ps2-mark-webster

March 29, 2022

1 Quantum Algorithms Problem Set 2 - Mark Webster

1.1 1. Generalised Bernstein-Vazirani

Let M, n be positive integers. Let $s \in \mathbb{Z}_M^n$ and define the function $f_s : \mathbb{Z}_M^n \to \mathbb{Z}_m$ by $f_s(x) = \langle s, x\rangle \mod M$. Given access to an oracle O_{f_s} which for $x \in \mathbb{Z}_M^n, b \in \mathbb{Z}_M$ acts as $O_{f_s}|x\rangle|b\rangle = |x\rangle|b + f_s(x) \mod M\rangle$, design a quantum algorithm that computes s with one application of O_{f_s}.

Define the action of the QFT and its inverse on the computational basis element $|b\rangle, b \in \mathbb{Z}_M$:
$F_M|b\rangle = \frac{1}{\sqrt{M}}\sum_{0\leq k<M} \omega^{-b\cdot k}|k\rangle$ and
$F_M^{-1}|b\rangle = \frac{1}{\sqrt{M}}\sum_{0\leq k<M} \omega^{b\cdot k}|k\rangle$.

The action of $F_M^{\otimes n}$ on a basis element $|b\rangle$ where $b \in \mathbb{Z}_M^n$ is given by:
$F_M^{\otimes n}|b\rangle = \frac{1}{\sqrt{M^n}}\sum_{k\in\mathbb{Z}_M^n} \omega^{-\langle b,k\rangle}|k\rangle$

Define $O'_f := (I \otimes F_M) \circ (O_f) \circ (I \otimes F_M^{-1})$. Claim that the action of O'_f is $O'_f|x\rangle|b\rangle = \omega^{-f(x)\cdot b}|x\rangle|b\rangle$:
$|\psi_0\rangle = |x\rangle|b\rangle$
$|\psi_1\rangle = \frac{1}{\sqrt{M}}\sum_{0\leq k<M} \omega^{b\cdot k}|x\rangle|k\rangle$ - Apply $I \otimes F_M^{-1}$
$|\psi_2\rangle = \frac{1}{\sqrt{M}}\sum_{0\leq k<M} \omega^{b\cdot k}|x\rangle|k + f(x) \mod M\rangle$ - Apply O_f

Now $b \cdot k = b \cdot (k + f(x) \mod M) - b \cdot f(x)$ so
$|\psi_2\rangle = \frac{1}{\sqrt{M}}\omega^{-b\cdot f(x)}\sum_{0\leq k<M} \omega^{b\cdot(k+f(x) \mod M)}|x\rangle|k + f(x) \mod M\rangle$

By re-indexing the sum, we have:
$|\psi_2\rangle = \frac{1}{\sqrt{M}}\omega^{-b\cdot f(x)}\sum_{0\leq k<M} \omega^{k}|x\rangle|k\rangle$
$|\psi_3\rangle = \frac{1}{\sqrt{M}}\omega^{-b\cdot f(x)}|x\rangle|b\rangle$ - Apply $I \otimes F_M$.

The generalised Bernstein-Vazirani algorithm is then the same as for qubits, but generalised to M-level systems: $((F_M^{\otimes n})^{-1} \otimes I) \circ O'_{f_s} \circ (F_M^{\otimes n} \otimes I)|0^n\rangle|1\rangle$.
$|\psi_0\rangle = |0^n\rangle|1\rangle$
$|\psi_1\rangle = \frac{1}{\sqrt{M^n}}\sum_{k\in\mathbb{Z}_M^n} |k\rangle|1\rangle$ - apply $F_M^n \otimes I$
$|\psi_2\rangle = \frac{1}{\sqrt{M^n}}\sum_{k\in\mathbb{Z}_M^n} \omega^{-f_s(k)}|k\rangle|1\rangle = \frac{1}{\sqrt{M^n}}\sum_{k\in\mathbb{Z}_M^n} \omega^{-\langle s,k\rangle}|k\rangle|1\rangle = (F_M^{\otimes n} \otimes I)|s\rangle|1\rangle$ - apply O'_{f_s}
$|\psi_3\rangle = |s\rangle|1\rangle$ - apply $(F_M^{\otimes n})^{-1} \otimes I$

Hence after measurment, we learn s with probability 1.

