The Classical Complexity of Boson Sampling : A multi-core CPU implementation

Manan Vaswani

Level I
40cp project

Supervisor: Dr. Raphaël Clifford
Date: 6th May, 2019

Acknowledgement of Sources

Contents

1 Introduction

2 Background

3 Preliminaries

3.1 Binary Gray Code

3.2 Permanent of a matrix

Computing the permanent of large matrices is one of the key parts of the Boson Sampling problem, as will be discussed in detail while explaining the problem in the subsequent sections of the paper.

3.2.1 Definition

Definition 3.1. *[1] The permanent of an $n \times n$ matrix $A = (a_{ij})$ is defined as*

$$\operatorname{Per} A = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i\sigma(i)} \tag{1}$$

where the sum is over all elements of the symmetric group S_n i.e. over all permutations of the numbers in $[n] = \{1, 2, ..., n\}$

Example 3.2. For a 2×2 matrix, the permanent is calculated as follows

$$\operatorname{Per} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad + bc \tag{2}$$

Example 3.3. For a 3×3 matrix, the permanent is calculated as follows

$$\operatorname{Per} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = aei + bfg + cdh + ceg + bdi + afh \tag{3}$$

One can observe that the definition of the permanent is similar to the more commonly used determinant function, differing in the fact that the permanent definition lacks the alternating signs. An important property to note about the permanent function is that it is invariant to transposition i.e. $\operatorname{Per} A = \operatorname{Per} A^T$ [2].

3.2.2 Computing the permanent

Valiant showed that the problem of computing the permanent of a matrix is in the class #P-complete which implies that it is unlikely to have a polynomial time algorithm which implies that it is unlikely to have a polynomial-time algorithm [3]. The naive algorithm obtained by directly translating the formula into an algorithm would run in $\mathcal{O}(n!n)$ time.

A significant improvement on the naive approach, Rysers algorithm uses a variant of the inclusion-exclusion principle and can be evaluated in $\mathcal{O}(n^2 2^n)$ time [2]. Nijenhuis and Wilf sped this up to $\mathcal{O}(n2^n)$ time by iterating over the sum in Gray Code order [4].

Another formula that is as fast as Rysers was independently derived by Balasubramanian[5], Bax[6], Franklin and Bax[7], and Glynn[8], all using different methods. We shall henceforth refer to this as Glynn's formula and it is described as follows.

Let $M = (m_{ij})$ be an $n \times n$ matrix with $m_{ij} \in \mathbb{C}$, then

$$\operatorname{Per} M = \frac{1}{2^{n-1}} \sum_{\delta} \left(\prod_{k=1}^{n} \delta_k \right) \prod_{j=1}^{n} \sum_{i=1}^{n} \delta_i m_{ij} \tag{4}$$

where $\delta \in \{-1, 1\}^n$ with $\delta_1 = 1$. Hence there are 2^{n-1} such values for δ.

Implementing the formula as is would require $\mathcal{O}(2^n n^2)$ time. However, iterating over the δ arrays in Gray code order reduces it to $\mathcal{O}(n2^n)$ time.

In order to exploit this trick, let v_j be the symbol used to denote the innermost sum i.e. $v_j = \sum_{i=1}^{n} \delta_i m_{ij}$. If δ is iterated through in Gray code order, then one can notice that the terms $\{v_j, j \in [n]\}$ can actually be computed in $\mathcal{O}(n)$ time for a given value of δ rather than $\mathcal{O}(n^2)$ which is how long it would take if done naively. This is because successive elements of δ differ in only one position, so each successive v_j differs only by the addition or subtraction of some m_{ij} where i is the position at which δ was last changed. Hence, the product of v_j terms can now be calculated in $\mathcal{O}(n)$ time.

4 Describing the paper and specifying the problem

Cliiford and Clifford [9] gave a study and analysis of the classical complexity of the exact Boson Sampling problem and proposed an algorithm that is significantly faster than previous algorithms for the problem. The algorithm is simple to implement, and is able to solve the Boson Sampling problem for system sizes much greater than quantum computing systems currently available, which reduces the likelihood of achieving quantum supremacy in the context of Boson Sampling in the near future rephrase last bit.

4.1 Explaining the problem

A summary of the problem in purely mathematical terms is as follows.

Let m and n be positive integers. Consider all possible multisets[1] of size n with elements in $[m]$, where $[m] = \{1, ..., m\}$. Let $z = [z_1, z_2, ..., z_n]$ be an array representation of such a multiset, with its elements in non-decreasing order. In other words, z is an array of n integers taken from $[m]$ (with repetition) and arranged in non-decreasing order. Define $\Phi_{m,n}$ to be the set of all distinct values that z can take. Define $\mu(z) = \prod_{j=1}^{m} s_j!$ where s_j is the multiplicity of j in the array z i.e. the number of times it appears in z.

