Project 2: Study Group Scheduler

Checkpoint 1: 11:55 pm ~~Thu 3/14~~ Tue 3/19

Checkpoint 2: 11:55 pm ~~Thu 3/21~~ Tue 3/26

Final Deadline: 11:55 pm ~~Thu 3/28~~ Tue 4/2

[Direct Autograder Link](https://autograder.io/web/project/2433)

[Starter Code](https://drive.google.com/drive/u/1/folders/14iwuu_c1GDlbKYqJaqhA8Z-MXkLtDLu1)

[Useful Qiskit Functionality](https://docs.google.com/spreadsheets/d/1TIvXnpwQgypIvLhB_rzWuUDpXYHY9D9DnLMXsImokP4/edit#gid=0)

*Note: To ensure compatibility with the autograder, please ensure your code runs in Python 3.7.5. Yes, this is quite old, but it ensures the best support with the myriad of dependencies on the autograder.*

# Introduction

You have been tasked with designing a program to form study groups that meet a set of constraints. Being enrolled in EECS 498-001, you believe you can speed this task up by writing a quantum algorithm, specifically using Grover's algorithm and quantum counting.

When forming study groups, there may be multiple restrictions on what constitutes a valid group. For example, there may be a minimum size, we may want to avoid putting people with known time conflicts together, or we may want to guarantee that at least one student who is on track to pass the class is in each group.

A common way to express constraints (and the format we will be following in this project) is the  [Conjunctive Normal Form](https://en.wikipedia.org/wiki/Conjunctive_normal_form) (CNF), i.e. an AND of ORs. For example, given three students Richard, Leon, and Jon, the constraints that

* At least least two people are placed in the group, and
* Leon and Jon should not both be picked

can be summarized in the following CNF form:

(Richard or Leon) and (Richard or Jon) and (Leon or Jon) and (Richard or Leon or Jon) and (~Leon or ~Jon)

(How to construct CNFs from constraints is beyond the scope of this project and not something we will worry about: we will just assume the CNFs are already provided for us)

This is a small enough example that it's easy to inspect by hand and verify that there are 2 possible solutions which meet these constraints

1. Richard and Leon
2. Richard and Jon

Each instance of a variable (in its regular or negated form) is called a "literal" and each OR statement forms a "clause".

Your work for this project will be divided across several components and implemented using the Qiskit SDK.

# Part I: Oracle Design

[Reference](https://qiskit.org/textbook/ch-gates/oracles.html)

Inside oracle.py, you will implement a function to generate a "Bitflip Oracle" from a CNF formula, and a function to convert a Bitflip oracle into a "Phase Oracle". The CNF formula will be passed in as a list of integer lists, where each integer corresponds to a unique variable (negative values indicating negation of the corresponding variable). Elements in the inner list form a clause of ORed values, and all clauses in the outer list are ANDed together. For example:

[[1,2,3],[2,-3],[4]]

represents the formula:

(var1 or var2 or var3) and (var2 or not(var3)) and (var4)

**Note that because each variable must have an negated value, indexing starts at 1, not 0.**

For simplicity of implementation, you may assume that a variable does not appear for the first time (reading left to right) before a higher valued integer, no integers are skipped, and that a variable will not appear in the same clause more than once in either its normal or negated form (you do not need to check these conditions - you may assume they are always followed). For example, the following are invalid inputs and you may assume will never be passed in as arguments:

[[1,4],[2,3]] # 4 appears before 3

[[1,-2],[2,4]] # 3 is skipped

[[1,2],[3,-3]] # 3 appears twice in second clause

You also don't have to worry about empty CNFs for this project (i.e. every test is guaranteed to have at least one variable).

You have flexibility in how you design your oracles. In general, oracles can often be designed without the need for "ancilla bits" (bits used to hold an intermediate value), but you will likely find it simpler to include them. A straightforward solution is to store the OR result of each individual clause in a separate ancilla bit, and then AND each ancilla bit to store the final result. We recommend using the [MCX gate](https://qiskit.org/documentation/stubs/qiskit.circuit.library.MCXGate.html) (i.e. a multi-controlled X gate) which can be used to implement both multi-bit AND and OR gates (review [De Morgan's Law](https://en.wikipedia.org/wiki/De_Morgan%27s_laws) if this is not clear). Individual inputs can be negated by setting "ctrl\_state" to the appropriate bit-mask.

qc = QuantumCircuit(5)

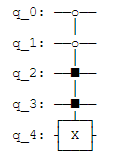
num\_ctrl\_bits = 4

mcx\_state = 0b1100 # invert all control bits except 3 and 2

gate = MCXGate(num\_ctrl\_bits, ctrl\_state=mcx\_state)

qc.append(gate, range(5))

The above code generates a quantum circuit which flips the state of q\_4 iff {q\_3,q\_2,q\_1,q\_0} = 4'b1100, i.e.:

****

Your bitflip oracle must read its inputs from the circuit's lowest indexed bits (with the higher indexed variables passed in via larger qubit index), the output should be stored in the next highest bit, and any ancilla bits should be placed on higher index bits. Any computation done on anything besides the target qubits must be "uncomputed" back to their original state so they can be reused for later computation.