1.2 2. Continued Fractions

1. Find the continued fraction expansion of $\frac{527}{1024}$

$$\frac{527}{1024} = 0 + \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{16+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{3+\frac{1}{4}}}}}}}$$

Determined using the Python program below:

[ ]:
```
from fractions import Fraction
import numpy as np

## convert rational q to continued fraction A
## maxQ is the maximum allowed value of the denominator of the approximation
def Q2cf(x,verbose=False,maxQ=None):
    print(f'Calculating Continued Fraction for {x}')
    ## maximum denominator for approximation
    maxQ = x.denominator if maxQ is None else maxQ
    ## array of the a_i to return
    A = []
    ## numerators and denominations of jth convergent
    ## initial values to make inductive expression work
    P = [0,1]
    Q = [1,0]
    ## remaining value to turn into continued fraction
    xj = x
    ## repeat the loop up to maxloop times
    maxloop = 10000
    for j in range(maxloop):
        ## a_j is the floor of x_j
        a = int(xj.__floor__())
        if verbose:
            print(f'x_{j} = {xj};        a_{j} = {a}')
        ## jth convergent is p/q
        p,q = a * P[j+1] + P[j], a * Q[j+1] + Q[j]
        if q > maxQ:
            break
        ## update A, P, Q
        A.append(a)
        ## p_{j+2} = a_{j+2}p_{j+1} + p_j
        P.append(p)
        ## q_{j+2} = a_{j+2}q_{j+1} + q_j
        Q.append(q)
        ## update x
        xj = xj - a
        ## if xj is zero, we have an exact approximation
        if xj == 0:
            break
        ## otherwise x_{j+1} = 1/x_{j}
        xj = 1/xj
    return A,P,Q
```

```
## convert continued fraction A to rational x
def cf2Q(A):
    x = None
    for i in range(len(A)-1,-1,-1):
        x = Fraction(A[i],1) if x is None else A[i] + 1/x
    return x

## TeX representation of continued fraction
def cf2str(A,i):
    if i == len(A) - 1:
        return str(A[i])
    else:
        return str(A[i]) + "+ \\frac{1}{" + cf2str(A,i+1) +"}"


## fraction to convert to continued fraction
x = Fraction(527,1024)

## Return continued fraction: A, P, Q are the lists of a_j, p_j, q_j␣
 ↪respectively
A,P,Q = Q2cf(x,verbose=True)
print('TeX Representation of continued fraction:',cf2str(A,0))
```

```
Calculating Continued Fraction for 527/1024
x_0 = 527/1024;       a_0 = 0
x_1 = 1024/527;       a_1 = 1
x_2 = 527/497;       a_2 = 1
x_3 = 497/30;       a_3 = 16
x_4 = 30/17;       a_4 = 1
x_5 = 17/13;       a_5 = 1
x_6 = 13/4;       a_6 = 3
x_7 = 4;       a_7 = 4
TeX Representation of continued fraction: 0+ \frac{1}{1+ \frac{1}{1+
\frac{1}{16+ \frac{1}{1+ \frac{1}{1+ \frac{1}{3+ \frac{1}{4}}}}}}}
```

2. Look at the j^{th} convergents of your expression and make a conjecture about the even numbered convergents (you do not need to prove it).

For the j^{th} convergent c_j, we have that $x \geq c_j$ if j is even and $x \leq c_j$ if j is odd as demonstrated in the example below:

```
[ ]: for j in range(len(A)):
         ## remember to offset P, Q by 2
         cj = Fraction(P[j+2],Q[j+2])
         s = "{:.2E}".format(float(x-cj))
         print(f'c_{j}={cj};    x - c_{j} = {s}')
```

```
c_0=0;    x - c_0 = 5.15E-01
c_1=1;    x - c_1 = -4.85E-01
```

```
c_2=1/2;    x - c_2 = 1.46E-02
c_3=17/33;    x - c_3 = -5.03E-04
c_4=18/35;    x - c_4 = 3.63E-04
c_5=35/68;    x - c_5 = -5.74E-05
c_6=123/239;    x - c_6 = 4.09E-06
c_7=527/1024;    x - c_7 = 0.00E+00
```

3. Write a program in any language to compute a continued fraction of an input number up to a given accuracy

See Question 1.

