15 strip	30 strip	WSOP
(100, 1000)	47.06%	45.95%	46.13%	**44.64**%	45.43%	47.19%	45.72%
(100, 2000)	46.13%	45.21%	45.52%	46.64%	46.75%	47.38%	45.54%)
(5000, 100)	29.71%	30.54%	26.25%	36.95%	30.05%	42.02%	22.15%
(10000, 100)	14.71%	**12.74**%	**11.99**%	19.71%	14.88%	16.85%	12.99%

In the above result, we bold the entries that correspond to the shuffle method that has lower average p-values than the WSOP one − we see that WSOP shuffle is no

better than 3-4 riffle shuffles and 15-30 strip shuffles, using this statistical test. As we can see from K=100 experiments, we should expect to see column 1 > column 3 > column 2 and column 4 > column 6 > column 5, as we increase T; assuming this is going to be the case, we reduced the computation for all the other values of K to just $T = 100$, to keep the computational cost minimal.

5 Variation on Overhand Shuffle

For small k, we use the comparison technique [2, 3] and the recent result on the k-cycle shuffles [8] to get an upperbound on the k-uniform overhand shuffle; before we start, we will make an observation of why the naive attempt of comparing to the classical result on random-to-random shuffles fails.

5.1 Introduction

The comparison technique can be widely applied. Here, we are going to use the technique, coupled with the recent result on the k-cycle shuffles, to get a reasonable upperbound. Before we start, let's define what we mean by the k-uniform shuffle first. For a fixed k and n, we uniformly sample k numbers from $1, \cdots, n$ and order them as $1 \leq a_1 \leq \cdots \leq a_k \leq n$. Then, visualizing $1, \cdots, n$ as a stack of cards and a_i's as the cut position, we stack the deck in the reverse order. As we already saw in Section 2, we are essentially interested in A, the crucial term needed to apply the comparison technique:

$$A = \max_{z \in E} \frac{1}{p(z)} \sum_{y \in S_n} |y| N(z, y) \tilde{p}(y).$$

The first obvious approach is to try to extend the approach used in the Crude Overhand Shuffle and Other Shuffles section of [2]. As we will see in the next section, however, this approach fails. The next approach is to compare this with the k-cycle shuffles, which is recently proved to have the following mixing time result:

Theorem 4. *the upperbound on the mixing time that has generators as a set of k-cycle in S_n is in the order of $\frac{n}{k} \log n$.*

The main result we wish to establish in this section is:

Theorem 5. *For small k, the upperbound on the mixing time of k-uniform overhand shuffle is in the order of $n \log n$.*

5.2 Special Case: $k = n - 2$

Since the k-uniform overhand shuffle is hard to analyze, we consider the following simplified case where $k = n - 2$ and $n > 2$. We define the Markov chain on n cards by allowing all the possible $n - 1$ sets of $(n - 2)$ cut-positions to be uniformly chosen. Using an elementary proof, we show that the cards do not mix to the uniform measure.

5.2.1 Notation

Before diving into the proof, to simplify the notation, let's define some notations. In the Boolean notation, let's agree that 0 means the entry must be 0 and 1 means not necessarily.

For a Boolean vector v, $\neg v$ means to negate each element of the vector, where a negation $\neg$ for a single Boolean entry is defined as $\neg 0 = 1$ and $\neg 1 = 0$. Now, let's define $S(v, i, f)$ to be the i-times concatenation of alternating v and $\neg v$, where we start with v if $f = 1$ and $\neg v$ otherwise. Moreover, define the vector sequences A_n by $A_1 = [0]$ and $A_i = S(A_{i-1}, n, i \mod 2)$ for all $i > 1$. Finally, define the matrix M_n by concatenating A_n column-wise using each entry of A_n as a flag f for S; to be more precise, ith row is of the form $S(A_n, 1, A_n(i))$.

