NAME OF THE TEAM

APPLICATION OF BOSON SAMPLING IN GENERATING RANDOM NUMBERS"

TEAM MEMBERS

1. Idris Ahmad#0267, https://github.com/IDRISAHMAD, iahmad.phy@buk.edu.ng
2. Fatou Cheikh Fall, fatou.c.fall@aims-senegal.org
3. Lib, lib2@carleton.edu
4. Saul Yael Puente Ruiz, saulpuente@live.com
5. Firew meka, firewm20@gmail.com
6. Progressmunoriarwa32@gmail.com

Pitch Presenter: Idris Ahmad

Name of challenge: Random-number-generation-using-boson-sampling---ORCA-Computing

Abstract

In this challenge, a boson sampler, a type of quantum computing method, was used to construct an application that can produce random numbers. Both Penceval and Strawberryfields were used to generate random numbers 0s and 1s. Von Neumann correction was applied to generated random numbers, which the sequence of unbiased random numbers. These results might be useful on a project that utilizes quantum computation. It is also carried out using Xanadu environment a publicly available quantum computer, this proving the superiority of quantum computing to classical computation.

Introduction

A lot of the technology we use every day is based on random number generation (RNG). The most significant of them is cryptography, which uses random numbers to create keys to encrypt confidential information so that it cannot be deciphered by an adversary. Comparable to generating a password, if your password must consist of a string of three letters, there are 52 potential characters (including upper- and lower-case letters), giving an attacker 523 = 140,608 different ways to guess your password. However, if the attacker is aware that your password is not entirely random, for instance, if they know it begins with a capital letter and the rest are lower case, it reduces the number of options to 263 = 17,576, which makes their work of guessing your password much more difficult.

A random number generator attack is what is used to take advantage of such vulnerabilities. Recently, there have been worries raised about the potential vulnerability of RSA, one of the most extensively used cryptographic standards. Everything on the internet, including the security of the internet of things and the confidentiality of your online banking information, is supported by this technology.

There are uses for random numbers besides encryption. They are necessary for lotteries and other chance-based games, as well as scientific investigations and surveys that involve random sampling.

The Boson sampling

In boson sampling, identical single photons are transmitted through a network of beam splitters that make up an optical interferometer, producing outputs that have a complicated probability distribution over the number of photons. The output of even relatively modest interferometers with only a few tens of photons is classically challenging to mimic within an acceptable length of time, despite the seeming simplicity of photon interference due to its quantum nature.

Procedure

To create an unbiased random string of 1s and 0s, we implement the RNG using Strawberryfields and Perceval simulators.

The post-processing process is broken down into the following steps.

Step 1: Conduct a Boson sampling and mark the results with the letter S1.

Step 2 Repeat the Boson sampling and note the outcome as S2.

Step 3: Comparing S1 and S2, a mode's output code is 0 bits if it detects a photon in S1 but not in S2, and vice versa. Similar to this, if a mode detects a photon in S2 but not S1, it produces a 1 bit output code. Nothing else is coded.