[1] A multiset is a special kind of set in which elements can be repeated

$A = (a_{ij})$ is a complex-valued $m \times n$ matrix constructed by taking the first n columns of a given $m \times m$ Haar random unitary matrix. For each z, build an $n \times n$ matrix A_z where the k^{th} row of A_z is the z_k^{th} row in A, for $k = 1, ..., n$. Finally, define a probability mass function over $\Phi_{m,n}$ as

$$q(z) = \frac{1}{\mu(z)} \left|\text{Per}\, A_z\right|^2 = \frac{1}{\mu(z)} \left| \sum_{\sigma} \prod_{k=1}^{n} a_{z_k \sigma_k} \right|^2, \quad z \in \Phi_{m,n} \tag{5}$$

where Per A_z is the permanent of A_z and $\pi[n]$ is the set of all permutations of $[n]$.

The computing task is to simulate random samples from the above pmf $q(z)$.

4.1.1 Calculating the size of the sample space

The size of the sample space $\Phi_{m,n}$ can be calculated using the 'stars and bars' technique from combinatorics [10]. In our problem, we have n 'stars' representing the elements of the array z, and m 'buckets' representing all the values in $[m]$. Recall that z is a sorted array representation of a multiset, so multiple 'stars' can be in one 'bucket'. Since there are m 'buckets', we need $m - 1$ 'bars' to divide the 'stars' into 'buckets'. Therefore, from a total of $m - 1 + n$ objects, we need to pick n of these to be the 'stars'. There are $\binom{m+n-1}{n}$ ways to do this. Hence, there are $\binom{m+n-1}{n}$ possible values of z.

5 The Boson Sampling Algorithm

5.1 The naive approach

Translating the formula above (5) directly to an algorithm would require $\mathcal{O}(\binom{m+n-1}{n} n2^n)$ time to evaluate. The $\binom{m+n-1}{n}$ term comes from the size of the sample space of the pmf, and calculating the value of the permanent using the fastest known methods takes $\mathcal{O}(n2^n)$ time (Section 3.2.2). With $m = \mathcal{O}(n^2)$ as suggested in Section????, the total running time is $\mathcal{O}(\binom{n^2+n-1}{n} n2^n) =$???. Hence, even for relatively small values of n ((give some actual numbers), computing the Boson Sampling problem would be intractable for even very powerful supercomputers.

In the literature [9], two new algorithms are proposed for exact Boson Sampling. These are referred to as Algorithm A and Algorithm B, and both provide a significant speed up on the naive algorithm. The following subsections summarise the approach taken to obtain them.

5.2 Algorithm A

The approach to Algorithm A starts by expanding the sample space to a much larger size, which seems counterintuitive at first, but it allows us to express the pmf (Equation 5) in a form that is much easier to compute. The sample space is expanded to the space of all arrays $\mathbf{r} = (r_1, r_2, ..., r_n)$ where each element r_k is in $[m]$, which implies that we are considering a distribution on the product space $[m]^n$. It is stated and proved that sampling from $q(\mathbf{z})$ is equivalent to sampling from the pmf

$$p(\mathbf{r}) = \frac{1}{n!}\left|\text{Per}\,A_{\mathbf{r}}\right|^2 = \frac{1}{n!}\left|\sum_{\sigma}\prod_{i=1}^{n} a_{r_i\sigma_i}\right|^2, \quad \mathbf{r} \in [m]^n \tag{6}$$

where as before, σ is the set of all permutations of $[n]$.

This method requires $p(\mathbf{r}) = p(r_1, ..., r_n)$ to be rewritten as a product of conditional probabilities using the chain rule i.e.

$$p(\mathbf{r}) = p(r_1)p(r_2|r_1)p(r_3|r_1, r_2)...p(r_n|r_1, r_2, ..., r_{n-1}) \tag{7}$$

We first sample r_1 from $p(r_1), r_1 \in [m]$. Then for $k = 2, ..n$, we sample r_k from the conditional pmf $p(r_k|r_1, r_2, ..., r_{k-1})$ with $r_1, ..., r_{k-1}$ fixed. After sampling all values of r_k, sort $(r_1, r_2, ..., r_n)$ in non-decreasing order, and that results in the array representation of a multiset sampled from the Boson Sampling distribution $q(\mathbf{z})$. In order to obtain the conditional probabilities, a formula for the joint pmf of the leading subsequences of $(r_1, ..., r_k)$ is given in Lemma 1 of the paper, and proved using arithmetic techniques and facts about probability measures. The formula in Lemma 1 is as follows:

$$p(r_1, ..., r_k) = \frac{(n-k)!}{n!}\sum_{c\in\mathcal{C}_k}\left|\text{Per}\,A^c_{r_1,...,r_k}\right|^2, \quad k = 1, ..., n \tag{8}$$

where $\mathcal{C}_k$ is the set of $k-$combinations taken without replacement from $[n]$ and $A^c_{r_1,...,r_k}$ is the matrix formed by taking only columns $c \in \mathcal{C}_k$ of the rows $(r_1, ..., r_k)$ of A.

Notice in equation 7, we need to sample r_k from the conditional probability distribution $p(r_k|r_1, ..., r_{k-1})$. This can be rewritten as $p(r_1, ..., r_k)/p(r_1...r_{k-1})$. Since r_k does not appear in the denominator, in order to sample r_k from the conditional pmf, we can equivalently sample from the pmf proportional to the numerator, as $(r_1, ...r_{k-1})$ are fixed, known values at this stage in the algorithm. Therefore, the formula is Lemma 1 is used to calculate the conditional pmfs at each stage, giving way to the following algorithm.

