Quantum random-walk search algorithm

Neil Shenvi, Julia Kempe and K. Birgitta Whaley

[a]*Report by Aditya Kumar(1611006)*

1. Aim

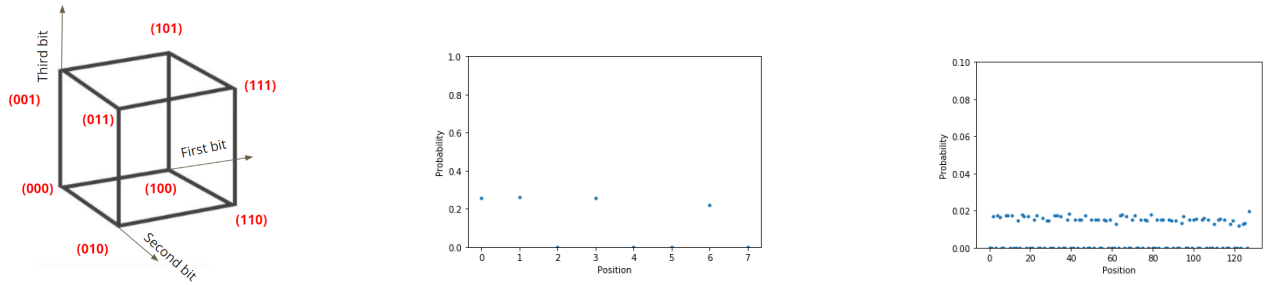
Random walks are well-studied topic in computer science and provide a promising methodology to make various algorithms. Classical random walks find there fare share of application in classical algorithms, references [1] to [5] show few algorithms that are based on a classical random walk. With the development in a quantum random walk, it was found that these random walks have some different properties from classical random walks. Quantum random walks feature exponentially fast hitting time and variance of order n in position after n steps. Despite offering such unique properties quantum random walk(QW) search algorithms haven't yet found their ways into quantum algorithms. In this paper authors have developed a search algorithm using quantum random walk which offers complexity of $\sqrt{n}$(same as Grover's algorithm.). This algorithm may provide a framework for the development of more QW based algorithms.

2. My understanding and work

Authors have done quantum random on a hypercube instead of random walk on a euclidian space. So I found it important to show how a classical random walk will behave on a hypercube. To do this I designed a python program to simulate random walk on a n cube.

To do this I first need to define a coin which will choose in which direction to traverse. This can be done easily in python by generating a random integer from 1 to n. Now for the shift operator, I devised an idea. All nodes of an n-cube can be equivalently written as a n bit string. If I need to traverse in i_{th} direction I just need to swap the i_{th} bit. Now to swap the i_{th} bit I just need to X0R that string with $(0_1 0_2 0_3 0_4 ... 1_i ... 0_n)$, this will be my shift operation.
To implement it into a python code and later Mathematica code I did all these calculation in decimals. Each node of n cube is can n bit binary string which will be a decimal number between 0 and $2^n - 1$. With a simple calculation, it can be seen that XORing n bit string with $0_1 0_2 0_3 0_4 ... 1_i ... 0_n$ is equivalent to adding 2^i and taking modulo 2^n. This is the idea I used to implement QW search algorithm on a Mathematica code as well.

Now the classical random walk on n cube works as follows: lest say algorithm starts with initial position p(a decimal number between 0 and 2^{n-1}), coin chooses a random number c between 1 to n(both 1 and n included). Shift operator calculates 2^{c-1} and adds it to p to give another number q. Then it takes modulo 2^n of q to give a new position. For T many steps it just needs to repeat this T times.Figure 1 shows probability vs position plot for 10 steps on a 3-cube and 7-cube.
Figure 1



(a) Nodes of 3-cube written as 3 bit string.

(b) Probability vs position plot for 3-cube after 10 steps

(c) Probability vs position plot for 7-cube after 10steps

Figure 1: Classical random walk on n cube. Please note that Probability of half the nodes is zero because because of discrete steps.

