Monte Carlo project: Approximating the permanent

Quentin Fortier, Thibaut Dubernet

Abstract

We study three different Monte Carlo methods to approximate the permanent of a matrix.

1 Introduction

Definition 1 (Permanent)**.** Let $A = (a_{i,j})$ be a $n \times n$ matrix.
The permanent $perm(A)$ is:

$$perm(A) = \sum_{\sigma \in \mathcal{S}_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

Where $\mathcal{S}_n$ denotes the symmetric group on $1, 2, ..., n$. In this paper we will only focus on computing the permanent of 0-1 matrices, i.e. with coefficients equal to 0 or 1.
The permanent looks very similar to the determinant (there is just a $(-1)^{\epsilon(\sigma)}$ missing). However, although the determinant can be computed in time $O(n^3)$ (using Gaussian elimination), computing the permanent is a much harder task: it is known to be #-**P** complete, even for 0-1 matrices [1]. The best known complexity for computing exactly the permanent is $O(n2^n)$, which is too much for most applications. We study faster approximation algorithms based on Monte Carlo methods, and more especially on importance sampling and Metropolis-Hastings algorithm.
Computing the permanent is a fundamental problem in statistical physics, for example in the monomer-dimer problem. It is also related, in computer science, to the number of perfect matchings in a bipartite graph:

Definition 2. A graph is *bipartite* if its vertices can be separated in two sets X, Y such that there is no edge between these sets.

Definition 3. A *$k-$matching* of a graph G is a set S of k edges of G such that no vertex of G is adjacent to two edges of S.
We say that a vertex is *matched* if it is adjacent to one edge in S.
A *perfect matching* is a matching such that every vertex is matched.

Indeed, if G is a bipartite graph with vertex sets $X = \{u_1, \dots, u_n\}$ and $Y = \{v_1, \dots, v_n\}$, we can define its $n \times n$ *adjacency matrix* $A = (a_{i,j})$ such that $a_{i,j} = 1$ if there is an edge between u_i and v_j and 0 otherwise. Then it is straightforward that $perm(A)$ is the number of perfect matchings of G.

2 Naïve method by rejection

The first method is maybe the easiest to implement: let $\sigma_1, \dots, \sigma_R$ be R permutations of $\mathcal{S}_n$ taken uniformly at random. Let $\prod(\sigma) = \prod_{i=1}^{n} a_{i,\sigma(i)}$. $\prod(\sigma)$ is a Bernoulli variable with paramater $\frac{perm(A)}{n!}$ so according to the law of large numbers:

$$\frac{1}{R}\sum_{i=1}^{R}\prod(\sigma_i) \underset{n\longrightarrow\infty}{\longrightarrow} \frac{perm(A)}{n!}$$

Therefore $\widehat{perm(A)} = \frac{n!}{R}\sum_{i=1}^{R}\prod(\sigma_i)$ is an estimator of $perm(A)$.
In practice, this algorithm performs badly. To see why, we can look at the variance of $n!\prod(\sigma)$:

$$var(n!\prod(\sigma)) = (n!)^2 var(\prod(\sigma)) = (n!)^2\frac{perm(A)}{n!}(1-\frac{perm(A)}{n!})$$
$$= perm(A)(n! - perm(A))$$

If for example $perm(A) = \frac{n!}{2}$, $var(n!\prod(\sigma)) = \frac{(n!)^2}{2}$ which is an enormous number.
Moreover, if $perm(A)$ is very small, for example if A is sparse (with a lot of 0), it will take a lot of time to find one *good* permutation, i.e. a permutation σ such that $\prod(\sigma) \neq 0$ and rejection performs very badly. This motivates the following section.

