CS333 - Programming Assignment

Flexibility
Here I lay out a suggested algorithm to implement. (One that is pretty interesting!) However, if you would like to implement a different algorithm, or even try to design your own, that is very much acceptable. The learning goals here are to learn to implement a (somewhat complex) algorithm with a quantum programming language, so as long as you are meeting those goals, feel free to adjust. (If you do want to go your own way, please read over this document first to see the suggested scope of the project.)

Random Walker

In a classical walk on a line, we have a line graph and a "walker" that starts at a node on the graph. At every step of the algorithm, you flip a coin, and the walker moves to the node to the left or right of their current node based on whether the value of the coin is heads or tails. See this youtube video for the key properties of such a classical random walk. (The first 2.5 minutes give you most info that you need to know, and the rest provides an explanation for why.)

Now we can consider a quantum version of this classical walker. For the quantum walk, we need 2 subsystems, a coin, which is represented by a qubit, and the walker, which is represented by an n-dimensional vector. The combined system is represented by the tensor product of these two spaces. For example, the system can be in the state

$$|1\rangle_C|15\rangle_W \tag{1}$$

which means the coin is in state 1 (which we can interpret as Tails/Move Left), and the walker is at position/node 15.

The walk consists of two unitaries. The step unitary, U_s, looks at the value of the coin, and if the coin is $|0\rangle$, moves the walker to the right, and if the coin is $|1\rangle$ it moves the walker to the left. In other words, if $|p\rangle$ is a standard basis state of the walker, then

$$\begin{aligned} U_s\left(|0\rangle_C|p\rangle_W\right) &= |0\rangle_C|p+1\rangle_W \\ U_s\left(|1\rangle_C|p\rangle_W\right) &= |1\rangle_C|p-1\rangle_W \end{aligned} \tag{2}$$

The coin flip unitary U_f, applies a Hadamard to the coin qubit (it is like flipping the coin):

$$U_f\left(|c\rangle_C|p\rangle_W\right) = \left(H|c\rangle_C\right)|p\rangle_W \tag{3}$$

Start with the system initially in the state $|\psi_0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)_C|p\rangle_W$, where $|p\rangle_W$ is in the middle of the line graph. Then apply U_s and U_f in alternation k times. After k steps do a partial measurement of the W subsystem in the standard basis, which collapses the position of the walker

to a single position. (The W subsystem collapses to a single standard basis state based on the outcome of the measurement.)

Use a quantum programming language and a noiseless simulator to simulate this algorithm. (Note the simulator will only be able to run on small systems, so you won't be able to make your line graph too large!) If you repeat many times and plot a distribution of where the walker is after k steps, how does it compare to the classical distribution after k steps? In particular, compare:

- Where are you most likely to find the walker?
- What is the average distance away from the starting position as a function of k? (In the classical walk, this scales like $\sqrt{k}$.)

Can you explain in general terms, based on your understanding of quantum algorithms, what is causing the behavior of the quantum walker?

If you are using a language that has access to a simulator that includes realistic errors (noise), simulate with noise. How do those errors affect the output of your algorithm?

Implementation

Choose a quantum programming language/framework to use. I suggest one of the following, which are some of the most frequently used, and which seem to have the best documentation/tutorials. Based on what I'm guessing your backgrounds are, I've likely listed these in order of challenge.

- Qiskit (IBM), python framework
- Q# (Microsoft) programming language similar to F# and C#, Quantum Katas for learning
- CIRQ (Google) python framework - a slightly newer entry, so there might not be as extensive documentation/tutorials
- Quipper (academic), based on Haskell. See tutorial paper

The final submission should ideally contain code, sample outputs, distribution plots, perhaps analysis plots, and a discussion of the questions I list above.

For first three options you should be able to use jupyter to do this all in one notebook. You might have to do a zip file with Quipper(?)