Now, let's write S_n to mean a set of all matrices of the forms either M_n or $\neg M_n$. Then, we can deduce the following property of S_n:

Proposition 1. *If $A, B \in S_n$ then $AB \in S_n$.*

Proof. Note that $A_n \cdot A_n^T = *$ and $A_n \cdot (\neg A_n)^T = 0$. Also, by symmetry, we only have two cases to check: when $A = B = M_n$ and $A = M_n$ while $B = \neg M_n$. Call $C = AB$.

Note that $C(i, j) = \sum A(i, k)B(k, j)$. Depending on whether $B = M_n$ or $B = \neg M_n$, we see this expression is either $A_n \cdot A_n^T$ or $A_n \cdot (\neg A_n)^T$, which we already know are either $*$ or 0; in fact, each row $C(i, \cdot)$ is of the form $A(i, \cdot)$ and $\neg A(i, \cdot)$, precisely depending on i and whether B is M_n or $\neg M_n$. So, we conclude that C must be of the form M_n or $\neg M_n$, as desired // □

5.2.2 Proof of $k = n - 2$ case

Using the usual notation, let $\pi_n(i, j)$ be the transition matrix for n cards, of order $n! \times n!$. Then, using the result of the proposition, we need to see that $\pi_n^k \in S_n$ for any positive integer k. This can be checked by the routine induction. First, we check the initial transition matrix is in S_n, which is trivial since there are $n - 1$ non-zero entries for each of the $n!$ rows, all of which lies in the $*$ position.

Then, for the induction hypothesis step, we appeal to the simple property of S_n proved above. In fact, whether we negate the matrix M_n or not depends precisely on the parity of k. Since there is 0 entry in each row of π_n^k regardless of what k is, we see that the Markov chain cannot converge to the uniform measure, as desired.

5.3 Comparison with classical results

Say π_k is the k-uniform overhand shuffle. Then, $p(\pi)$ is analogously defined to be the uniform measure with mass $\frac{1}{\binom{n+k-1}{k}}$ if $\pi = \pi_k$ and 0 otherwise. Also, $\tilde{p}(\pi) = \frac{1}{n^2}$ if $\pi = c_{ab}$ for some a and b but 0 otherwise. Written in this way, even if we can bound $|\pi|N(\pi_k, \pi)$ by some constant (which actually can only be bounded by $O(n^{k-2})$ so far), we will have $\frac{1}{k!} \cdot n^{k-2}$, which is, obviously, very poor bound, given the result of Pemantle and Johansson.

5.4 Comparison to k-cycle

Now, let's turn our attention to the comparison to k-cycle. As before, we define $p(\pi)$ to be the uniform measure with mass $\frac{1}{\binom{n+k-1}{k}}$ if $\pi = \pi_k$ and 0 otherwise. Now, $\tilde{p}(\pi) = \frac{1}{\binom{n}{k} \cdot (k-1)!}$ if $\pi = C_k$ but 0 otherwise. This is indeed the right mass as there are precisely $\binom{n}{k} \cdot (k-1)!$ k-cycle in S_n.

Also, since k-cycle can be written as a product of $k-1$ transpositions, it suffices to write transposition as a m composition of π_k's. Then, $|\pi| = m \cdot (k-1)$. We have the following **Lemma**:

Lemma 1. *When $k > 3$, we have $N(\pi_k, \pi) \leq 8k$ where $|\pi| = 2$.*

Proof. When $k > 3$, we can always write a transposition as a composition of two π_k's; there are $8k-4$ different representations when $n = k+1$ and $8k-12$ otherwise. So, we always have $\leq 8k$, as desired. Supply with more detailed proof later. □

Also, we have the following estimation:

Lemma 2. *For all $n \geq k \geq 3$, we have $\frac{\binom{n+k-1}{k}}{\binom{n}{k} \cdot (k-2)!} \leq 21$.*

Proof. By inspection, the peak happens when $n = k = 5$, with the value 21; we should supply more detailed proof later, if necessary. □

Combining this estimation yields:

$$\tau_{mix} \leq \frac{\binom{n+k-1}{k}}{\binom{n}{k} \cdot (k-1)!} \cdot 2(k-1) \cdot 8k \cdot \frac{n}{k} \log n \leq 21 \cdot 2 \cdot 8 \cdot n \log n = 336 n \log n$$

6 Future Work

In section 3, it would be interesting to develop a mathematical framework to find the optimal shuffling sequence. In section 4, a better test function should be found to replace the current one, if there is any. Also, the study is easily generalizable to a shuffling other than the WSOP variant we considered in the thesis. In section 5, it would be interesting to settle the question for any k; also, it would be interesting to extend the analysis to shufflings of more general type.

