Quantum Oracle Builder

Sidharth Srinivas — James Liu

November 2022

1 Premises

The input will be a 3-SAT CNF expression. Specifically, each input expression will be formatted as a conjunction (and operators) of clauses containing variables that are disjunctions (or operators) of each other. Additionally, each clause will contain exactly three literals. We use this to validate our input along with a series of ASSERTS to ensure that all inputted expressions follow these patterns.

Example: (!x1 | x2 | !x3) & (x2 | !x4 | x1)

2 Creating the Oracle

2.1 Setup

The starting point of translation is defined by a string containing the full boolean expression with the following symbols representing the operators.

- logical or — |
- logical and — &
- Complement — !

The string is then parsed into a list of separate clauses and into a dictionary that maps variable names to qubits for applying gates. Each clause is computed individually in the quantum circuit through a loop which separates each clause with two layers of barriers. During each iteration of the loop (Which is explained in 2.2 Procedure), the circuit is created from the clause in the following steps.

2.2 Procedure

First the compliments (!) are inputted into the circuit through X gates on the corresponding qubit in the dictionary. Afterword, the first two variables are operated on through an "or" and the corresponding CNOT and X gates are applied employing DeMorgan's Law logic. The result is stored in the output qubit and the same process repeats itself with the output qubit and the 3rd variable's qubit (dictated by the qubit dictionary). The new result is stored in an i indexed auxiliary qubit such that $0 < i <$ *number-of-clauses*. After storing the result in the auxillary, the output qubit is uncomputed to be used in future computations.

2.3 Finalize State

Once each clause is individually computed and stored in the auxiliary qubits, a multi-controlled X gate is applied that collectively operates an "and" on the auxiliary qubits and stores the final state in the output qubit. We then also apply the oracle again in reverse (minus the multicontrolled X) in order to reset the input and auxillary bits for future iterations of Grover's. This then completes the creation of our oracle. When using the oracle, the input qubits and output qubits are then measured onto classical qubits, before running a simulation with 1024 shots and displaying results on a histogram.

3 Classical Solver

3.1 Variables Needed

The main variables we need here are a list to store boolean values (T/F), a map to store relationships between variable names and where their value is stored (ex: the index in the previously mentioned list), and the actual expression that we are solving.

3.2 Methods

First, we need to find how many variables and clauses there are. We do this by splitting the expression along the "&" and storing the resulting clauses in a list. Then, we run through the list and start adding variables to our variable map. If the variable is already in the list, then it is not added; otherwise, it is added along with an index to represent where its value can be found in the value list. Once we have all the clauses and variables, we initialize our value list by creating a list of size equal to the number of variables, and setting all the values initially to false.

With these variables, we then move onto the recursive algorithm to actually find the answers to the expression. Essentially, what we are doing is generating every possible combination of values for the variables, and then checking if they satisfy the original expression. We do this with two methods: bool_recurse and solve_bool. Bool_recurse takes in an integer n to represent the number of variables, an integer i to represent how deep the recursion has gone, the value list, and a list of clauses. At each level of the recursion, the method sets the ith value of the value list to true at first, then recurses again. After that recursion finishes, it then sets the ith value of the value list to false and recurses again. For each of these recursions, the value of i is incremented to represent going down a layer. The method By doing this, we generate every combination of variable values. Whenever the recursion reaches it's lowest depth (when i equals n), then we call solve_bool.

Solve_bool uses the list of clauses and the list of variable values and returns if the values satisfy the expression. This is done with the following steps: for each clause, start with a boolean set to false, and iterate through the clauses. In each clause, split it by " | " to separate each variable and apply the OR operation on the variables value and the starting boolean; if the variable is negated (!) is negated, apply the inverted value instead. Save the end value of each clause in another list - we'll call this clause_list. After each clause is evaluated, we can start applying the AND operation to every value in the clause_list. We first create a new boolean set to false. We iterate through all the end values of clauses and set the boolean equal to the boolean AND the clause's value. After running through every clause's value, the final value of that boolean is the answer to the CNF, and that is returned.

Then back in bool_recurse, if solve_bool returns true, then we add our answer to a global answers list and return 1; this one indicates that a base case with an answer has been found. Whenever it isn't the base case, the method returns the total number of solutions that have already been found, which is stored in a variable called "total".

Following this, we now have the number of solutions and a list of what those answers are.

3.3 Validation of Oracle

We validate our oracle by first solving with our classical solver. Then we take our generated oracle, make a copy of it, and create another circuit with all the input qubits having H gate's applied and apply our copied oracle. We then simulate the circuit with 1024 shots, and add all the measurements with a 1 in the output bit into a list in bitstring form. This list is then compared with the list of answers from our classical solver to see if the oracle is valid or not.

Additionally, we wrote a script to generate random 3SAT boolean expressions. We applied several test cases to the code and verified that the quantum circuit was correct. However, once the number of qubits reached a significant number (around 8-10 or higher), the oracle became inaccurate which was difficult to manually pinpoint due to large amount of bitstrings.

4 Grover's Algorithm

After having our classical solver find the number of solutions, and generating our oracle from an expression, we can then use Grover's algorithm to find the solutions. We do this by simply running our normal Grover's algorithm with our oracle in place. First we create our diffuser; this is done by applying H gates, then X gates, then a multi-controlled Z gate, then X gates again, and finally H gates again to our input qubits. This diffuser is used to apply a negative phase to any state's that do not satisfy the oracle. We make our multi-controlled Z gate using the CCnZ method from lectures. We also calculate how many iterations of Grover's we need by using the following formula:

$$R = \lfloor \pi \sqrt{N/M}/4 \rfloor,$$

where N is 2 to the power of the number of inputs and M is the number of solutions (classically calculated). Then we start setting up our qubits for Grover's. First, we apply an X and H gate to the output qubit. Then apply H gate's to all the input qubits. Then, in a loop of length R, we apply the oracle, then the diffuser. Finally we apply an H and an X to the output and measure all the inputs. We then run a simulation with 1024 shots and display the histogram. We can see that our Grover's is indeed working as the correct answers are significantly more common than the incorrect answers.

5 Issues Encountered

There was some friction in parsing the string expression into a format that made it easy to input gates into the quantum circuit. Initially, each clause was stored individually and added to a fixed set of input qubits (3 due to 3-SAT);however, this led to some complications when test cases re-used variables in separate clauses. This resulted in an alternative solution which calculated and mapped all the variables to qubits alongside an array that collectively stored each clause individually. The mappings of variables to qubits are stored in a dictionary which is referenced above several times.

Another issue is that while our Grover's does amplify the correct solutions, we don't believe that it is doing it to the extent we would want it to. Originally, our Grover's wasn't working at all; it would sort of work for some cases and be completely incorrect for others. We figured out that the issue was that we were not uncomputing our auxiliary qubits leading to issues later down the road. Once we added our oracle in reverse to uncompute these qubits, Grover's started correctly amplifying answers; however, it still wasn't as drastic of a difference between correct and incorrect answers as we would have liked to see. We believe that the large size of our circuit is opening our circuit up to a lot of noise that may be reducing the circuits effectiveness.