Algorithm 1 Boson Sampler: Single sample $\mathbf{z}$ from $q(\mathbf{z})$ in $\mathcal{O}(mn3^n)$ time

```
Require: $m, n \in \mathbb{Z}_+$; $A$ formed by first $n$ columns of $m \times m$ Haar random unitary
    matrix
$\mathbf{r} \leftarrow \emptyset$
for $k \leftarrow 1 \to n$ do
    $w_i \leftarrow \sum_{c\in\mathcal{C}_k}\left|\text{Per}\,A^c_{(\mathbf{r},i)}\right|^2, i \in [m]$
    $x \leftarrow \text{Sample}(w)$
    $\mathbf{r} \leftarrow (\mathbf{r}, x)$
$\mathbf{z} \leftarrow \text{IncSort}(\mathbf{r})$
return $\mathbf{z}$
```

The correctness of the algorithm is clear as it is simply an evaluation of the required pmf using the chain rule of probability. The literature gives a mathematical proof to derive the runtime, however here we shall give an informal explanation to obtain the runtime using the algorithm above. The runtime is dominated by the **for** loop in lines 2 to 6, as the sorting step in line 7 takes only $\mathcal{O}(n \log n)$ using the fastest methods for sorting (insert reference). In each iteration of the **for** loop, we first construct the conditional pmf on the sample space $[m]$. This is represented by a weighted array $\mathbf{w}$. This involves calculating the permanents of $|\mathcal{C}_k|$ $k \times k$ matrices for each $i \in [m]$, where k is the loop index variable. Sampling from this distribution (line 4) takes $\mathcal{O}(m)$ time [11], but this too is dominated by the calculations in line 3, so we can ignore it. Therefore, the total time taken in the k^{th} iteration is $\mathcal{O}(m\binom{n}{k}k2^k)$, as $|\mathcal{C}_k| = \binom{n}{k}$ and calculating the permanent of a $k \times k$ matrix takes $\mathcal{O}(k2^k)$ using the fastest methods. So for n iterations,

$$\sum_{k=1}^{n} mk2^k \binom{n}{k} = m\frac{2}{3}n3^n = \mathcal{O}(mn3^n) \tag{9}$$

Therefore the total time taken is $\mathcal{O}(mn3^n)$. In terms of space complexity, we only need to store one permanent calculation at a time, and the weight array for each pmf is of size m. These can also be stored only one at a time, so $\mathcal{O}(m)$ additional space is required.

5.3 Algorithm B

The main theorem of the paper(Theorem 1) states that the time complexity of the Boson Sampling problem is $\mathcal{O}(n2^n + \text{poly}(m, n))$ where $\text{poly}(m, n) = \mathcal{O}(mn^2)$, with $\mathcal{O}(m)$ additional space required. This is achieved using the second algorithm proposed in the paper i.e. Algorithm B.

Algorithm B makes use of the Laplace Expansion [1] of permanents to obtain a significant speed up. The Laplace expansion allows us to calculate the permanent of a matrix using its permanent minors (the permanent of a submatrix with a column and a row removed). For any $k \times k$ matrix $B = (b_{i,j})$,

$$\text{Per}\, B = \sum_{l=1}^{k} b_{k,l}\, \text{Per}\, B^{\diamond}_{k,l}, \tag{10}$$

where $B^{\diamond}_{k,l}$ is the submatrix of B with row k and column l removed. Hence the permanent of B can be calculated in $\mathcal{O}(k)$ steps provided the permanent minors i.e. the values $\{\text{Per}\, B^{\diamond}_{k,l}\}$ are known.

Computing $\{\text{Per}\, B^{\diamond}_{k,l}\}$ by calculating each of the permanents independently would collectively take $\mathcal{O}(k^2 2^k)$ time as there are k permanents to compute and to compute the value of each one takes $\mathcal{O}(k2^k)$ time. Lemma 2 of the paper states that the collection $\{\text{Per}\, B^{\diamond}_{k,l}\}$ can be collectively evaluated in $\mathcal{O}(k2^k)$ time and $\mathcal{O}(k)$ extra space. using Glynn's formula to evaluate a permanent minor for a given value of l, we have

$$\text{Per}\, B^{\diamond}_{k,l} = \frac{1}{2^{k-2}} \sum_{\delta} \left(\prod_{i=1}^{k-1} \delta_i \right) \prod_{j \in [k] \setminus l} v_j(\delta), \quad l \in [k], \tag{11}$$

where $\delta \in \{-1,1\}^{k-1}$ with $\delta_1 = 1$ and $v_j(\delta) = \sum_{i=1}^{k-1} \delta_i b_{ij}$. The usual trick of iterating over the values of δ in Gray code order is required to compute $\operatorname{Per} B^{\diamond}_{k,l}$ in $\mathcal{O}(k2^k)$ time for each value of l, but this would still have to be repeated k times to cover all $l \in [k]$. The second and more novel trick used to achieve an additional speed up is used while calculating the products $\prod_{j\in[k]\setminus l} v_j(\delta)$. Observe that for each l, $\prod_{j\in[k]\setminus l} v_j(\delta) = \prod_{j\in[k]} v_j(\delta)/v_l(\delta)$. However, we cannot compute this method to compute the partial product as it would not work for cases when $v_l(\delta) = 0$. Instead, we first precompute two arrays $\mathbf{f}$ and $\mathbf{b}$ containing the forwards and backwards cumulative products of $\mathbf{v}(\delta) = (v_j(\delta), j \in [k])$ respectively as follows,

$$f_i = \prod_{j=1}^{i} v_j(\delta), \quad f_0 = 1, \quad b_i = \prod_{j=i}^{k} v_j(\delta), \quad b_{k+1} = 1 \quad i \in [k]. \tag{12}$$