3 Importance sampling ([3], [4])

It would be interesting to sample from $\mathcal{S}_n$ with a probability p (not uniform) such that the probability to get a good permutation is high (this way we avoid the above problem). Let fix a probability p which support contains the good permutations and which can be "easily" computed. Then:

$$\frac{1}{R}\sum_{X\sim\ \mathcal{U}(\mathcal{S}_n)}\mathbb{1}_{Xgood} = \frac{1}{R}\sum_{Y\sim p}\frac{\mathbb{1}_{Ygood}}{p(Y)}\frac{1}{n!} \longrightarrow \frac{perm(A)}{n!}$$

So:

$$\frac{1}{R}\sum_{Y\sim p}\frac{\mathbb{1}_{Ygood}}{p(Y)} \longrightarrow perm(A)$$

where, for each sum, there are R independent samples.
The variance becomes:

$$var(\frac{\mathbb{1}_{Ygood}}{p(Y)}) = \mathbb{E}(\frac{\mathbb{1}_{Ygood}}{p(Y)^2}) - perm(A)^2$$
$$= \sum_{\sigma\in\mathcal{S}_n}\frac{\mathbb{1}_{\sigma good}}{p(\sigma)} - perm(A)^2$$

If $p(\sigma) \approx \frac{1}{perm(A)}$ for a good permutation σ the variance will be low (of course if $p(\sigma) = \frac{1}{perm(A)}$ we will not be able to compute $p(\sigma)$ since we don't know

$perm(A)$).
Now imagine we pick $\sigma(1)$, ..., $\sigma(n)$ sequentially. Instead of picking $\sigma(1)$ uniformly at random, we want to pick it so that the whole permutation is likely to be good. Of course, if possible, we will not pick $\sigma(1)$ such that $a_{1,\sigma(1)} = 0$. Moreover if we pick $\sigma(1)$, none of the $\sigma(i), i = 2 \dots n$ can be equal to $\sigma(1)$. So it makes sense to choose $\sigma(1)$ depending on the number of $a_{i,\sigma(1)}$ being equal to 1 (this is, intuitively, an estimation of the number of good permutations we may forbid by taking this $\sigma(1)$).
For example, if $s_j = \sum_{i=1}^{n} a_{i,j}$, we can take $\sigma(1)$ with probability:

$$\begin{cases} \propto \frac{1}{s_{\sigma(1)}-1} & \text{if } s_{\sigma(1)} \notin \{0,1\} \\ 1 & \text{if } s_{\sigma(1)} = 1 \\ \text{We report the permutation as not good if } s_{\sigma(1)} = 0 \end{cases}$$

If there are i, j, such that $s_i = s_j = 1$ we report the permutation as not good (we can't extend it into a good permutation). When $\sigma(1)$ is chosen, we set $a_{1,j}$ and $a_{j,\sigma(1)}$ to 0 in a temporary matrix and we choose $\sigma(2)$, and so on.
This gives us a probability, which support contains the good permutations and that we can compute easily: it is the product of the probabilities to choose $\sigma(1) \dots \sigma(n)$ [1].
We compared (Figure 8) the number of good permutations divided by the total number of permutation generated by importance sampling and rejection, for random matrices (we call *density* the probability that this random matrix has a 1 in an entry, i.e $\mathbb{P}(a_{i,j} = 1)$).

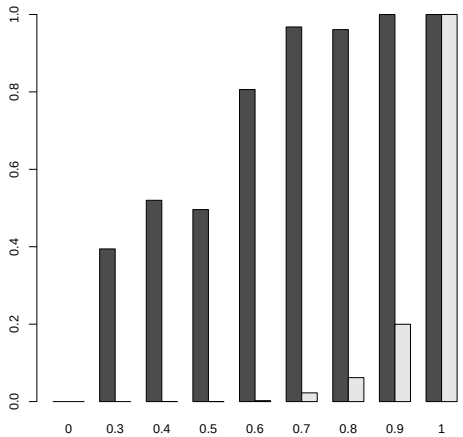

Figure 1: Proportion of good permutation generated by importance sampling (black) and rejection (white), with respect to the density of a random matrix

Figure 3 shows that the distribution used is almost uniform on the good permutations if the density is high, but this is not the case for lower density.