Step 4: The coded bits for modes 1 through M

Modified Code

```
#one can choose which mode he/she wants at input, or we can choose it randomly
def Generating_Input(n, m, modes = None):
    "This function randomly chooses an input with n photons in m modes."
    if modes == None :
        modes = sorted(random.sample(range(m),n))
    state = "|"
    for i in range(m):
        state = state + "0"*(1 - (i in modes)) +"1"*(i in modes)+ ","*(i < m-1)
    return pcvl.BasicState(state + ">")

input_state = Generating_Input(n, m)
print("The input state: ", input_state)
```

```
The input state:
|0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,1,0,0,0,0,1,1,0,0,0,0,0,
0,1,0,0,0,1,0,0,0,0,0,0,0>
```

```
s1 = input_state
print(s1)
```

```
|0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,1,0,0,0,0,1,1,0,0,0,0,0,
0,1,0,0,0,1,0,0,0,0,0,0,0>
```

```
input_state = Generating_Input(n, m)
#print("The input state: ", input_state)
s2 = input_state
print(s2)
```

```
|0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,1,
1,0,0,0,0,1,1,1,0,0,0,0,0>
```

```
coding = []
for i in range(1, 50):
    if 1:
        coding.append('0')
    elif 2:
        coding.append('1')
    else: coding.append('*')
    #else coding[i] = '*'

coding = []
```

```
for i in range(0, 50):
    if s1[i] == 1 and s2[i] == 0:
        coding.append('0')
    elif s1[i] == 0 and s2[i] == 1:
        coding.append('1')
    else: coding.append('*')
    #else coding[i] = '*'
print("The first sampling result S1" , s1)
print("The second sampling results S2" , s2)
print("Coding" , coding)

r = ''
for i in coding:
    if i != '*':
      # r.append(i)
        r = r + i
print("The final random number sequence:", r)

worker_id=1

#store the input and the unitary
with open("%dphotons_%dmodes_%dsamples-worker%s-unitary.pkl" %(n,m,N,worker_id), 'wb') as f:
    pickle.dump(Unitary_60, f)

with open("%dphotons_%dmodes_%dsamples-worker%s-inputstate.pkl" %(n,m,N,worker_id), 'w') as f:
    f.write(str(input_state)+"\n")


with gzip.open("%dphotons_%dmodes_%dsamples-worker%s-samples.txt.gz" %(n,m,N,worker_id), 'wb') as f:
    start = time.time()
    for i in range(N):
        f.write((str(Sampling_Backend(Unitary_60).sample(pcvl.BasicState(input_state)))+"\n").encode());
    end = time.time()
    f.write(str("==> %d\n" % (end-start)).encode())
f.close()

import gzip
worker_id = 1
count = 0
bunching_distribution = Counter()

with gzip.open("%dphotons_%dmodes_%dsamples-worker%s-samples.txt.gz"%(n,m,N,worker_id), "rt") as f:
    for l in f:
        l = l.strip()
        if l.startswith("|") and l.endswith(">"):
            try:
                st = pcvl.BasicState(l)
                count+=1
                bunching_distribution[st.photon2mode(st.n-1)]+=1
            except Exception:
                pass
print(count, "samples")
print("Bunching Distribution:", "\t".join([str(bunching_distribution[k]) for k in range(m)]))
```

Results

The following results were obtained

```
n = 10      #number of photons at the input
m = 50      #number of modes
N = 50000  #number of samplings

The first sampling result S1
|0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,1,0,0,0,0,1
,1,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0>
The second sampling results S2
|0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0
,0,0,1,0,0,1,1,0,0,0,0,1,1,1,0,0,0,0,0>
Coding ['*', '0', '*', '0', '*', '0', '*', '*', '*', '1', '1',
'*', '*', '*', '*', '*', '0', '*', '*', '*', '*', '*', '0',
'*', '*', '0', '*', '1', '*', '1', '0', '0', '*', '1', '*',
'*', '1', '1', '0', '*', '*', '*', '*', '1', '1', '*', '*',
'*', '*', '*']
The final random number sequence: 000110001100111011
```

```
n = 14      #number of photons at the input
m = 20      #number of modes
N = 5000000  #number of samplings

S1 |1,1,1,0,0,0,1,1,1,1,0,0,1,0,1,1,1,1,1,1>
S2 |1,0,1,1,0,0,1,0,1,1,1,1,1,0,1,1,1,0,1,1>
Coding ['*', '0', '*', '1', '*', '*', '*', '0', '*', '*', '1',
'1', '*', '*', '*', '*', '*', '0', '*', '*']
The final quantum random number: 010110
```

Conclusion

The advantage of Boson sampling over other sample distributions is that the output probability distribution is not very near to the uniform distribution in terms of variation distance. The random number generator must have a uniform distribution of 01 bits in the output sequence, where the probability of each bit being 0 or 1 is both 0.5, which is unbiased otherwise, the likelihood of successfully "guessing" increases.

The suggested technique uses the Von Neumann correction method to ensure impartiality. The output probability distribution produced by Boson sampling, which is still not uniform, has not changed, though. The likelihood of each mode coded as 0 or 1 is comparable when the post-processing is complete, and the 01 bits in the final random number sequence likewise have a uniform distribution. This approach has the advantage of maintaining the nonuniform distribution of boson sampling, unlike other schemes that directly alter the probability distribution of quantum systems. Additionally, the Boson sampling-based QRNG has a number of attractive properties.

Reference

Hui Wang, et al. Boson Sampling with 20 Input Photons and a 60-Mode Interferometer in a 1014-Dimensional Hilbert Space. Physical Review Letters, 123(25):250503, December 2019. Publisher: American Physical Society.

Jinjing Shi1 , Tongge Zhao1 , Yizhi Wang2 , Chunlin Yu3 , Yuhu Lu1 , Ronghua Shi1 , Shichao Zhang1 , and Junjie Wu2 An Unbiased Quantum Random Number Generator Based on Boson Sampling