1.3 3. Factoring 21: Let's factor the number $M = 21$ using Shor's algorithm.

1. List all numbers in $_{21}$ that are relatively prime to 21. These are the elements of the multiplicative group $^{\times}_{21}$. Compute the order $\text{ord}_{21}(x)$ of all elements in $^{\times}_{21}$.

[ ]:

```
M = 21

def order(x,M):
    xi = x
    for i in range(M):
        if xi % M == 1:
            return i+1
        xi *= x
    return False

def print_table(data):
    sep = " | "
    temp = []
    data = [[str(a) for a in r ] for r in data]
    colwidth = [max([len(a) for a in c]) for c in np.transpose(data)]
    for r in data:
        r = [r[j].rjust(colwidth[j]) for j in range(len(r))]
        temp.append(sep + sep.join(r) + sep)
    return "\n".join(temp)

data = [["x", 'ord(x) mod 21']]

ord6 = []

# Calculate GCD
G = np.gcd(M,range(M))
for x in range(M):
    if G[x]==1:
        ## calculate order if relatively prime
        o = order(x,M)
        if o is not False:
            data.append([x,o])
            if o == 6:
                ord6.append(x)

print(print_table(data))
```

```
|  x | ord(x) mod 21 |
|  1 |             1 |
|  2 |             6 |
|  4 |             3 |
|  5 |             6 |
|  8 |             2 |
| 10 |             6 |
| 11 |             6 |
| 13 |             2 |
| 16 |             3 |
| 17 |             6 |
| 19 |             6 |
| 20 |             2 |
```

2. Recall that in Shor's algorithm we want to find an x of even order d such that $x^{d/2} \neq -1 \mod M$. Call such an x *good*. Identify all the good $x \in_{21}^{\times}$ with $\mathrm{ord}_{21}(x) = 6$ and for these verify that $\gcd(x^3 \pm 1, 21)$ gives a nontrivial factor of 21.

[ ]:
```
data = [['x','gcd(x^3+1,M)','gcd(x^3-1,M)']]
for x in ord6:
    x3 = x**3
    if (x3 + 1) % M > 0:
        g1 = np.gcd(x3+1,M)
        g2 = np.gcd(x3-1,M)
        data.append([x,g1,g2])

print(print_table(data))
```

```
|  x | gcd(x^3+1,M)  | gcd(x^3-1,M)  |
|  2 |             3 |             7 |
| 10 |             7 |             3 |
| 11 |             3 |             7 |
| 19 |             7 |             3 |
```

3. Choose a good x of order 6 from the previous step. Now let's simulate finding the period of $f(j) = x^j \bmod 21$. Using the Octave FTperiod program. This simulates randomly sampling a state g_t and measuring $F_N g_t$ to see an index b. Use continued fraction expansion on b/N and see if you can recover $\mathrm{ord}_{21}(x)$. It may take several attempts. Record the values you see and how many attempts it takes.