[1] One can also use an importance sampling ration estimator by rescaling the weights

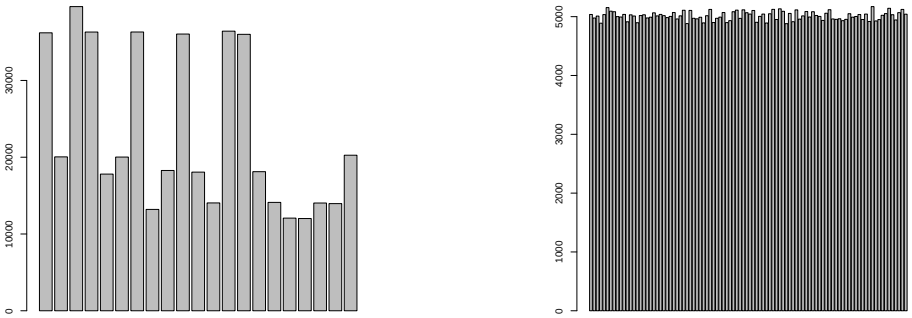

Figure 2: Number of samples (y-axis) of a good permutation (x-axis) for n=5. Left: density=0.6, Right: density=0.9

4 Metropolis-Hastings [5]

Here we show how we can devise a Markov chain to approximate the number of perfect matching in a bipartite graph G with two set of vertices of size n[1].
Let M_k be the set of all $k-$matchings of G, M_n is also the number of perfect matchings of G.
One idea, taken from [2] is to devise a Markov chain on perfect matchings of G such that its stationnary distribution is uniform. Unfortunately, in our case this problem is not substentially easier. However, we will show that it is easy to devise a Markov chain on $M_k \cup M_{k-1}$ with uniform stationnary distribution. Then we can write:

$$\mid M_n \mid = \frac{\mid M_n \mid}{\mid M_{n-1} \mid} \times \ldots \times \frac{\mid M_2 \mid}{\mid M_1 \mid} \times \mid M_1 \mid$$

$\mid M_1 \mid$ being the number of edges of G.
So if we have a large uniform sampling from $M_k \cup M_{k-1}$, we can approximate $\frac{\mid M_k \mid}{\mid M_{k-1} \mid}$ by the number of samples in M_k divided by the number of samples in M_{k-1}, and then we get an approximation of $\mid M_n \mid$.
We define on $M_k \cup M_{k-1}$ the following Markov chain: if we are in a state $m \in M_k \cup M_{k-1}$, take an edge $e = (u, v)$ in G uniformly at random and:

$$\begin{cases} \text{if m } \in M_k \text{ and } e \in m \text{ go to } m - e \\ \text{if m } \in M_{k-1} \text{ and } e \text{ is not in } m \text{ go to } m + e \\ \text{if m } \in M_{k-1}, u \text{ is matched by } e' \\ \quad \text{and } v \text{ is not matched, go to } m + e - e' \text{ (and the symmetric case)} \\ \text{Otherwise stay in the same state} \end{cases}$$

Here the proposal is symmetric and we want the stationnary distribution to be uniform, so the acceptance is always equal to 1.

[1] We saw in the introduction that this is equivalent to computing the permanent, and it is more convenient

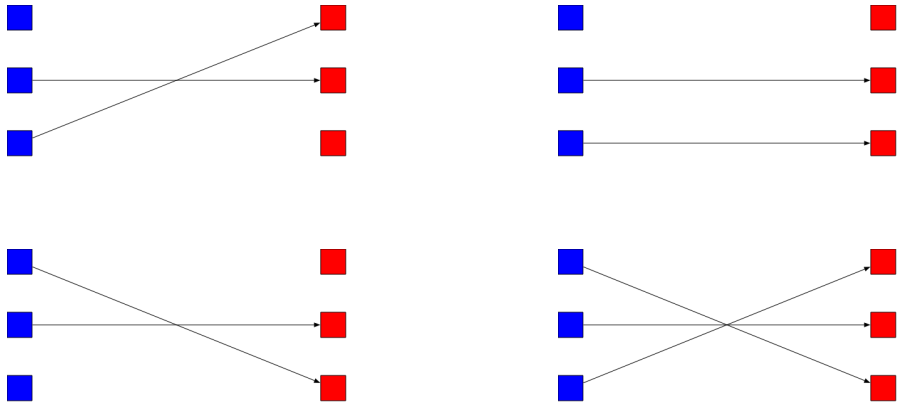

Figure 3: Four steps of the Markov chain on $M_2 \cup M_3$, with G the complete bipartite graph (i.e. the bipartite graph with all possible edges) with 6 vertices (only the edges in the state of the Markov chain are drawn).