Then each partial product $\prod_{j\in[k]\setminus l} v_j(\delta)$ can be expressed as a product of two terms from $\mathbf{f}$ and $\mathbf{b}$, $f_{l-1}b_{l+1}$ for any $l \in [k]$. Since it takes $\mathcal{O}(k)$ time to compute the forward and backward cumulative product arrays and $\mathcal{O}(1)$ time to obtain each partial product $\prod_{j\in[k]\setminus l} v_j(\delta)$, for each value of δ, the total time taken to simultaneously calculate the values of $\{\operatorname{Per} B^{\diamond}_{k,l}\}$ is $\mathcal{O}(k2^k)$. Additional space required to store the partial products is $\mathcal{O}(k)$.

Coming back to formulating the second Boson Sampling algorithm, the sample space is now expanded even further with an auxiliary array $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$ where $\alpha \in \pi(n)$, which is the set of permutation of $[n]$. The approach is similar to that of Algorithm A in that the goal is to create a succession of pmfs for leading subsequences of $\mathbf{r}$ and then progressively sampling r_i from conditional pmfs on $(r_1, ..., r_{i-1})$. The conditioning variable α helps us reformulate the pmf in such a way that we can use the permanent minors that we are able to calculate efficiently, while still ensuring that it is equivalent to sampling from the pmf $p(\mathbf{r})$ in equation 6. Define

$$\phi(r_1, ...r_k|\alpha) = \frac{1}{k!}\left|\operatorname{Per} A^{[n]\setminus\{\alpha_{k+1},...,\alpha_n\}}_{r_1,...r_k}\right|^2, \quad k = 1, ..., n-1. \tag{13}$$

In Lemma 3 of the paper, it is shown that $p(\mathbf{r}) = \mathbb{E}_\alpha\{\phi(\mathbf{r}|\alpha)\}$, which is the expectation taken over α, uniformly distributed over $\pi(n)$ for fixed $\mathbf{r}$. The chain rule of expectation is used to prove the equality, and this full proof is provided in the literature. As in Algorithm A, a chain of conditional pmfs as follows is used,

$$\phi(\mathbf{r}|\alpha) = \phi(r_1|\alpha)\phi(r_2|r_1,\alpha)\phi(r_3|r_1,r_2,\alpha)...\phi(r_n|r_1,...r_{n-1},\alpha), \tag{14}$$

where $\phi(r_k|r_1,...r_{k-1},\alpha) = \phi(r_1, r_2, ...r_k|\alpha)/\sum_{r_k}\phi(r_1,...r_k|\alpha)$, using conditional probabilities and the law of total probability. The algorithm starts off by sampling r_1 from the pmf $\phi(r_1|\alpha)$. Then for stages $k = 2, ..., n$, r_k is sampled from $\phi(r_k|r_1,...r_{k-1},\alpha)$ with $(r_1, ..., r_{k-1})$ fixed. Since the terms in the denominator are fixed, known values, sampling from $\phi(r_k|r_1, ...r_{k-1}$ is equivalent to sampling from the numerator, which can be evaluated by taking advantage of the Laplace Expansion of permanents. At stage n, the array $(r_1, ..., r_n)$ will have been sampled from $p(\mathbf{r})$. Sorting $(r_1, ..., r_n)$ in non-decreasing

Algorithm 2 Boson Sampler: Single sample $\mathbf{z}$ from $q(\mathbf{z})$ in $\mathcal{O}(n2^n + \text{poly}(m, n))$ time

```
Require: $m, n \in \mathbb{Z}_+$; $A$ formed by first $n$ columns of $m \times m$ Haar random unitary
    matrix
 1: $\mathbf{r} \leftarrow \emptyset$                                      ▷ Empty array
 2: $A \leftarrow \text{Permute}(A)$                         ▷ Randomly permute columns of $A$
 3: $w_i \leftarrow |a_{i,1}|^2, i \in [m]$                    ▷ Make indexed weighted array $w$
 4: $x \leftarrow \text{Sample}(w)$                                ▷ Sample index $x$ from $w$
 5: $\mathbf{r} \leftarrow (\mathbf{r}, x)$                                    ▷ Append $x$ to $\mathbf{r}$
 6: for $k \leftarrow 2$ to $n$ do
 7:     $B^{\diamond}_k \leftarrow A^{[k]}_{\mathbf{r}}$
 8:     Compute $\{\text{Per}\, B^{\diamond}_{k,l}, l \in [k]\}$                 ▷ From Lemma 2
 9:     $w_i \leftarrow \left|\sum_{l=1}^{k} \text{Per}\, B^{\diamond}_{k,l}\right|^2, i \in [m]$     ▷ Using Laplace Expansion
10:     $x \leftarrow \text{Sample}(w)$
11:     $\mathbf{r} \leftarrow (\mathbf{r}, x)$
12: $\mathbf{z} \leftarrow \text{IncSort}(\mathbf{r})$                     ▷ Sort $\mathbf{r}$ in non-decreasing order
13: return $\mathbf{z}$
```

order gives $\mathbf{z}$, the array representation of the multiset sampled from the Boson Sampling distribution $q(\mathbf{z})$.