[ ]:
```
import math
import numpy as np
from scipy import fftpack
from scipy.linalg import dft
import matplotlib.pyplot as plt
```

```
def sampleb(probs):
    # sample entry b with prob |Y(b,j)|^2
    thresh = np.random.rand(1)
    b = -1
    prob_sum = 0
    while prob_sum < thresh:
        b += 1
        prob_sum += probs[b]
    return b

N = M**2
## choose a 'good' x with order 6
x = 19
s = order(x,M)

T = math.floor(N/s)
R= N % s

I = np.identity(s)
X = np.tile(I,(T,1))
X = np.vstack((X,I[0:R,:]))
# normalize X so columns have unit norm
X = X/np.sqrt(sum(X))

# Fourier transform
F_N = dft(N)/np.sqrt(N)
Y = F_N @ X

for i in range(100):
    # sampling
    # sample a random coset state
    j = np.random.randint(s)
    probs = abs(Y[:,j])**2
    b = sampleb(probs)
    print(f'\nIteration {i+1}')
    print('b = ',b)
    A,P,Q = Q2cf(Fraction(b,N),maxQ=M)
    p,q = P[-1],Q[-1]
    print(f'Continued fraction estimate: {p}/{q}')
    xq = x**q % M
    print(f'Checking order: x={x}, x^{q} = {xq} mod {M}')
    if 1 == xq:
        print('Period finding succeeded')
        break
    print('Period finding failed')
```

```
Iteration 1
b =  0
Calculating Continued Fraction for 0
Continued fraction estimate: 0/1
Checking order: x=19, x^1 = 19 mod 21
Period finding failed

Iteration 2
b =  294
Calculating Continued Fraction for 2/3
Continued fraction estimate: 2/3
Checking order: x=19, x^3 = 13 mod 21
Period finding failed

Iteration 3
b =  147
Calculating Continued Fraction for 1/3
Continued fraction estimate: 1/3
Checking order: x=19, x^3 = 13 mod 21
Period finding failed

Iteration 4
b =  368
Calculating Continued Fraction for 368/441
Continued fraction estimate: 5/6
Checking order: x=19, x^6 = 1 mod 21
Period finding succeeded
```

1.4 4. Assumptions

Where in the proof of Shor's algorithm for the general period finding problem with a function $f : \mathbb{Z}_N \to [M]$ do we use the assumption that $N > M^2$? What can go wrong without this assumption?

When measuring the first register in Step 3 of the algorithm, we obtain a value b which is close to kN/s with high probability, where s is the period of the function and k is an integer. More specifically, $|b/N - k/s| \leq 1/2N$. We know b/N and we want to find k, s. Two distinct fractions with denominators $\leq M$ must be at least $1/M^2$ apart. Due to the injectivity assumption, $s \leq M$.

Using the assumption $N > M^2$, we see that $|b/N - k/s| \leq 1/2N \leq 1/2M^2$ and there is a unique fraction k/s with denominator of maximum size M meeting this condition. We can determine this fraction using either continued fractions or an integer linear program, and use this to derive the period s.

If the assumption were not true, there could be more than one fraction meeting this condition, and we would not necessarily be able to determine the value of s.

1.5 5. Nielsen and Chuang

At the end of this problem set is attached the section from Nielsen and Chuang on period finding (section 5.4.1). Compared to our description in lecture, the Nielsen and Chuang presentation removes a key assumption about the function f. Does the algorithm still work without this assumption? If not, what can go wrong? You can describe it in words or show a numerical example in your favourite language.

The missing assumption is the injectivity of the periodic function f on $[0 \ldots s]$ where s is the period of the function. That is, they assume that f meets the condition $f(x+s) = f(x)$ but omit the condition that $f(x) \neq f(y)$ if $x, y < r$.

In this case, different values of f could occur with different periods which would lead to incorrect values for the period of the function itself. The injectivity assumption implies that the values of f occur with the same periodicity.

As an example, let $M = 8, N = 65$ and $f(x) = GCD(x, M)$. The value $f(x) = 1$ has periodicity 2, $f(x) = 2$ has periodicity 4 and the values $f(x) = 4, 8$ have periodicity 8. The correct period of the function itself is 8 as illustrated below.

In Step 2b, when measuring the value of $f(x)$ on the 2nd register, if we happen to obtain a value other than 4 or 8, we would obtain the incorrect value of the period of the function.