We don't discuss how to generate the first matching, but this can be done easily with algorithms in graph theory.
Under some assumptions, [5] proves that this Markov chain is rapidly mixing: the number of iterations required is polynomial in n and the precision required (however the degree of the polynomial is high). One advantage of using a Markov chain is that we can easily adapt it for other distributions or other problems.

5 Comparisons

In Figure 1 we compare the variances of the different algorithms. Figures 5 and 7 show the relative error of these algorithms for a 10×10 and 17×17 matrix, respectively. As the size of matrices increases, the performances of the rejection method decreases, since the space becomes too large and a random permutation is very likely to be not good. It may look strange that the rejection method always performs better than the Metropolis Hastings algorithm, for high densities, but is explained by the fact that, if for example the matrix is full of 1 (the extremal case), the rejection algorithm consists just in directly computing the number of permutations, i.e. in computing $n!$, and we can't do faster.

Table 1: Variance for 7×7 random matrices with different density d. Each algorithm was run during 1 second.

	d = 0.6	d = 0.7	d = 0.8	d = 0.9
Importance	2.1	2.2	2.5	4.9
Rejection	344.0	161.6	209.0	901.7
Metropolis	3876.1	5765.6	27224.9	109330.8

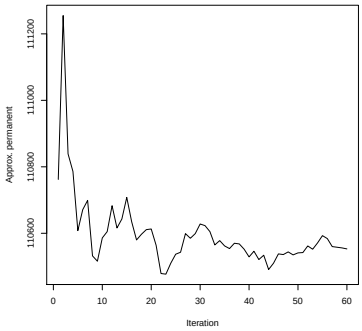

Figure 4: Evolution of the permanent approximation given by importance sampling, for a 10×10 matrix.

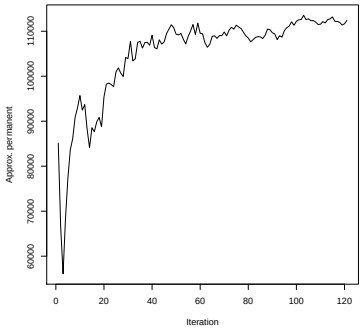

Figure 5: Evolution of the permanent approximation given by Metropolis Hastings, for a 10×10 matrix.

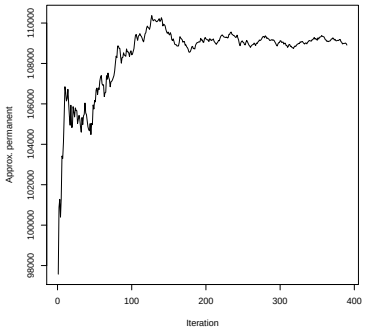

Figure 6: Evolution of the permanent approximation given by rejection, for a 10×10 matrix.

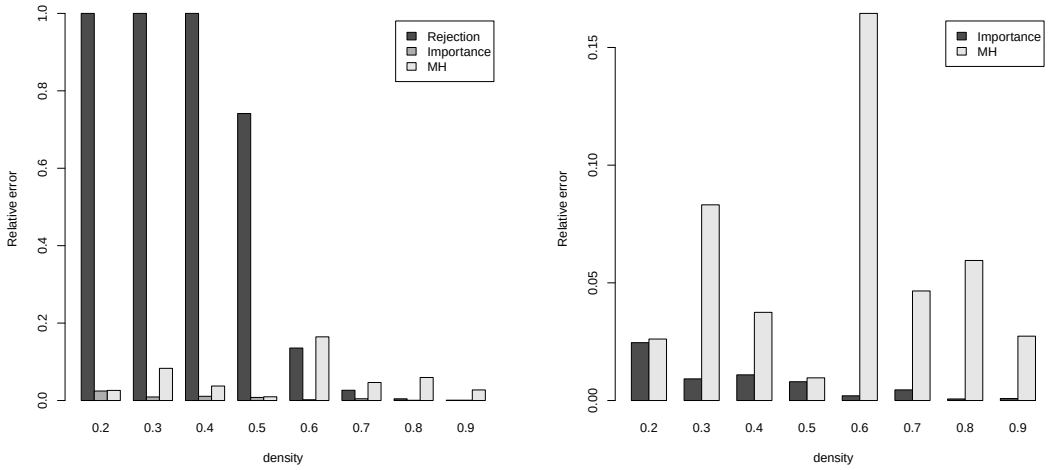

Figure 7: Relative error, for random matrices with different density and size 17×17. Each algorithm was run during 1 second.