7 Appendix

7.1 Downloads

The source code is available in either `C++` or `MATLAB` versions from:

`www.stanford.edu/~yongwhan/math/thesis/code.zip`.

The electronic version of this thesis can be downloaded from:

`www.stanford.edu/~yongwhan/math/thesis/YLim.pdf`.

7.2 Code

7.2.1 Section 3

We provide the exact code used in simulation. Note that `t` is the transition matrix, `mp` saves the index for the permutation `v`, and default value for `mul` is 1. Also, `t` is normalized at the end, using the following code:

```
for (int j=0; j<n; j++) {
        double sum=0;
        for (int k=0; k<n; k++) sum+=t[j][k];
        for (int k=0; k<n; k++) t[j][k]/=sum;
}
```

The following code is called to get the transition matrix for each type:

```
// ncard: #cards in a deck
// type 0: cut
// type 1: overhand
// type 2: riffle
// type 3: transposition
// type 4: top-to-random
int get_transition_single(int ncard, int type) {
        vector<int> v(ncard);
        for (int i=0; i<ncard; i++) v[i]=i;

        int cur=0;
        do { mp[v]=cur; cur++; }
        while(next_permutation(v.begin(),v.end()));

        int n=cur;

        for (int i=0; i<ncard; i++) v[i]=i;
        cur=0;
        do {
                for (int j=0; j<n; j++) t[cur][j]=0;
                proc(v, ncard, type, 1);
                cur++;
        } while(next_permutation(v.begin(),v.end()));

        return n;
}
```

The proc function is implemented as:

```
void proc(vector<int> v, int ncard, int type, double mul) {
        int ll=(ncard+1);
        vector<int> w=v;
        switch(type) {
                case 0: // cut
                //      cout << "CUT" << endl;
                        for (int i=0; i<=ncard; i++) {
                                vector<int> w;
                                for (int j=0; j<ncard; j++)
                                        w.push_back( v[(i+j)%ncard] );
                                t[mp[v]][mp[w]]+=mul;
                        }
                        break;
                case 1: // overhand
                //      cout << "OVERHAND" << endl;
                        for (int i=0; i<(1<<ll); i++) {
                                vector< vector<int> > list;
                                int idx=0;
                                vector<int> cur;
                                for (int j=0; j<ll; j++)
                                        if(i&(1<<j)) {
                                                cur.clear();
                                                for (int k=idx; k<j; k++)
cur.push_back(v[k]);
                                                list.push_back(cur);
                                                idx=j;
                                        }
                                cur.clear();
                                for (int k=idx; k<ncard; k++)
cur.push_back(v[k]);
                                list.push_back(cur);

                                int ct=1;
                                double alpha=1;

                                int sz=list.size();
                                vector<int> w;
                                for (int j=sz-1; j>=0; j--) {
                                        cur=list[j];
                                        int l=cur.size();
                                        ct+=l; alpha*=(1.0/ct);
                                        for (int k=0; k<l; k++)
w.push_back(cur[k]);
```

```
                        }
                        if(ct!=ll) alpha*=(1.0/(ncard+1));
                        t[mp[v]][mp[w]]+=mul*alpha;
                    }
                    break;
                case 2: // riffle
                //      cout << "RIFFLE" << endl;
                        for (int i=0; i<=ncard; i++) {
                                vector<int> u,w;
                                for (int j=0; j<i; j++)
u.push_back(v[j]);
                                for (int j=i; j<ncard; j++)
w.push_back(v[j]);
                                for (int j=0; j<(1<<ncard); j++) {
                                        int cur=0;
                                        for (int k=0; k<ncard;
k++)
                                                if(j&(1<<k)) cur++;
                                        if(cur==i) {
                                                vector<int> vv;
                                                int ii=0, jj=0;
                                                for (int k=0;
k<ncard; k++) {
                                                        if(j&(1<<k)) {
vv.push_back(u[jj]); jj++; }
                                                        else {
vv.push_back(w[ii]); ii++; }
                                                }
                                                t[mp[v]][mp[vv]]+=mul*
C[ncard][i];
                                        }
                                }
                        }
                        break;
                case 3: // transposition
                //      cout << "TRANSPOSITION" << endl;
                        for (int i=0; i<ncard; i++) {
                                for (int j=0; j<ncard; j++) {
                                        vector<int> w=v;
                                        swap(w[i],w[j]);
                                        t[mp[v]][mp[w]]+=mul;
                                }
                        }
                        break;
                case 4: // top-to-random
```

```
                //      cout << "TOP-TO-BOTTOM" << endl;
                        t[mp[v]][mp[w]]+=mul;
                        for (int i=1; i<ncard; i++) {
                                swap(w[i-1],w[i]);
                                t[mp[v]][mp[w]]+=mul;
                        }
                        break;
                default:
                        break;
        }
}
```