[ ]:
```python
M= 8
N = M**2+1
c = M
data = [['x','f(x)'] * c]
G = np.gcd(M, range(N))
r = N//c + 1 if N % c > 0 else 0
for x in range(r+1):
    myrow = []
    for y in range(c):
        i = c*y + x
        if i < N:
            myrow.extend([i,G[i]])
        else:
            myrow.extend(["",""])
    data.append(myrow)
print(print_table(data))
```

```
 | x | f(x) |  x | f(x) |  x | f(x) |  x | f(x) |  x | f(x) |  x | f(x) |  x |
f(x) |  x | f(x) |
 | 0 |    8 |  8 |    8 | 16 |    8 | 24 |    8 | 32 |    8 | 40 |    8 | 48 |
8 | 56 |    8 |
 | 1 |    1 |  9 |    1 | 17 |    1 | 25 |    1 | 33 |    1 | 41 |    1 | 49 |
1 | 57 |    1 |
 | 2 |    2 | 10 |    2 | 18 |    2 | 26 |    2 | 34 |    2 | 42 |    2 | 50 |
2 | 58 |    2 |
 | 3 |    1 | 11 |    1 | 19 |    1 | 27 |    1 | 35 |    1 | 43 |    1 | 51 |
1 | 59 |    1 |
 | 4 |    4 | 12 |    4 | 20 |    4 | 28 |    4 | 36 |    4 | 44 |    4 | 52 |
4 | 60 |    4 |
 | 5 |    1 | 13 |    1 | 21 |    1 | 29 |    1 | 37 |    1 | 45 |    1 | 53 |
1 | 61 |     1 |
 | 6 |     2 | 14 |     2 | 22 |     2 | 30 |     2 | 38 |     2 | 46 |     2 | 54 |
2 | 62 |     2 |
 | 7 |     1 | 15 |     1 | 23 |     1 | 31 |     1 | 39 |     1 | 47 |     1 | 55 |
1 | 63 |     1 |
 | 8 |     8 | 16 |     8 | 24 |     8 | 32 |     8 | 40 |     8 | 48 |     8 | 56 |
8 | 64 |     8 |
 | 9 |     1 | 17 |     1 | 25 |     1 | 33 |     1 | 41 |     1 | 49 |     1 | 57 |
1 |     |       |
```

1.6 6. Finding all ones

Let $N = 2^n$ and $x \in \{0,1\}^N$ and assume you know that x has k many ones. 1. In lecture we showed how to find an $i \in N$ such that $x_i = 1$ with constant probability by a quantum algorithm after $O(\sqrt{N/k})$ many queries to x. Show how to boost this success probability to $1 - 1/N^2$ using $O(\sqrt{N/k}\log(N))$ many queries to x.

From lectures, each time we apply Grover's algorithm, the probability of finding an $i \in N$ such that $x_i = 1$ is $1 - k/N$. Hence, the probability of failure is k/N. We use $O(\sqrt{N/k})$ queries each time.

Now consider applying Grover's algorithm l times. We want to find l such that the probability of failure is less than $1/N^2$. Hence: $(k/N)^l < 1/N^2$ and so $l\log(k/N) < 2\log(1/N)$. We can assume $k/N < 1/2$ as otherwise we have an efficient classical algorithm via random sampling. Hence $-2l < -2\log(N)$ and so we can set $l > \log(N)$.

The total number of queries required is $O(\sqrt{N/k}\log(N))$ and the probability of success is at least $1 - 1/N^2$.

2. Give a quantum algorithm to find all the ones in x with constant probability after $O(\sqrt{kN}\log(N))$ many queries to x.

Modify the oracle to apply a phase of -1 only where we have not previously found the solution.

This can be achieved by applying a circuit of the following form for solution i with $O(n) = O(\log(N))$ gates. Here, i is the previous solution represented as a length n binary string, j the input of the gate on n qubits and $|b\rangle$ is the output qubit from the oracle function which acquires a phase of -1 if the input j matches i. By applying a series of such circuits for each existing solution after the oracle, we exclude any such solutions.

Now consider applying the algorithm from part 1 k times. As we exclude previously found solutions, the probability of finding all k solutions is $(1 - 1/N^2)^k$ and this requires $k\sqrt{N/k}\log(N) = \sqrt{kN}\log(N)$ queries as required.