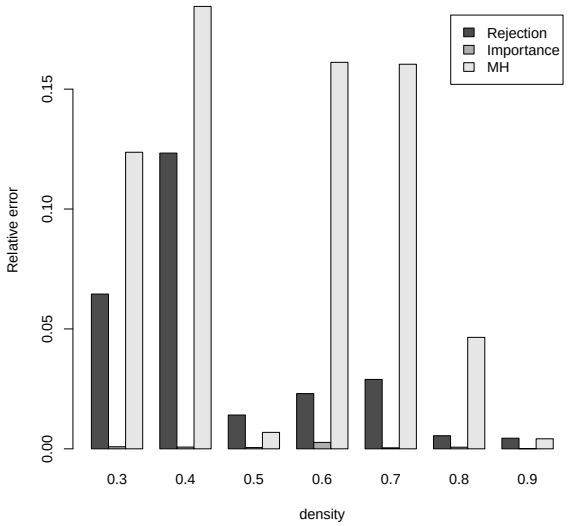

Figure 8: Relative error, for random matrices with different density and size 10×10. Each algorithm was run during 1 second.

6 Code

The code can be found at http://perso.ens-lyon.fr/quentin.fortier/MC/mc.html. The code we wrote is divided into two parts. The first (core) part is in oriented-object C++ and implements the three algorithms described above (we also tried to modify the Metropolis Hastings algorithms, to do a "mix" of Metropolis Hastings and importance sampling, i.e. moving preferentially to matchings with "a lot" of neighbors, but this doesn't improve significantly the performances). This code outputs a .R file with all relevant information about the simulation, and we used R to analyze the data.
The first part uses two librairies: OGDF for graph manipulation and Armadillo, a linear algebra library. The three algorithms are separated into three classes: `Importance_Matching` (Importance sampling), `MH_Matching` (Metropolis Hastings) and `Rejection_Matching`. Moreover, `Number_Matching` uses `MH_Matching` to simulate Markov chains on matching of different dimensions and deduce an approximation of the permanent. The code is quite long (more than 600 lines) so we decided not to write it entirely here. Every class has a function of the form:

```
get_approx(string file_name, int nSteps, int step_write,
    double max_second)
```

giving an approximation of the permanent after a simulation which ends either after `max_second` seconds or after `nSteps`, and writing information in the .R file named `file_name`. The initiatialization takes place in the constructor of the class which take (among others) as parameter the matrix (or the graph) for which we want to compute the permanent (or the number of perfect matchings).

The function `init` of `MH_Matching` sets the initial state (i.e. matching) of the Markov chain.
Compilation was done with Visual C++ 2008 and need both OGDF and Armadillo. However, we also provide for convenience an executable cmd.exe allowing to do basic tests without compiling anything (it takes the size of a matrix, a density, a number of seconds and outputs the corresponding simulation results and write basic information in files imp.r, rej.r, mh.r). The corresponding source code is in the `main` of cmd.cpp and can be used to understand how the whole code works.

References

[1] Leslie G. Valiant. The Complexity of Computing the Permanent. Theoretical Computer Science. 1979.

[2] M. Jerrum. A very simple algorithm for estimating the number of k-colorings of a low-degree graph. http://www.lfcs.inf.ed.ac.uk/reports/94/ECS-LFCS-94-290/ECS-LFCS-94-290.ps.

[3] Liu, Jun S., Monte Carlo Strategies in Scientific Computing (2001) Springer Series in Statistics, Springer Verlag, pp 90-92.

[4] Beichl, I., Sullivan, F. (1999). Approximating the permanent via importance sampling with application to the dimer covering problem. Journal of Computational Physics.

[5] Jerrum, M., Mathematical foundations of the Markov chain Monte Carlo method, ProbabilisticMethods for Algorithmic DiscreteMathematics (Habib, M., McDiarmid, C., RamirezAlfonsin, J., Reed, B., eds.), Springer, Berlin, 1998.