Now, we provide the rest of the code, so that the interested readers may find it useful for extending the above work. We provide the most up-to-date version of the complete code here. Here is the main entry function:

```
int main() {
        srand ( time(NULL) );
        int n;

        for (int i=0; i<MAX_SIZE; i++) {
                C[i][0]=C[i][i]=1;
                for (int j=1; j<i; j++) C[i][j]=C[i-1][j]+C[i-1][j-1];
        }

        for (int type=0; type<MAX_TYPE; type++) {
                n=get_transition_single(ncard, type);

                // normalize
                for (int j=0; j<n; j++) {
                        double sum=0;
                        for (int k=0; k<n; k++) sum+=t[j][k];
                        for (int k=0; k<n; k++) t[j][k]/=sum;
                }

                for (int j=0; j<n; j++)
                for (int k=0; k<n; k++)
                        cache[type][j][k]=t[j][k];
                print(cache[type], n);
        }

        vector<int> list_type;
        for (int i=MIN_RANGE; i<=MAX_RANGE; i++)
                list_type.push_back(i);
```

```
        int sz_type=list_type.size();

        for (int t=1; t<(1<<sz_type); t++) {
                vector<int> list_cur;
                for (int c=0; c<sz_type; c++)
                        if(t&(1<<c)) list_cur.push_back(list_type[c]);
                int sz_cur=list_cur.size();
                cout << "----- ----- -----" << endl;
                cout << "[[" << t << "]]" << endl;
                print3(list_cur);

                int iter=0, ans_=INT_MAX, iter_cur=MAX_ITER;
                // vector<double> D_; vector<int> T_;
                map<int, multiset< pdt > > mp_;

                if(sz_cur==1) iter_cur=1;

                while(iter<iter_cur) {
                        // initialize state by transition matrix of
randomly chosen shuffle type
                        int type=list_cur[get_rand(0, sz_cur-1)];
                        for (int j=0; j<n; j++)
                        for (int k=0; k<n; k++) s[j][k]=cache[type][j][k];

                        vector<double> D;
                        vector<int> T;

                        int ct=0;
                        double cur;
                        while(1) {
                                if(ct>=MAX_ITER2) break;
                                cur=get_dist_uniform(s,n,0);
                                T.push_back(type);
                                D.push_back(cur);

                                cout << ct << " " << type << " " <<
setprecision(NUM_PRECISION) << cur << endl;

                                if(cur<EPS) break;

                                type=list_cur[get_rand(0, sz_cur-1)];
                                mul(s,cache[type],n);
                                ct++;
                        }
```

```
                        // print distance and type sequences
                        print2(D);
                        print2(T);

                        int ans=T.size();
                        ans_=min(ans_, ans);

                        mp_[ans].insert(make_pair(D, T));

                        cout << "(" << iter << ") " << ans <<
" -> " << ans_ << endl;
                        iter++;
                }
                cout << endl;
                cout << "min = " << ans_ << endl;
                multiset< pdt > st=mp_[ans_];

                for(multiset< pdt >::iterator it=st.begin();
it!=st.end(); it++)
                        print2((*it).second);

                cout << endl;

                get_stat(mp_);
        }

        return 0;
}
```