After understanding how to implement random walk on a hypercube I made a program in Mathematica to perform random walk search on n-cube. I used the same technique that I used for a classical random walk to form the shift operator. The code of Mathematica program to perform quantum random walk search is attached below

```
Needs["Quantum`Notation`"];
SetQuantumAliases[];

n = 3;  (*n coressponds to dimension of hypercube*)
d = Array[2^(# - 1) &, n];   (*a array of 2^(i-1)*)
shifto = Sum[Sum[ | i_CS, (BitXor[x, d[[i]]])_PS⟩ · ⟨i_CS, x_PS | , {x, 0, 2^n - 1}], {i, 1, n}];
(*shifto is the shift operator for quantum random walk on a hypercube.*)
sc = 1 / √n Sum[ | i_CS⟩, {i, 1, n}]; (*uniform superposition of all directions*)
idenc = Sum[ | i_CS⟩ · ⟨i_CS | , {i, 1, n}]; (*direction identity matrix*)
idenp = Sum[ | x_PS⟩ · ⟨x_PS | , {x, 0, (2^n) - 1}];(*position identity matrix*)
C0 = Expand[-idenc + 2 sc · (sc)†]; (*Grovers coin*)
coin = Expand[ C0 ⊗ idenp];
C1 = -idenc;
C2 = Expand[C1 - C0]; (*Inversion operator*)
C3 = Expand[ C2 ⊗  | 0_PS⟩ · ⟨0_PS | ]; (*inverts the target,Implementing oracle into coin*)

pcoin = Expand[coin + C3]; (*Implementing oracle into coin*)
Evolve = Expand[ pcoin  · shifto]; (*Evolution operator is product of shift operator and
 oracle implemented coin*)
ψ =  1 / (√(n * 2^n)) Sum[Sum[ | i_CS, x_PS⟩, {x, 0, 2^n - 1}], {i, 1, n}]
(*Uniform superposition of all states(The initial state)*)
t = (π / 2 * 2^(n / 2)) (*number of times it needs to iterate*)
Do[ψ = FullSimplify[Expand[ Evolve · ψ]], t]
(*Evolving the initial state t number of times*)
```

This program should work for any value of n although I have only checked up to n=5, for that program had to run overnight. In this algorithms authors have taken $\left|\overrightarrow{0}\right\rangle$ as their target, the algorithm will work the same for any arbitrary choice of target.

For n= 3 there are 3 coin states for each position state and 8 position states which gives a total of 24 states. To get the probability of target we need to sum the probability of all three states corresponding to target position state. The algorithm starts with uniform superposition of all states. So initially we have 8 position states each with probability 0.125.

$$\frac{1}{2\sqrt{6}}\left(\left|1_{\hat{CS}}, 0_{\hat{PS}}\right\rangle + \left|1_{\hat{CS}}, 1_{\hat{PS}}\right\rangle + \left|1_{\hat{CS}}, 2_{\hat{PS}}\right\rangle + \left|1_{\hat{CS}}, 3_{\hat{PS}}\right\rangle + \left|1_{\hat{CS}}, 4_{\hat{PS}}\right\rangle + \left|1_{\hat{CS}}, 5_{\hat{PS}}\right\rangle + \left|1_{\hat{CS}}, 6_{\hat{PS}}\right\rangle + \left|1_{\hat{CS}}, 7_{\hat{PS}}\right\rangle + \left|2_{\hat{CS}}, 0_{\hat{PS}}\right\rangle + \left|2_{\hat{CS}}, 1_{\hat{PS}}\right\rangle + \left|2_{\hat{CS}}, 2_{\hat{PS}}\right\rangle + \left|2_{\hat{CS}}, 3_{\hat{PS}}\right\rangle + \left|2_{\hat{CS}}, 4_{\hat{PS}}\right\rangle + \left|2_{\hat{CS}}, 5_{\hat{PS}}\right\rangle + \left|2_{\hat{CS}}, 6_{\hat{PS}}\right\rangle + \left|2_{\hat{CS}}, 7_{\hat{PS}}\right\rangle + \left|3_{\hat{CS}}, 0_{\hat{PS}}\right\rangle + \left|3_{\hat{CS}}, 1_{\hat{PS}}\right\rangle + \left|3_{\hat{CS}}, 2_{\hat{PS}}\right\rangle + \left|3_{\hat{CS}}, 3_{\hat{PS}}\right\rangle + \left|3_{\hat{CS}}, 4_{\hat{PS}}\right\rangle + \left|3_{\hat{CS}}, 5_{\hat{PS}}\right\rangle + \left|3_{\hat{CS}}, 6_{\hat{PS}}\right\rangle + \left|3_{\hat{CS}}, 7_{\hat{PS}}\right\rangle\right)$$

