THE CLASSICAL BOIS

QUANTUM PROGRAMMING

Madhav Sankar Krishnakumar madhavsankar@ucla.edu

Parth Shettiwar parthshettiwar@g.ucla.edu **Rahul Kapur** rahulkapur@g.ucla.edu

March 27, 2022

Contents

ABSTRACT

We write the code in Qiskit for six popular quantum algorithms (Deutsch-Jozsa, Bernstein-Vazirani, Simon's, Grover's search, Shor's and QAOA algorithms) and run it on an IBM Quantum Computer. For each of the algorithms, we benchmark the performance of our execution, measuring the scalability of compilation and execution with the number of qubits and dependence of the execution time on the oracle U_f. We then discuss our code organization and implementation of quantum circuits.

1 Design and Evaluation

1.1 Execution Backend:

We tried running most of the code on IBM Quantum computers. For programs requiring less than 5 qubits. we have done this. We also tried to run on IBM Backend Simulators but for more that 5 qubits, it generally returned a task too long error. We also tried to copy the noise model of an IBM Quantum Computer and apply it over Qiskit Simulator. This again enforced the qubit count limitation in the simulator. Therefore we used a custom noise model that is very close to the real IBM Computers. We have applied depolarizing single qubit and multi-qubit errors and even examined the code execution by varying these parameters.

1.2 Simon's algorithm

1.2.1 Problem Statement:

Given a function f : $\{0,1\}^n \rightarrow \{0,1\}^n$*, there exists* $s \in \{0,1\}^n$*, such that* $\forall$ *x,y [f(x) = f(y)] iff (x+y)* $\in \{0^n, s\}$*. Return s.*. The following circuit is used to solve the quantum part of Simons's Algorithm.

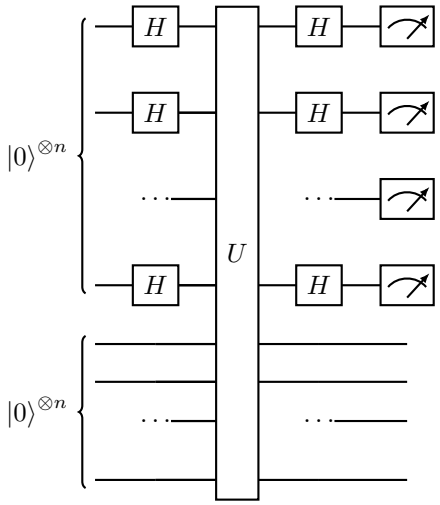


Figure 1: Simon algorithm quantum circuit

The overall process to solve the Simon's problem is as follows:

- First pass an all 0 input to a quantum circuit, described in above figure, which produces the desired bit strings which are orthogonal to s. Do this, n-1 times, where n is as defined in problem statement.
- Check whether the n-1 bit strings are linearly independent. If not, go to step 1.

- Having n-1 linearly independent equations, find a vector orthogonal to all of them. This is our required s. Do a manual check on function f for 2 values, 0^n and s. If f(0^n) and f(s) are equal, then return same s as final answer. Else, return $s = 0^n$

Points to note:

- If n = 1, we solve it classically only, by just observing the function values are different or same for the 2 possible inputs. If same, then output $s = 1$, else output $s = 0$.
- The classical solution requires $\Omega(\sqrt{2^n})$ iterations to get the value of s, which is exponential, the Simon's algorithm solves it in linear time approximately. Specifically, if we run the algorithm for $4m(n-1)$ iterations where m is such that probability of successfully finding the s is atleast 1-e^{-m}.
- As we can see, the algorithm is a hybrid algorithm of quantum and classical parts. We later analyse the times taken by each of them
- The step 1 of Simon's algorithm generates bitstrings y such that $y \cdot s = 0$.