Here are some helper functions. This one returns the random integer between n and m:

```
int get_rand(int n, int m) {
        return rand()%(m-n+1) + n;
}
```

This function implements the multiply operator:

```
// multiply matrix s by matrix t and save result to s; n is a dimension
double mul(double s[MAX_DIM][MAX_DIM], double t[MAX_DIM][MAX_DIM], int n) {
        double ans=0;
        for (int i=0; i<n; i++) {
                for (int j=0; j<n; j++) {
                        u[i][j]=0;
                        for (int k=0; k<n; k++) u[i][j]+=s[i][k]*t[k][j];
                        ans+=fabs(u[i][j]-s[i][j]);
                }
```

```
        }

        for (int i=0; i<n; i++)
        for (int j=0; j<n; j++) swap(u[i][j],s[i][j]);
        ans/=(n*n); // average difference between
prev and curr in each entry
        return ans;
}
```

This function measures the distance to the uniform measure:

```
// get distance from uniform measure with metric
// type 0: total variation;
double get_dist_uniform(double s[MAX_DIM][MAX_DIM], int n, int type) {
        double ans=0;
        switch(type) {
                case 0:
                        for (int i=0; i<n; i++) ans+=fabs(s[0][i] - 1.0/n);
                        ans/=2;
                        break;
                default: break;
        }
        return ans;
}
```

There are several print routines:

```
// print matrix s
void print(double s[MAX_DIM][MAX_DIM], int n) {
        for (int i=0; i<n; i++) {
                for (int j=0; j<n; j++) {
                        if(j) cout << " ";
                        cout << setprecision(NUM_PRECISION)
<< fixed << s[i][j];
                }
                cout << endl;
        }
}

// print vector v
template <class T>
void print2(vector<T> v) {
        int sz=v.size();
        cout << "{";
        for (int i=0; i<sz; i++) {
                if(i) cout << ", ";
                cout << setprecision(NUM_PRECISION) << fixed << v[i];
```

```
        }
        cout << "}" << endl;
}


string list_name[5]={"CUT", "OVERHAND", "RIFFLE",
"TRANSPOSITION", "TOP-TO-BOTTOM"};
void print3(vector<int> v) {
        int sz=v.size();
        cout << "{";
        for (int i=0; i<sz; i++) {
                int cur=v[i];
                if(i) cout << ", ";
                cout << cur << " : " << list_name[cur];
        }
        cout << "}" << endl;
}
```

This one outputs the statistics:

```
void get_stat(map<int, multiset<pdt> > mp) {
        for (map<int, multiset<pdt> >::iterator it=mp.begin();
it!=mp.end(); it++) {
                cout << ") " << (*it).first << " ";
                multiset<pdt> st=(*it).second;
                cout << st.size() << endl;
                for (multiset<pdt>::iterator it2=st.begin();
it2!=st.end(); it2++) {
                        print2((*it2).second);
                }
        }
        cout << endl;

        cout << "{";
        bool ok=false;
        for (map<int, multiset<pdt> >::iterator it=mp.begin();
it!=mp.end(); it++) {
                if(ok) cout << ", ";
                cout << (*it).second.size();
                ok=true;
        }
        cout << "}" << endl;
}
```

Here is the header I used for my implementation:

```
#include<limits.h>
#include<time.h>
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<algorithm>
#include<iostream>
#include<vector>
#include<map>
#include<set>
#include<iomanip>
#include<utility>

#define MAX_DIM 5050
#define MAX_SIZE 20
#define MIN_RANGE 0
#define MAX_RANGE 4
#define MAX_TYPE 5
#define MAX_ITER 500
#define MAX_ITER2 100
#define NUM_PRECISION 3
#define EPS 1e-5

using namespace std;

int ncard=5;
map<vector<int>, int> mp;
double cache[MAX_TYPE][MAX_DIM][MAX_DIM], s[MAX_DIM][MAX_DIM],
t[MAX_DIM][MAX_DIM], u[MAX_DIM][MAX_DIM];

typedef pair< vector<double>, vector<int> > pdt;
```