Correctness holds as Algorithm B samples from successive conditional pmfs, which is equivalent to sampling from $\phi(r|\alpha)$ due to the chain rule of probability (Equation 14), for a given α. Since α is uniformly distributed over the sample space $\pi(n)$, the algorithm equivalently samples from $\mathbb{E}_\alpha\{\phi(\mathbf{r}|\alpha)\}$ which is shown to be equal to $p(r)$ in Lemma 3.

Once again, we will analyse the time complexity of the algorithm by going through the actual steps of the algorithm rather than the more mathematical explanation given in [9]. The time complexity of the algorithm is dominated by the **for** loop as expected. Permuting the columns of A (Line 2) can be carried out in $\mathcal{O}(n)$ time using the Knuth shuffling algorithm [12]. Lines 3 and 4 take $\mathcal{O}(m)$ time, in order to generate the weight array representing the pmf and sample from it. In the **for** loop from lines 6 to 11, computing the permanent minors (Line 8) takes $\mathcal{O}(k2^k)$ time in the k^{th} iteration, as discussed previously. Subsequently, the weight array in Line 9 is computed in $\mathcal{O}(mk)$ time as it is an array of size m, with each element taking $\mathcal{O}(k)$ time to compute as it is a sum of k known terms. Sampling from this pmf takes $\mathcal{O}(m)$ time. The final sorting step outside the loop, in Line 12 takes $\mathcal{O}(n \log n)$ time, so it is dominated by the loop. Each iteration of the loop takes $\mathcal{O}(k2^k) + \mathcal{O}(mk)$ time in total. So for all iterations, this takes

$$\sum_{k=2}^{n} \mathcal{O}(k2^k) + \mathcal{O}(mk) = \mathcal{O}(n2^n) + \mathcal{O}(mn^2) \tag{15}$$

time.

The space is dominated by the weight array in Line 9, which is of size $\mathcal{O}(m)$.

Hence, the Boson Sampling Problem can be run in $\mathcal{O}(n2^n + \text{poly}(m, n))$ time using Algorithm B.

6 Implementation

In this section, we will discuss the actual implementation of Algorithm B described above. The paper by Clifford and Clifford which demonstrated the two new algorithms is still relatively new, which explains why there have not been any open-source large scale benchmark tests for actual implementations of the algorithm.

6.1 Original R Implementation

At the time of writing this paper, the only open-source implementation is at [13], by the authors of the paper. The implementation is in written as a package written in R and is freely available to download and use from CRAN. This package, named 'Boson Sampling' contains an implementation of Algorithm B, along with the required functions that it depends on. The relevant functions are described below.

`bosonSampler`

The package provides a function '`bosonSampler(A, sampleSize, perm=FALSE)`', which implements the Boson Sampling Algorithm B from [9]. As the function definition suggests, it takes three arguments

- `A`: The first n columns of an $m \times m$ random unitary matrix.
- `sampleSize`: The number of independent sample values to be taken from the distribution, where each value is a vector of size n.
- `perm`: Takes the value `TRUE` if the permanents and pmfs associated to each sample are required to be returned. By default, `perm=FALSE` unless specified otherwise.

Note that we do not need to explicitly provide the values m and n as the algorithm suggests since those values are obtained by simply checking the dimensions of the input $m \times n$ matrix using the `ncol(A)` and `nrow(A)` functions from R. This implementation allows us to obtain multiple samples from the Boson Sampling distribution using the same input matrix A by repeatedly looping over the algorithm the desired number of times. One notable difference is that the input matrix A is transposed right at the start, with the reason being that R stores matrices in column order so transposing the matrix makes operations easier. When accessing elements of A, the indices have been adjusted accordingly. Semantically, the function implements the algorithm exactly as it is, which ensures its' correctness (provided the inputs received and additional functions used are also correct).

`cxPermMinors`

The package provides a set of functions for computing permanents of matrices: `cxPerm(A)`, `rePerm(B)`, `cxPermMinors(C)`. Algorithm B requires the values

$\{\mathrm{Per}\, B^{\diamond}_{k,l}, l \in [k]\}$ to be computed in each iteration of the loop, and in order to do this, the bosonSampler function makes a call to the cxPermMinors function. This function takes an input matrix A of size $k \times k-1$. Note that this is actually the transpose of the $B^{\diamond}_{k}$ in Line 7 of the algorithm, since the original matrix in bosonSampler was transposed. cxPermMinors is written in C++ and is integrated into the R package using 'RCpp', which is another package in R [14]. The package facilitates mapping R datatypes to C++ equivalents and vice-versa. The function also extensively uses the Armadillo package from C++[15], which is a linear algebra library providing efficient functions and classes for vectors, matrices, and operations on them. This function calculates the permanent minors using Glynn's algorithm [8] using the technique of iterating over Gray codes, by Nijenhuis and Wilf[4] to iterate over Gray codes , and the trick of calculating forward and backward cumulative products described in Section 5.3.

randomUnitary

The function randomUnitary(size) takes in an integer size as an input, and returns a size × size complex-valued random unitary matrix.