After running the algorithm $\pi/2 \times \sqrt{2^3} = 4.442 \approx 4$ times the state we get is

$$
\begin{aligned}
&-0.340207\ |1_{\hat{cs}}, 0_{\hat{ps}}\rangle + 0.264605\ |1_{\hat{cs}}, 1_{\hat{ps}}\rangle - 0.189004\ |1_{\hat{cs}}, 2_{\hat{ps}}\rangle + 0.113402\ |1_{\hat{cs}}, 3_{\hat{ps}}\rangle - 0.189004\ |1_{\hat{cs}}, 4_{\hat{ps}}\rangle + 0.113402\ |1_{\hat{cs}}, 5_{\hat{ps}}\rangle - \\
&0.158763\ |1_{\hat{cs}}, 6_{\hat{ps}}\rangle - 0.158763\ |1_{\hat{cs}}, 7_{\hat{ps}}\rangle - 0.340207\ |2_{\hat{cs}}, 0_{\hat{ps}}\rangle - 0.189004\ |2_{\hat{cs}}, 1_{\hat{ps}}\rangle + 0.264605\ |2_{\hat{cs}}, 2_{\hat{ps}}\rangle + 0.113402\ |2_{\hat{cs}}, 3_{\hat{ps}}\rangle - \\
&0.189004\ |2_{\hat{cs}}, 4_{\hat{ps}}\rangle - 0.158763\ |2_{\hat{cs}}, 5_{\hat{ps}}\rangle + 0.113402\ |2_{\hat{cs}}, 6_{\hat{ps}}\rangle - 0.158763\ |2_{\hat{cs}}, 7_{\hat{ps}}\rangle - 0.340207\ |3_{\hat{cs}}, 0_{\hat{ps}}\rangle - 0.189004\ |3_{\hat{cs}}, 1_{\hat{ps}}\rangle - \\
&0.189004\ |3_{\hat{cs}}, 2_{\hat{ps}}\rangle - 0.158763\ |3_{\hat{cs}}, 3_{\hat{ps}}\rangle + 0.264605\ |3_{\hat{cs}}, 4_{\hat{ps}}\rangle + 0.113402\ |3_{\hat{cs}}, 5_{\hat{ps}}\rangle + 0.113402\ |3_{\hat{cs}}, 6_{\hat{ps}}\rangle - 0.158763\ |3_{\hat{cs}}, 7_{\hat{ps}}\rangle
\end{aligned}
$$

This state can be measured in $|i_{\hat{cs}}, x_{\hat{ps}}\rangle$ basis to give the probability of being at state $\left|\overrightarrow{0}\right\rangle$. The probability comes out to be