For example, if we are provided a CNF for two variables and we use an additional 2 ancilla bits, the bits should be used as follows:

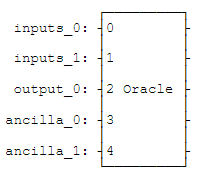
inputs = QuantumRegister(2, "inputs")

output = QuantumRegister(1, "output")

ancilla = AncillaRegister(2, "ancilla")

qc = QuantumCircuit(inputs, output, ancilla)

qc.append(oracle, range(5))



Where var\_2 is fed into inputs\_1 and var\_1 into inputs\_0 (the indices differ by one since the variable indices must start at 1, not 0). You should assume that all non-input qubits are initialized to |0> when passed into the circuit.

Your Phase Oracle must follow the same rules, except that there is no requirement of an "output bit" (the output is instead encoded in the relative phase of the state). However, the simplest solution is to keep the output bit, but prepare it in the |-> state, as described in lecture. You should assume that all non-input qubits are initialized to |0> when passed into the circuit and should be uncomputed back to |0> by the end.

# Part II: Grover's Algorithm and Quantum Counting

References [[1]](https://qiskit.org/textbook/ch-algorithms/grover.html) [[2]](https://qiskit.org/textbook/ch-algorithms/quantum-phase-estimation.html) [[3]](https://qiskit.org/textbook/ch-algorithms/quantum-counting.html) [[4]](https://qiskit.org/textbook/ch-labs/Lab06_Grover_search_with_an_unknown_number_of_solutions.html)

Inside grover.py, you will implement functions to create a single iteration of the Grover operator (i.e. phase oracle implementing the provided CNF followed by a diffuser), as well as a full Grover implementation for a specified number of iterations. You can use your own oracle functions to test these functions, **but they should work with any oracle implementations that meet the above specifications** (i.e. your Grover implementation should work with oracles containing any number of ancilla bits).

counter.py contains the prototypes for functions to implement a quantum counting circuit, so that you can estimate the number of solutions to a constraint problem. Note that this circuit must return an estimate for the number of **solutions**. Implementing the diffuser as described in class results in a phase that would calculate the number of non-solutions. To fix this, you will need to alter the phase of the diffuser by -1. A simple way to do this is by placing the sequence ZXZX in the circuit.

The [control](https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.control.html) method will be helpful for creating controlled versions of the Grover operator.

# Part III: Driver

Once the other components of the project are completed, you will have everything you need to implement the constraint solver. driver.py will be run with a command line argument specifying the name of a comma-separated-value (CSV) file describing the constraints.

Each row of the CSV file is a comma-separated list of names (optionally prefixed by a tilde (~) character to indicate its negation) which forms a single clause. Each row is ANDed together to form the overall CNF formula. For example, the contents of file test\_1.csv:

Richard,Leon,Jon

~Leon,~Jon

correspond to the CNF:

(Richard or Leon or Jon) and (~Leon or ~Jon)

Your driver should operate as follows, printing the specified messages to standard output when appropriate:

* Read in the CSV, and generate a corresponding CNF formatted array as described in Part I
  + Print "COUNT - Counting solutions for [N] variables..." where [N] is replaced by the number of distinct variables in the CNF
* Generate a quantum counter circuit using oracles implementing the CNF, using a precision of 5 bits in order to calculate the expected number of solutions to the constraint problem
  + Print "COUNT - Estimated number of solutions: [S]" where [S] is replaced by the estimated number of solutions (**not** rounded), with two decimal places.
* If the number of estimated iterations needed for optimality is less than 1 (this may or may not correspond to the solution space taking up half or more of the search space), then a dummy variable (set to false) should be added to the CNF and the quantum counting algorithm should be rerun
  + Print "COUNT - Solution space too large, rerunning with additional variable"
* If the number of expected solutions (rounded to the nearest integer) is 0, the program should terminate.
  + Print "COUNT - No solutions expected, exiting"
  + EDIT: This check should take priority. You don't need to do any other checks (like whether the number of iterations is less than 1).
* Otherwise, print the estimation for the number of iterations (using the rounded number of solutions above, **but not rounding** the number of iterations)
  + Print "COUNT - Estimated number of Grover Iterations: [I]" where [I] is replaced by the estimated number of optimal iterations
* Grover's algorithm should then be run using the number of iterations estimated by the quantum counter, rounded down to the nearest integer
  + For this calculation, you should round the number of expected solutions to the nearest integer
  + Print "GROVER - Running search with [I] Grover iteration(s)" where [I] is replaced with the number of iterations
* The returned value from Grover's algorithm should be checked for correctness. If verified, a solution has been found and the program ends.
  + Print "GROVER - Solution identified: " followed by a space-separated list of names that satisfy the constraints (for simplicity, leave an extra trailing space at the end). Any valid solution can be accepted. **Each name should appear in the order they first appeared in the original CSV file**
* If a solution is not found, the algorithm should be run again for a maximum of 10 times before exiting.
  + Print ""GROVER: No solution found after 10 attempts"
  + NOTE: this stipulation is just made for completion sake. The probability of not measuring the correct solution after 10 attempts for our test cases is very low and is not something we will be checking.