Example

The function was run in the R console to produce 5 samples with $n = 10$ and $m = 100$.

```
> library(BosonSampling)        # load the BosonSampling package
> set.seed(7)        # set the random seed in R
> n <- 10
> m <- 100
> A <- randomUnitary(m)[,1:n]        #construct random unitary matrix
> valueList <- bosonSampler(A, sampleSize = 5)$values
> #run Boson Sampling algorithm
> apply(valueList, 2, sort) #sort each sample
      [,1] [,2] [,3] [,4] [,5]
 [1,]    6    2    6   15    5
 [2,]    6    5    8   15   15
 [3,]   24    5    8   24   23
 [4,]   36   15   30   45   24
 [5,]   54   16   32   60   44
 [6,]   61   52   35   65   50
 [7,]   77   68   41   77   62
 [8,]   78   69   46   80   77
 [9,]   78   81   50   80   89
[10,]   79   99   52   96   95
```

6.2 Implementation in C++

While choosing a programming language to implement a more efficient version of the Boson Sampling algorithm, performance was the key deciding factor. The language chosen was C++ due to its reputation for speed in mathematical problems, ease of use especially for multi-threaded programs, and the availability of highly efficient linear algebra libraries such as Armadillo. As with the *R* implementation, three core functions are required for the algorithm: a Boson Sampler (which runs the actual algorithm), a random unitary matrix generator, and a function to calculate the permanent minors of a matrix.

Armadillo Library: In the implementation, we make extensive use of the C++ Armadillo library[15]. Armadillo is a high quality linear algebra library that provides us with efficient implementations of matrix and vector operations. It greatly simplifies operations such as adding vectors, calculating cumulative products, and accessing matrix columns to name a few. Armadillo supports complex numbers as well, which was a necessary requirement for Boson Sampling. In addition, it supports the use of OpenMP for parallelisation, and automatically uses it in some cases to speed up computationally expensive operations.

6.2.1 Generating a random unitary matrix

The first step carried out was to create a function `randomUnitary(int m)` to generate a complex-valued random unitary matrix A which will be passed as an argument to the Boson Sampler. In order to generate a Haar random unitary matrix, the following algorithm from [16] was used. QR Decomposition decomposes a given matrix A into an orthogonal matrix Q and a right triangular matrix R such that $Q * R = A$. Armadillo provides the all the required supplementary functions used in the implementation, which made the C++ implementation straightforward. Since this function is run only once as a preprocessing step before the Boson Sampling, extra steps to optimise or parallelise it were not taken. Firstly, because the primary aim of the project is to optimise the implementation of the Boson Sampling algorithm itself, which receives the random unitary matrix as an input, and secondly, it is already a highly efficient implementation since all the functions used are from Armadillo so have already been optimised, as

Algorithm 3 Random Unitary: Generate an $m \times m$ complex-valued random unitary matrix

```
Require: $m \in \mathbb{Z}_+$
$A \leftarrow$ createRandomMatrix($m$)
$Q, R \leftarrow$ QRDecomposition($A$)
$R_{\text{diag}} \leftarrow$ Sign (Real (Diagonal ($R$)))
$U \leftarrow Q * R_{\text{diag}}$
return $U$
```

mentioned before. The object returned by `randomUnitary(int m)` is an $m \times m$ random unitary armadillo matrix.

6.2.2 Computing the Permanent Minors

Line 8 of the algorithm requires the permanent minors of a given $k-1 \times k$ matrix to be computed. We use a separate function `cxPermMinors(arma::cx_mat C)` which does exactly this for us. Once again, Armadillo data types and functions and data classes have been used extensively. Since the original R package (Section 6.1) already had an implementation of this particular function in C++, we used this in our implementation to begin with and then optimised it as required. In our code, `m` denotes number of rows and `n` denotes number of columns of the input matrix. Since the transpose of the matrix is being used from the Boson Sampling function, `m == n+1`. Using this notation, recall the formula for calculating the permanent minors,

$$\operatorname{Per} B^{\diamond}_{m,l} = \frac{1}{2^{m-2}} \sum_{\delta} \left(\prod_{s=1}^{m-1} \delta_s \right) \prod_{i \in [m] \setminus l} v_i(\delta), \quad l \in [m], \qquad (16)$$

where $\delta \in \{-1,1\}^{m-1}$ with $\delta_1 = 1$ and $v_i(\delta) = \sum_{j=1}^{m-1} \delta_j b_{ji}$. The two tricks to speed up the naive implementation were to iterate the values of δ over the Gray code, and to compute the innermost product $\prod_{i \in [m] \setminus l} v_i(\delta), l \in [m]$ with the help of the forward and backward cumulative products (Section 5.3). Since there is a lot going on in this single formula, we divide the operations being performed into 5 parts for a clearer explanation:

a) Iterating δ over Gray code: In order to iterate over values of $\delta \in \{-1,1\}^{m-1}$ with $\delta_1 = 1$ in Gray code order (Section 3.1), the code uses the help of two additional variables, `int j` and `arma::ivec g`. `arma::ivec` is a vector of integers from Armadillo. `j` is the active index i.e. to go from one value in the Gray code to the next, the j^{th} bit is flipped. Additionally, the original code uses an auxiliary array `g` to change from one value of `j` to the next. The condition of the `while` loop ensures that all the gray codes of size $n-2$ are iterated over. δ is represented by a boolean valued vector `d`, and `true` is considered equivalent to 1, and `false` equivalent to -1. The relevant snippets of code from the function are shown below.

```
int j = 0, k;
arma::uvec d(n); d.fill(true);
arma::ivec g = arma::regspace< arma::ivec>(0,(n-1));
...
while(j < n-1){
   ...
    d[j] = !d[j];
    // Iterate Gray code: j is active index
    if( j > 0){
        k = j + 1; g[j] = g[k]; g[k] = k; j = 0;
    } else {
        j = g[1]; g[1] = 1;
    }
}
```

b) Computing $\mathbf{v}(\delta) = \{v_i(\delta), i \in [m]\}$ efficiently for each δ: At the start of the function, $\mathbf{v}(\delta)$ is initialised to the row-sums of the input matrix `C`, divided by two. This is because δ is initialised to an all-1 (or equivalently all-`true`) vector, so $v_i(\delta) = \sum_{j=1}^{m-1} \delta_j b_{ji} = \sum_{j=1}^{m-1} b_{ji}$. The actual code does a row sum instead of a column sum like the formula suggests due to the matrix being transposed. The reason for dividing by 2 is explained later. In each iteration of the loop over δ, the active index j is used to check the value of δ_j, which is the element of δ about to be flipped, and accordingly adds or subtracts the j^{th} column to $\mathbf{v}(\delta)$. These operations are performed in the following lines of code.

```
...
v = arma::sum(C,1)/2;
...
while( ... ){
    if(d[j]) v -= C.col(j); else v += C.col(j);
    ...
}
```

c) Computing the partial products $\prod_{i \in [m] \setminus l} v_i(\delta), \quad l \in [m]$: In order to exploit the trick of calculating the partial products of $\mathbf{v}(\delta)$ quickly, we use a vector `p` to store the accumulated result of calculating the permanent minors, and directly add or subtract the partial products to this vector. `p` is initialised with the partial products of the initial values of `v` which can be seen in the lines of code that follow

```
...
arma::cx_vec p(m), q(m);
arma::cx_double t;
...
q = arma::cumprod(v);

t = v[m-1];         // last element of v
p[m-1] = q[m-2];
for(i = m-2; i > 0; i--) {
    p[i] = t*q[i-1];
    t *= v[i];
}
p[0] = t;
...
}
```

In this code, `q` is the vector of forward cumulative products, and `t` is a single variable being used to successively store values of the backward cumulative product without having to precompute and store all of them. The same method as above is used while adding or subtracting these partial products in each iteration of the loop.

d) Computing the value $\left(\prod_{s=1}^{m-1} \delta_s\right)$: Another result of iterating over δ in Gray code order is that the product $\prod_{s=1}^{m-1} \delta_s$ does not need to be calculated in each iteration of the loop. The reason being that exactly one element of δ is flipped in each iteration, so $\prod_{s=1}^{m-1} \delta_s$ alternates between the values $+1$ and -1 in every successive iteration. Equivalently, it means that we alternate between adding and subtracting the partial products to the accumulator in successive iterations of the loop. Therefore, this product is represented by a single boolean variable `s` that is negated at the end of each loop iteration.

```
...
s = true;
...
while(j < n-1){
   ...
   if(s){
        //Subtract partial products of v from p
    } else {
        //Subtract partial products from p
    }
    s = !s;
    ...
}
```

e) Multiplying by $\frac{1}{2^{m-2}}$ term: Finally, the sum in the formula is divided by 2^{m-2}, but rather than evaluating this separately, and dividing all the elements of the result by it, we initially divide each element of $\mathbf{v}(\delta)$ by 2. Then, when the partial products of $\mathbf{v}(\delta)$ are being computed, each partial product is divided by a factor 2^{m-1}, and then assigned to the accumulator `p`. Before returning `p` at the end of the function, it is multiplied by 2 in order to give the desired value of 2^{m-2} in the denominator.

The object `p` returned is a vector containing the set of permanent minors of the input matrix `C`.

6.3 Simulating exact Boson Sampling using Algorithm B

The function `bosonSampler(arma::cx_mat A, int n, int m)` is an implementation of Algorithm B (Section 5.3) of the exact Boson Sampling problem in C++. We followed the algorithm exactly as it is, and describe a line-by-line explanation below (excluding trivial steps). The function takes as arguments a complex-valued Armadillo matrix `A` which is a random unitary matrix as required by the algorithm, an integer `n` representing input size, and an integer `m` representing output modes.