$$P = 0.340207^2 + 0.340207^2 + 0.340207^2 = 0.347224$$

Now if we still evolve the state we get

$$
\begin{aligned}
&-0.264605\ |1_{\hat{cs}}, 0_{\hat{ps}}\rangle + 0.264605\ |1_{\hat{cs}}, 1_{\hat{ps}}\rangle - 0.189004\ |1_{\hat{cs}}, 2_{\hat{ps}}\rangle - 0.168843\ |1_{\hat{cs}}, 3_{\hat{ps}}\rangle - 0.189004\ |1_{\hat{cs}}, 4_{\hat{ps}}\rangle - 0.168843\ |1_{\hat{cs}}, 5_{\hat{ps}}\rangle - \\
&0.199084\ |1_{\hat{cs}}, 6_{\hat{ps}}\rangle - 0.158763\ |1_{\hat{cs}}, 7_{\hat{ps}}\rangle - 0.264605\ |2_{\hat{cs}}, 0_{\hat{ps}}\rangle - 0.189004\ |2_{\hat{cs}}, 1_{\hat{ps}}\rangle + 0.264605\ |2_{\hat{cs}}, 2_{\hat{ps}}\rangle - 0.168843\ |2_{\hat{cs}}, 3_{\hat{ps}}\rangle - \\
&0.189004\ |2_{\hat{cs}}, 4_{\hat{ps}}\rangle - 0.199084\ |2_{\hat{cs}}, 5_{\hat{ps}}\rangle - 0.168843\ |2_{\hat{cs}}, 6_{\hat{ps}}\rangle - 0.158763\ |2_{\hat{cs}}, 7_{\hat{ps}}\rangle - 0.264605\ |3_{\hat{cs}}, 0_{\hat{ps}}\rangle - 0.189004\ |3_{\hat{cs}}, 1_{\hat{ps}}\rangle - \\
&0.189004\ |3_{\hat{cs}}, 2_{\hat{ps}}\rangle - 0.199084\ |3_{\hat{cs}}, 3_{\hat{ps}}\rangle + 0.264605\ |3_{\hat{cs}}, 4_{\hat{ps}}\rangle - 0.168843\ |3_{\hat{cs}}, 5_{\hat{ps}}\rangle - 0.168843\ |3_{\hat{cs}}, 6_{\hat{ps}}\rangle - 0.158763\ |3_{\hat{cs}}, 7_{\hat{ps}}\rangle
\end{aligned}
$$

Here again, the probability of target state has decreased but evolving it further tells us that unlike Grover's algorithm the behaviour of this algorithm does not seem to be periodic.

3. What authors could have done more

Authors could have mentioned that they can go to any arbitrary probability close to 1 for n=2 by repeating the algorithm. They claim that this algorithm will give a target with probability $\frac{1}{2} - O(1/N)$ from which they can increase this probability by repeating the algorithm constant number of times by probability amplification. Probability amplification works when the probability of one event is relatively larger than the probability of other events but in this case, the probability of each state remains 0.25 no matter how many times the algorithm is repeated.
Authors have mentioned that this methodology can be used on any n-dimensional lattice and they will investigate optimality of such algorithms. They could have also mentioned what properties should a graph have to be the optimal candidate for this kind of search algorithm. They could have mentioned what will be the problems if this algorithm is made to work on a number line like the graph.

References

[1] Zhang, H., Raitoharju, J., Kiranyaz, S. et al. Limited random walk algorithm for big graph data clustering. J Big Data 3, 26 (2016)

[2] Cong Wan, Yanhui Fang, Cong Wang, Yanxia Lv, Zejie Tian, and Yun Wang, "SignRank: A Novel Random Walking Based Ranking Algorithm in Signed Networks," Wireless Communications and Mobile Computing, vol. 2019, Article ID 4813717, 8 pages, 2019.

[3] Ma, J., Fan, J., Liu, F. et al. J. Shanghai Jiaotong Univ. (Sci.) (2019) 24: 71. https://doi.org/10.1007/s12204-019-2041-2

[4] A Random Walk Approach for Efficient and Accurate Dynamic SimRank, Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong

[5] Jeong H, Qian X, Yoon BJ. CUFID-query: accurate network querying through random walk based network flow estimation. BMC Bioinformatics. 2017;18(Suppl 14):500. Published 2017 Dec 28. DOI:10.1186/s12859-017-1899-y