7.2.2 Section 4

Here we provide the main portion of the code we used to run the above simulation. It is implemented in MATLAB. Here is the main function:

```
% 100 iterations of 5000 trials of each method using
the custom method as a test statstics
clear;
init;

for T=1:N
    for ITER=1:MAX_ITER
        num=nums{T};
        seq=seqs{T};
```

```
        result=zeros(1,ntrial);
        x_min=inf; x_max=0;
        for k=1:ntrial
            if mod(k,100)==0, fprintf('%d/%d :: %d/%d :: %f%% DONE\n',
T, N, ITER, MAX_ITER, 100*k/ntrial); end
            cur=shuffle([1:ncard], seq, theta);
            x=test_statistic(cur, 2);
            result(k)=x;
            x_min=min(x_min,x); x_max=max(x_max,x);
        end
        record=histc(result,[x_min:x_max])/ntrial;

        [h,p,st]=gof(result, ntrial, ncard, 2);
        cache_chi(T,ITER)=st.chi2stat;
        cache_p(T,ITER)=p;
    end
end
save(['dat_' num2str(N) '_' num2str(ntrial) '_' num2str(MAX_ITER) '.mat']);
```

Here is the helper function to initialize variables:

```
ncard=52;
ntrial=100000;
MAX_ITER=5;
theta=4/51;

nums={[7 0 0 0 0 0 0 0 0], [3 0 0 0 0 0 0 0 0],
[4 0 0 0 0 0 0 0 0], [0 0 0 50 0 0 0 0 0], [0 0 0 15 0 0 0 0 0],
[0 0 0 30 0 0 0 0 0], [3 0 1 1 0 0 0 0 0]};
seqs={getseq(nums{1}), getseq(nums{2}), getseq(nums{3}),
getseq(nums{4}), getseq(nums{5}), getseq(nums{6}),
[1 1 4 1 3]};

N=numel(nums);
cache_chi=zeros(N,MAX_ITER);
cache_p=zeros(N,MAX_ITER);
```

Here is the helper function to shuffle the deck using the standard ones:

```
function tmp=shuffle(cur, seq, theta)
% 1==riffle
% 2==trans
% 3==cut
% 4==strip
% 5==hindu
% 6==wash
```

```
% 7==pile
% 8==mongean
% 9==faro
    n=numel(seq);
    tmp=cur;
    for k=1:n
        idx=seq(k);
        if idx==1, tmp=riffle(tmp);
        elseif idx==2, tmp=trans(tmp);
        elseif idx==3, tmp=cut(tmp);
        elseif idx==4, tmp=strip(tmp,theta);
        % implement the rest
        elseif idx==5, tmp=hindu(tmp);
        elseif idx==6, tmp=wash(tmp);
        elseif idx==7, tmp=pile(tmp);
        elseif idx==8, tmp=mongean(tmp);
        elseif idx==9, tmp=faro(tmp);
        end
    end
end
```

Here is the helper function for test statistics; we implemented the standard test statistics. Also, customized ones can be used by implementing custom function.

```
function n=test_statistic(cur, idx)
    if idx==1, n=top_card(cur);
    elseif idx==2, n=custom(cur);
        %implement the rest
    elseif idx==3, n=descent_count(cur);
    elseif idx==4, n=posA(cur);
    end
end
```

Here is the helper function to test the goodness of fit. We used two of the most popular choices: χ-square and χ-normal.

```
function [h,p,st]=gof(cur, ntrial, ncard, idx)
    if idx==1, [h,p,st]=chi_uniform(cur, ntrial, ncard);
    elseif idx==2, [h,p,st]=chi_normal(cur);
    end
end
```