EDIT: It's recommended that you keep the number of "shots" low for these experiments low to avoid timing out. In particular, you probably only need one or two shots for running Grover's algorithm.

For the given file test\_1.csv

Richard,Leon,Jon

~Leon,~Jon

The output should be (blank lines are optional and ignored):

COUNT - Counting solutions for 3 variables...

COUNT - Estimated number of solutions: 4.78

COUNT - Estimated number of Grover Iterations: 0.99

COUNT - Solution space too large, rerunning with additional variable

COUNT - Counting solutions for 4 variables...

COUNT - Estimated number of solutions: 4.94

COUNT - Estimated number of Grover Iterations: 1.40

GROVER - Running search with 1 Grover iteration(s)

GROVER - Solution identified: Richard Jon

Any of the other four possible solutions are also valid.

test\_2.csv gives an example where no solutions are possible:

Richard

Leon,Jon

~Richard,~Leon

~Richard,~Jon

~Leon,~Jon

The output should be:

COUNT - Counting solutions for 3 variables...

COUNT - Estimated number of solutions: 0.00

COUNT - No solutions expected, exiting

# Restrictions

While you are encouraged to reference these for your own testing and understanding, your submitted code may not use the following Qiskit libraries:

* qiskit.circuit.classicalfunction
* qiskit.circuit.library.PhaseOracle
* qiskit.circuit.library.GroverOperator
* qiskit.circuit.library.QFT
* qiskit.algorithms

Otherwise, you may use anything in the Qiskit SDK and the numpy, math, random, and unittest packages.

# Testing

You must provide a set of test functions written in tests\_p2\_oracle.py and tests\_p2\_algorithms.py to the autograder. tests\_p2\_oracle.py should contain unit tests for each method in oracle.py and tests\_p2\_algorithms should contain unit tests for each method in ~~oracle~~ counter.py and grover.py. You must use the [Unittest model](https://docs.python.org/3/library/unittest.html) discussed in lab. Your tests will be graded on whether they cause assertion failures when run on buggy solutions, but do not cause assertion failures on correct implementations.

# Submission and Grading

Submit oracle.py, grover.py, counter.py driver.py, tests\_p2\_oracle.py, and tests\_p2\_algorithms.py to the autograder using the direct link at the top of this page.

Because this project is larger in scope than P1, there are 2 checkpoints worth a moderate amount of your overall grade to let you know if you are on track to finish:

* Checkpoint 1 is worth 5% of the overall project grade. It will be calculated using your score for the oracle public tests, private tests, and mutation tests (i.e. the contents of oracle.py and tests\_p2\_oracle.py).
* Checkpoint 2 is worth 5% of the overall project grade. It will be calculated using your score for the algorithm public tests, private tests, and mutation tests (i.e. the contents of grover.py, counter.py and tests\_p2\_algorithms.py).
* The final submission is worth 90% of the overall project grade. It will be calculated by running all tests (including those from the checkpoints and the driver). Therefore, you can still earn some points that you missed from the checkpoints.

We will grade your code on functional correctness. As a reminder, you may not share any part of your solution with others. This includes both code and test cases. You are however encouraged to discuss the projects in a way that does not involve sharing code. You will get feedback on your total score, but you will not have access to what the private test cases are checking for.

Efficiency is not graded, but your code must complete in a reasonable amount of time. Note that for general quantum circuits, simulation takes an exponential amount of time. We will only grade your code on CNFs with up to 4 variables and 4 clauses.

Due to rounding, your unitary matrices and state vector calculations may slightly deviate from the correct answer. We will check that every value calculated is within .00001.

Your driver is the only design file that should add measurements to your circuits. The private tests will assume that you do not already have measurements on the circuits produced by oracle.py, grover.py, and counter.py. The private tests will fail if you already have measurements added or classical bits in your circuit. Accordingly, your test functions should add measurements when necessary.

In addition to checking simulated output, we will also test your code by checking the unitary matrices corresponding to your circuits\*. You should therefore ensure that any phases match the documentation and that you properly uncompute when necessary.

*\*Note that because the spec offers some ambiguity in how you implement your functions, we will only check sub-matrices corresponding to bits that are actually measured. We will also give credit for any matrix that is within a global phase of the expected output.*