Preprocessing steps Before we start the actual algorithm, A is required to be an $m \times n$ matrix as we require only the first n columns. A random seed is generated using a Mersenne twister engine which is a uniform fast pseudo-random number generator [17]. The random seed is required later for the sampling steps in the algorithm.

```
// Take first n columns of A
A.set_size(m, n);

// Generate random seed
random_device rd;
mt19937 gen(rd());
```

Line 2 requires the columns of A to be randomly permuted. In order to do this, we have implemented a simple Knuth-shuffle algorithm [12].

```
// Line 2 : Permute columns
for (int i = 0; i <= n-2; i++) {
    uniform_int_distribution<int> uni(i, n-1);
    int j = uni(gen);
    A.swap_cols(i, j);
}
```

Line 3 and 4 involves making a weighted array $\mathbf{w}$ and sampling a value x from it. We created a C++ standard vector of doubles to represent the weighted array using the formula provided in Line 3 of the algorithm. C++ provides a class for creating creating discrete distributions using indexed weight arrays or vectors. This was used to generate the distribution and the random seed generated earlier was used to sample x from it. 1 is added to the value sampled, as indices of C++ vectors start at 0, but the problem requires the indices to start at 1. This sampling was carried out in the code snippet below.

```
// Line 3
vector<double> w;
for (int i = 1; i <= m; i++) {
    w.push_back(norm(A(i-1, 1)));
}

// Line 4
discrete_distribution<> d(w.begin(), w.end());
int x = d(gen) + 1;
```

Lines 6 to 11 contains the loop for repeatedly generating the marginal distributions and sampling values from them. Firstly, the permanent minors are computed of the submatrix of A and stored in an Armadillo vector, by making a call to `cxPermMinors(arma::cx_mat)`:

```
// Line 8
arma::cx_vec perms;
perms = cxPermMinors(B_k);
```

The permanent minors are used to create a weight vector using the formula in Line 9. As before, an integer x is sampled from the discrete distribution represented by a weight vector $\mathbf{w}$, and then appended to the vector $\mathbf{r}$.

Line 12 is the final step of sorting the vector of sampled values r in non-decreasing order. The `sort()` function in the standard C++ library does exactly this for vectors.

```
// Line 12
vector<int> z;
sort(r.begin(), r.end());
z = r;
```

6.4 Running the code

The files containing the functions were initially compiled on a personal computer, using the g++ compiler with no extra flags added for optimisation. The `main()` function required the user to enter a value of n when running the function, and then set $m = n^2$. An $m \times m$ complex valued random unitary matrix is generated using the `randomUnitary(int)` function. Subsequently, the `bosonSampler(arma::cx_mat, int, int)` function is called, and a single sample from the Boson Sampling distribution with input size n and m output modes is obtained. Finally, this sample is printed out to the command line. An example of this process with $n = 10$ is shown below. The following instructions are run in the same directory as the C++ files.

```
> g++-8 bosonSampling_b_arma.cpp cxPermMinors.cpp randomUnitary.cpp \
>         -o a.out -std=c++11 -larmadillo
> ./a.out 25
[ 7 21 28 37 38 47 63 67 91 97 ]
```

6.5 Profiling the code

7 Results

8 Conclusion

References

[1] Marvin Marcus and Henryk Minc. Permanents. *The American Mathematical Monthly*, vol. 72:577–591, 1965.

[2] Herbert John Ryser. *Combinatorial Mathematics.* Mathematical Association of America, 1963.

[3] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979.

[4] Albert Nijenhuis and Herbert S. Wilf. 23 - the permanent function (perman). In *Combinatorial Algorithms (Second Edition)*, pages 217 – 225. Academic Press, second edition edition, 1978.

[5] K Balasubramanian. *Combinatorics and diagonals of matrices.* PhD thesis, Indian Statistical Institute, Calcutta, Dec 1980.

[6] Eric Bax. *Finite-difference algorithms for counting problems.* PhD thesis, California Institute of Technology, February 1998.

[7] Eric Bax and Joel Franklin. A finite-difference sieve to count paths and cycles by length. *Information Processing Letters*, 60(4):171 – 176, 1996.

[8] David G. Glynn. The permanent of a square matrix. *European Journal of Combinatorics*, 31(7):1887 – 1891, 2010.

[9] Peter Clifford and Raphaël Clifford. The classical complexity of boson sampling. *CoRR*, abs/1706.01260, 2017.

[10] William Feller. *An Introduction to Probability Theory and its Applications*, volume 1. Wiley, 1968.

[11] A. J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8):127–128, April 1974.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms.* Addison-Wesley, 1969.

[13] Peter Clifford and Raphaël Clifford. *BosonSampling: Classical Boson Sampling*, 2017. R package version 0.1.1.

[14] Dirk Eddelbuettel, Romain Francois, JJ Allaire, Kevin Ushey, Qiang Kou, Nathan Russell, Douglas Bates, and John Chambers. *RCpp*, 2017. R package version 0.12.12.

[15] Conrad Sanderson. Armadillo c++ linear algebra library, June 2016.

[16] Maris Ozols. How to generate a random unitary matrix. Technical report, 2009.

[17] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.