7.2.3 Section 5

Here I provide the code I used to get started on the proof:

```
#include <fstream>
```

```
#include <vector>
#include <list>
#include <map>
#include <set>
#include <deque>
#include <queue>
#include <stack>
#include <bitset>
#include <algorithm>
#include <functional>
#include <numeric>
#include <utility>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <cctype>
#include <string>
#include <cstring>
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <ctime>
using namespace std;

#define MAX 1000

typedef long long ll;

ll mat[MAX][MAX], res[MAX][MAX], tmp[MAX][MAX];

vector<int> perm(vector<int> & v, int x) {
  int sz=v.size();
  vector<int> ans;
  for (int i=sz-1; i>=0; i--) ans.push_back(v[i]);
  swap(ans[x], ans[x+1]);
  return ans;
}

int print(int n) {
  int ct=0;
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
```

```
      if(!res[i][j]) { cout << "0"; ct++; }
      else cout << "*";
    }
    cout << endl;
  }

  return ct;
}

int main() {
  int n; cin>>n;

  int f=1;
  for (int i=1; i<=n; i++) f*=i;


  // initialize matrix
  for (int i=0; i<f; i++)
    for (int j=0; j<f; j++)
      mat[i][j]=res[i][j]=0;

  map< vector<int>, int > mp;

  vector<int> cur;
  for (int i=0; i<n; i++) cur.push_back(i);
  int val=0;
  do {
    mp[cur]=val++;
  } while(next_permutation(cur.begin(), cur.end()));

  for (int i=0; i<n; i++) cur[i]=i;

  do {
    for (int i=0; i<n-1; i++) {
      vector<int> key=perm(cur, i);
      int x=mp[cur], y=mp[key];
      if(x!=y)
        res[x][y]=mat[x][y]=1;
    }
  } while(next_permutation(cur.begin(), cur.end()));

  for (int c=0; c<100; c++) {
    cout << "[" << c+1 << "]" << endl;
    print(f);
```

```
    for (int i=0; i<f; i++) {
      for (int j=0; j<f; j++) {
        tmp[i][j]=0;
        for (int k=0; k<f; k++)
          tmp[i][j]+=mat[i][k]*res[k][j];
      }
    }

    for (int i=0; i<f; i++)
      for (int j=0; j<f; j++)
        res[i][j]=tmp[i][j];
  }
  return 0;
}
```

References

[1] Trailing the Dovetail Shuffle to its Lair. P. Diaconis D. Bayer, Ann. Appl. Prob., 2(2):294-313

[2] Comparison Techniques for Random Walk on Finite Groups. P. Diaconis L. Saloff-Coste, Ann. Prob., 21 (4):2131-2156

[3] Comparison Theorems for Reversible Markov Chains. P. Diaconis L. Saloff-Coste, Ann. Appl. Prob, 3(3):696-730.

[4] Group Representations in Probability and Statistics. P. Diaconis, Lecture Notes-Monograph Series (1988).

[5] Finite Fourier Methods: Access to Tools. P. Diaconis, Probabalistic combinatorics, Proc. Symposia Appl. Math., B. Bollabos (ed), 44 Amer. Math. Soc. Providence, R.I., 171-194.

[6] Markov Chains and Mixing Times. Levin, David Asher, Yuval Peres, and Elizabeth Lee Wilmer. Amer Mathematical Society, 2009.

[7] Reversible Markov Chains and Random Walks on Graphs. Aldous, D. and Fill, J. Berkeley (2002).

[8] The random k-cycle walk on the symmetric group. Bob Hough. Preprint.

[9] `http://www.homepokertourney.com/docs/WSOP-Dealer-Guide-2011.pdf`

[10] Randomization time for the overhand shuffle. R. Pemantle. Journal of Theoretical Probability 2.1 (1989): 37-49.

[11] The Overhand Shuffle Mixes in $\Theta(n^2 \log n)$ Steps. Johan Jonasson, The Annals of Applied Probability , Vol. 16, No. 1 (Feb., 2006), pp. 231-243