1.2.2 Implementation of U_f:

- The blackbox function was written by creating a matrix of size $2^{2n} \times 2^{2n}$, which maps all the possible inputs in $\{0,1\}^{2n}$ to $\{0,1\}^{2n}$.
- The blackbox function works as follows:

$$U_f : Qubit^{\otimes 2n} \rightarrow Qubit^{\otimes 2n}$$
$$U_f|x\rangle|b\rangle = |x\rangle|b \oplus f(x)\rangle$$

- The implementation of this was done in python, by iterating over all possible inputs. The index was converted to binary representation, and then fed to function to get $f(x)$, which was then operated in the way described in formula above, to generate the output. The (i, j) entry of U_f was set to 1, if the decimal value of iterating input value = i and output of above operation (converted to decimal value) = j.

1.2.3 Design considerations:

- Design of function f: A random function was generated according to a randomized generate s for given value of n. If $s = 0$, then a one to one function was created, otherwise we always generated a two to one function. For generating one to one function, simply stored all the possible bit strings in an array and permuted them to create a one to one mapping with inputs. For two to one function, iterated over all possible inputs, checked whether the current input has been assigned value or not. If not, then assign a value to it, which is not been assigned to any other input, and assign the same value to $input \oplus s$. The function is easy to read and is implemented in few lines of code in python.
- The quantum circuit created in qiskit according to Figure 1. The first n lines were measured and used for getting the bit string orthogonal to s. The quantum circuit was ran until n-1 independent bit strings are obtained. To check the independence of bit strings, we used the numpy matrix rank calculator and checked whether the matrix of bit strings is full rank, $i.e.$ equal to n-1. For this we had first converted the output to Galois Field order 2, to compute all operations in modulo 2.
- Getting independent linear equations, the solution to getting a vector orthogonal to all of them was modelled as a Boolean SAT problem, where we model the dot product of s and bit string as XOR of the individual product of bits of the two bit strings and equated to 0. These constraints were put in the SAT solver and output was noted. The library `ortools` was used for solving the SAT problem.
- To simulate the program on quantum computer, it was infeasible to carry out simulations above n = 2. The reason being that IBM quantum computer supports maximum 5 qubits, and for n-space function in Simons, we need 2n qubits. Hence for n = 3, we need 6 qubits which is not possible to carry out on IBM quantum computer. Hence we develop a noise model which models the depolarizing error prevalent in IBM quantum computers and applied to all gates.

1.2.4 Code readability:

The whole code was divided into multiple functions. Have used the parameter of `verbose`, which prints all the intermediate steps of the program if set to True. The following are the main functions and their purpose:

- `getFx(n,verbose)`: Generates a random function. 2 cases are taken, $s = 0$ and s not equal to 0. Outputs the function values as a list, where the ith index contains the output of function for binary representation of i as input to function.
- `createUf(n,func)`: Takes n and function as inputs and generates the matrix U_f. The implementation is discussed above.
- `ver(n)`: The function verifies the U_f matrix, whether each entry of the matrix is correct or not. If not correct, it prints the input for which it was wrong. The verification is done by randomly creating a function using `getFx` and iterating over all possible inputs and checking it with matrix output.
- `runMainCircuit(n,verbose = True,withNoise = 0,p1=0.01,p2=0.1)`: This is the main function which calls all other functions. Takes n as input, and first generates the randomized function, then creates the oracle, then creates the quantum circuit and finally executes the quantum circuit to generate the bit strings. After getting the bit strings matrix, passes it to SAT solver which returns the s value. All intermediate steps are printed if verbose set to True. If withNoise = 1, noise is added to all gates. p1 and p2 tell the noise probabilties to be added to single and multi qubit gates respectively.
- `SAT_Solver(arr)`: Takes the matrix of independent bit strings which are orthogonal to s, of size $n - 1 \times n$. The output is the value of s which is orthogonal to all above bit strings. The function calls the class `sat_solve_multiple` which returns all the possible solutions, in case more than 1 solution exists (Note: s = 0 is always a solution).
- `get_accr(bits,s)`: To get accuracy of the quantum circuit. It computes the number of strings which have a dot product with s = 0. For a simulator, this comes to be 1, but for a noisy model or actual IBM quantum computer, accuracy drops.

