Quantum Simulator: FRI

James Liu : JSL3486

April 2022

1 Background and Imported Libraries

My programming language of choice for the quantum simulator was Java, along with some external libraries to simplify some general operations such as evaluating expressions and complex numbers. I used the exp4j-0.4.8 expression evaluation library in order to simplify the process of evaluating the expressions in the Unitary gates, and I used code from https://www.math.ksu.edu/ bennett/jomacg/c.html to represent complex numbers and operate on them. Finally, I used general java imports such as java.util.* and java.import.* to handle data structures and user inputs.

2 Design Choices

My simulator utilizes five main classes: QubitString, Qubit, Gate, Runner, and Complex. The Complex class is imported from https://www.math.ksu.edu/ bennett/jomacg/c.html and is used to represent the complex values that are used in the gate operations. It stores the complex value as a real component and an imaginary component. Just to emphasize, this code was written by someone else and I am using it to simplify operations. The other four classes are written by me and are used to simulate specific parts of a quantum computer. To begin, the first class I will cover is the Qubit class. All the Qubit class stores is a integer value 1 or 0. Technically this Qubit class is wholly unnecessary, but I included it to make my program more analogous to quantum computing. Additionally, the Qubit class is the basis for the second class, QubitString. In order to represent the basis states, the QubitString stores an array of Qubits of length specified by the "qreg" instruction. It also stores a Complex number that represents the factor for that state; taking the conjugate square of the factor would then give us the probability of that basis state. The third class that I made is the Gate class that stores and applies operations onto Qubits. The Gate class stores five values: the name of the operation, the qubits that the operation uses (one qubit for most instructions, two for CNOT and SWAP, and three for Toffoli), and a 2x2 array to represent the matrix of the operation. For most gates, the matrix values are hard coded in; however, for the unitary, phase, rz,rx, and ry gates, the values are calculated using the exp4j-0.4.8 library

on the theta, phi, and lambda values given by the qasm code, which are then plugged into the unitary gate formula and Euler's formula. Additionally, the CNOT,Toffoli, and SWAP gate instructions instead only store the indexes of the qubits that they use. This leads us to the most important method in the Gate class: apply. This method applies the operation of the gate on the specified qubit for every QubitString in the program. For gates that aren't CNOT, SWAP, Toffoli, or barrier, the operation is done by matrix multiplication in a method called multiply. The end result is a 2x1 matrix where the first value represents the factor of the |0>component, and the second value represents the factor of the |1>component. I then return two new QubitStrings, and both have the same order of qubits as the original QubitString, except one has the chosen qubit as 0 and the other has it as 1. I then set the first QubitString's (the one with 0) factor value to the original factor * the factor of the |0>(the first value in the 2x1 matrix) and I set the second QubitString's (the one with 1) factor value to the original factor * the factor of the |1>(the second value in the 2x1 matrix). This allows the probability of each basis state to propagate properly between operations. I remove the original QubitString and replace it with the two new ones. One important thing to note is that if a probability is 0, then I remove it from the set of possible basis states. For CNOT, and Toffoli, I simply read the values of the qubits I need to measure, and if they are 1, I invert the given qubit. If the instruction is SWAP, I simply just swap the values of the two chosen qubits.

Finally, the class that actually reads in the QASM file is the Runner class. First, the program reads in the "qreg" instruction to figure out how many qubits, n, to use. It then creates a QubitString object with n bits set to |0>- this serves as the starting state of the system before any operations are applied to it. This QubitString, and an other future QubitStrings are stored in an arraylist called bstring. Then, the program reads in a QASM file line by line and for every instruction, it creates a Gate object and stores it in an arraylist called instructions. The Runner then runs the operate method, which iterates through every gate stored in instructions. Each gate then applies its operation to each QubitString in bstring. Finally, after applying every operation, the Runner then prints out each basis state and its probability. However, one issue that arises is that there will be many QubitStrings with the same bit pattern, and different probabilities. The proper probability for the unique QubitString will be the sum of all the duplicates. Thus,the program then runs through and sums up all the duplicates and removes them, leaving only unique QubitStrings and their correct factors. Using these values, we can then determine the final state of the simulation before measurement. Finally, the factors are conjugate squared to find the probability of each basis state.

3 Implementing Shots

Now we move onto implementing shots. Knowing the probabilities of each basis state, we can randomly generate a value between 0 and 100000 and use that

to calculate what state a shot will be in. My shot simulation works by assigning each basis state a range between 0 and 100000 proportional to the state's probability. Then, I see which range the randomly generated value falls into, which is then the value of the shot. I run this in a loop to simulate the multiple number of shots, and then print the number of times the random value landed in the range for each state basis.

4 Issues Encountered

One issue that occurs is that due to the rounding of the variables. This leads to inaccurate probability values as they are off by a slight margin. However, one could take a positive attitude about this and claim that this rounding error can represent quantum noise. Another issue that arose was the complications that came with the CNOT gate. Originally, I planned to store the qubits as an array with a 0 and 1 component, and then combine them to form the Qubitstring. However, this lead to issues with the CNOT as a CNOT leads to entanglement and entanglement cannot be "factored". Thus, when a CNOT was involved, it was impossible (or at least very difficult) to evaluate each qubit separately and then combine them. Thus, I came to the conclusion that I need to operate on the QubitString and store the probabilities rather than store the state of each qubit individually. This allowed me to easily implement the CNOT gate by simply checking the qubits of each possible state combination and applying the not to the chosen qubit if the condition is satisfied. Additionally, storing the qubitstrings instead of individual qubit states also for easy implementation of gates such as Hadamard that can put the qubit into superposition. This can be done by simply calculating the components of the chosen qubit after applying the gate to it, then multiplying its probability by the probability of the original qubitstring. For example, if the original basis state were |111>with a factor of 0.25, and a Hadamard were applied to qubit 0, then the resulting state would be 0.25*(0.707|111>+ 0.707|11>). This demonstrates the reasoning behind multiplying the factors when going into superposition. Another issue that I encountered was the issues of pass-by-reference. Because I store everything in a QubitString, I would run into issues when applying gates that lead to superposition, as it would modify one of the superpositions and modify the others at the same time. This was quickly solved by simply making a deep copy of the QubitString before doing any operations. A final large issue that I have not yet solved is implementing Non-unitary operations. This arises from the fact that I store the state of the system by their basis states rather than individual qubits. Thus, for operations such as reset that affect an individual qubit non-reversibly, I would probably need to revamp my Qubit class from the ground up.

5 Operations Implemented

For my simulator, I only implemented the Hadamard, Classical, Phase, and Quantum gates (minus RXX and RZZ).

CNOT	Z	X
H	I	T
S	Tdg	Sdg
Y	U	Barrier
Toffoli	SWAP	Phase
RZ	SX	SXdg
RX	RY	

6 Running the Simulator

The program is submitted in a zip file with a folder inside. This folder contains a jar file of my simulator and a file called "test.qasm". This qasm file is where instructions should be written. The jar file can be run using the command "java -jar QuantumSim.jar".

7 Reflection

All in all, I think this project really helped me understand the actual operations that we were doing during class. I think the most important part of this project was designing a method to implement the CNOT gate. My original idea of storing each Qubit and its state seemed the simplest at first, but when it encountered the CNOT, entanglement made me realize that I had to think about it another way. This pivot in how I had to store the state of the system also affected how I view these qubit operations when I do them on paper. Ultimately, I think that having students design their own quantum simulator is a great way to cement everything we have covered in the past semester.