1.2.5 Parametrizing the solution in n

Given n, we first generate the s value, followed by using that s to generate a random function. We then use the function to create the U_f matrix. this works for any arbitrary n and hence scalable. The time however is exponential with n.

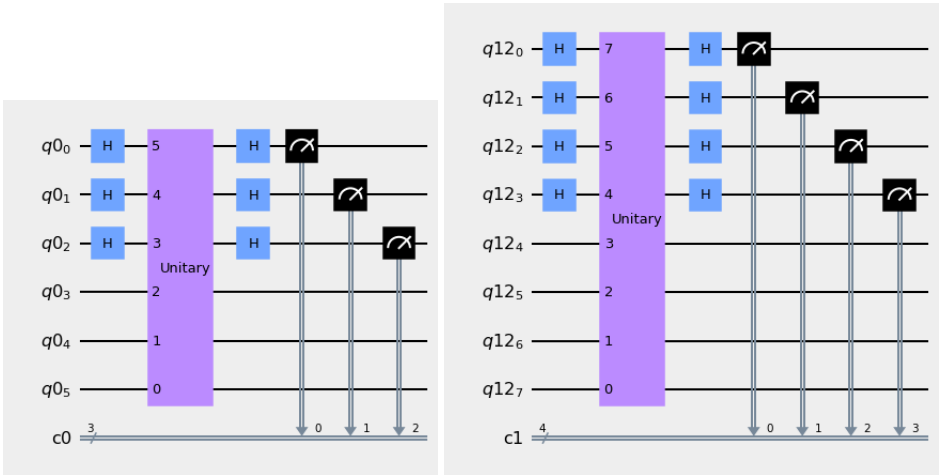


Figure 2: Circuits for Simon programme generated in qiskit for n = 3 and 4. The last line is the classical register line which stores the output of circuits.

1.2.6 Testing

The testing was done rigorously on the amount of time taken for various components of circuit. Specifically, we divided the total time taken into three parts: 1) Time taken for generating the Uf matrix, 2) Time taken to execute the quantum circuit and 3) Time taken to solve the independent bit strings using SAT solver. We do this across n, i.e. we increase values of n and note down the values. Each case is executed 30 times and averaged out the time over these 100 iterations. The histograms are also plotted for each of the above three parts for fixed value of $n = 3$. We further plotted the plots where we observe the number of iterations required to get the correct value of s. An ablation was also made to observe the effect of noise on the circuit.All the analysis is done on qiskit simulator with and without noise. Following are the results:

1.2.7 Variation of circuit Time with n

: In this experiment, we computed the various circuit times as we increase n. All analysis was by averaging over 30 iterations and n was varied over 2,3, and 4.

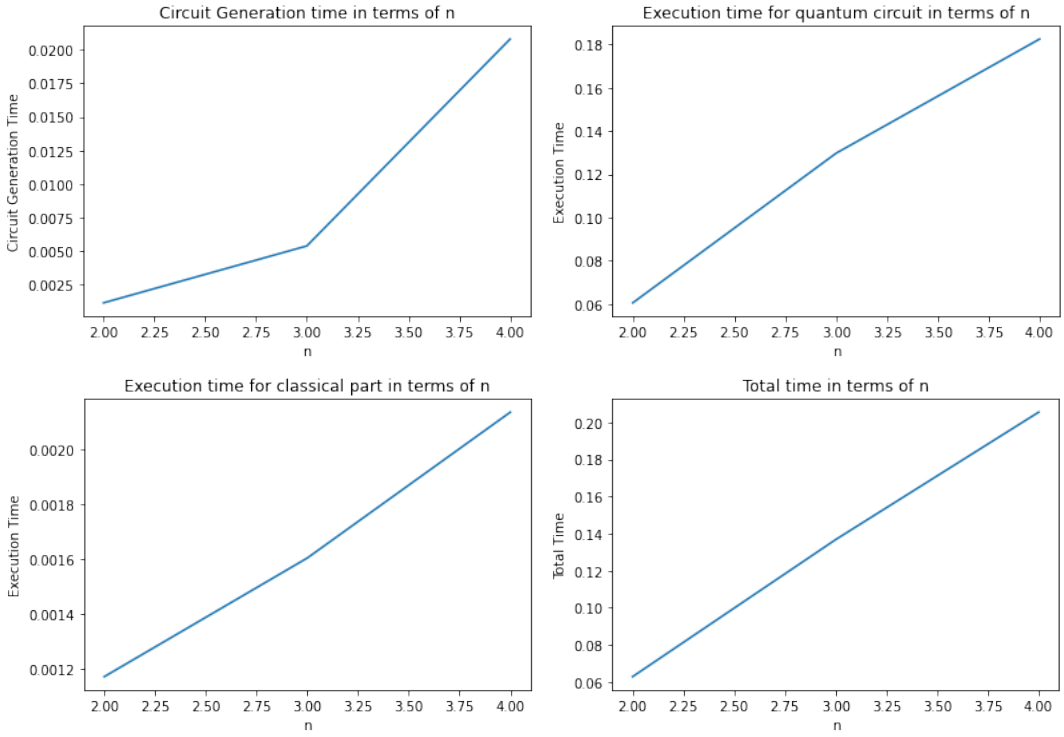


Figure 3: Execution Times for different parts of circuit in qiskit simulator

- We can see clearly that matrix generation and quantum circuit execution time rises with n, which is consistent with what we expect. As n increases, matrix size increases exponentially and hence the observation. Also as n increases the amount of time increases for processing the qubits in the circuit, and hence the observation.
- Similar observation is made for the execution times of classical part, SAT solver. However, its noted that it would depend much on the equations to be solved. As n increases, the time required to solve the increasing number of equations also increases and hence the observation.
- Of all the times, the quantum circuit times have the highest value and which is expected, since we quantum processing is slow.
- Lastly, the total time taken as we can see is increasing with n mainly due to quantum circuit times.

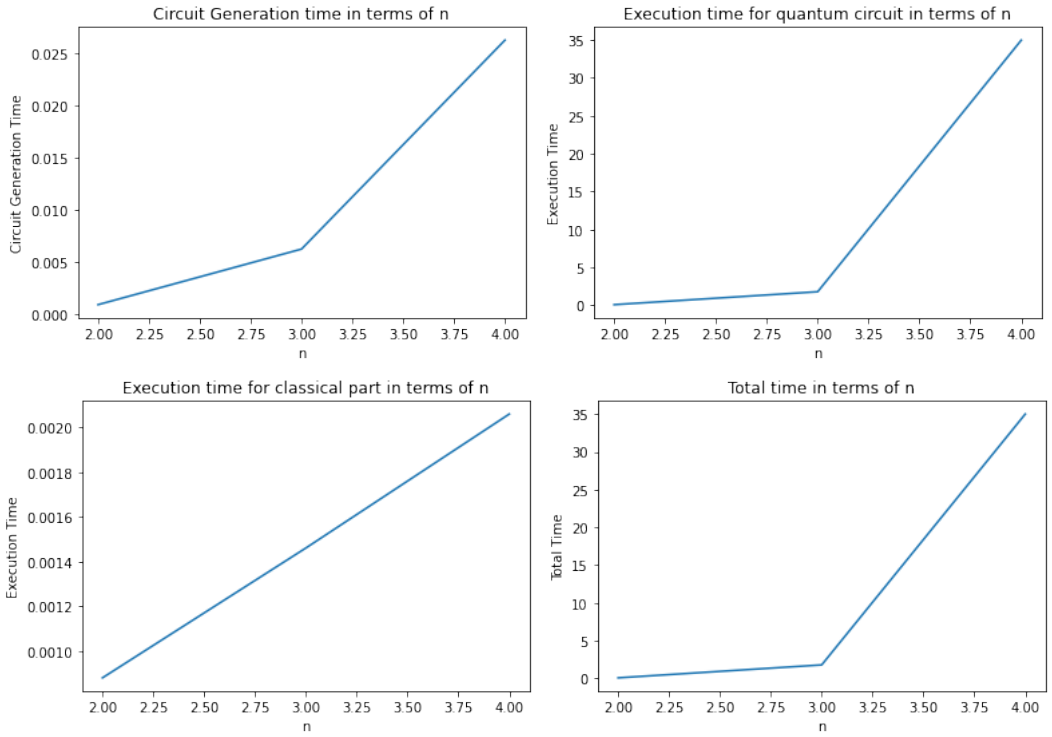


Figure 4: Execution Times for different parts of circuit with a noisy model

- Again all the times increase with n. An interesting observation can be made that, times increase exponentially with n for quantum circuit processing. The major reason behind is that as n increases, the noise added affects the space of equations, that getting n-1 independent linear equations becomes difficult (the noise added is random, however set of feasible space of equations is continuous, hence random noise takes the output to outside feasible set often)
- The total circuit time, again depends a lot on quantum circuit times as it is the highest among all times.
- As compared to noiseless model, the circuit generation and classical part takes similar times, but quantum circuit processing times increases 100x. Thats a huge difference. The reason is simple again, as explained before, adding noise makes it difficult to get independent set of equation as n increases and hence we run for large number of iterations before we get the required set of orthogonal vectors to s.

1.2.8 Variation of Histogram curves for different parts of circuit

In this experiment, we see the histograms for various circuits and see the variability in times taken for n = 3 across 30 iterations.

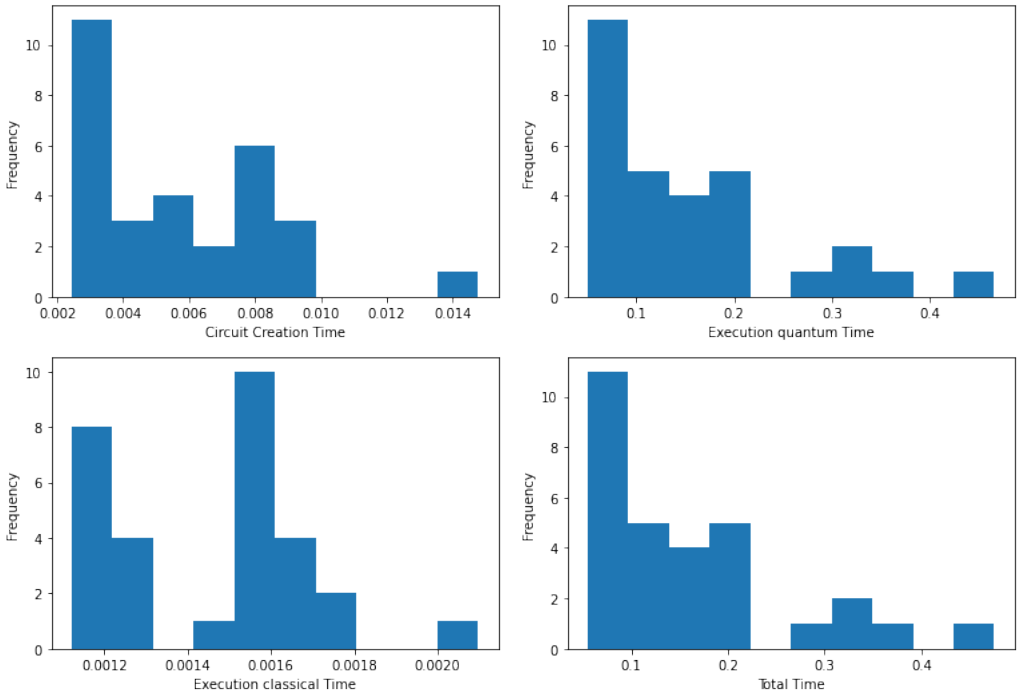


Figure 5: Histograms for circuit times for 30 iterations for n = 3 for noiseless model

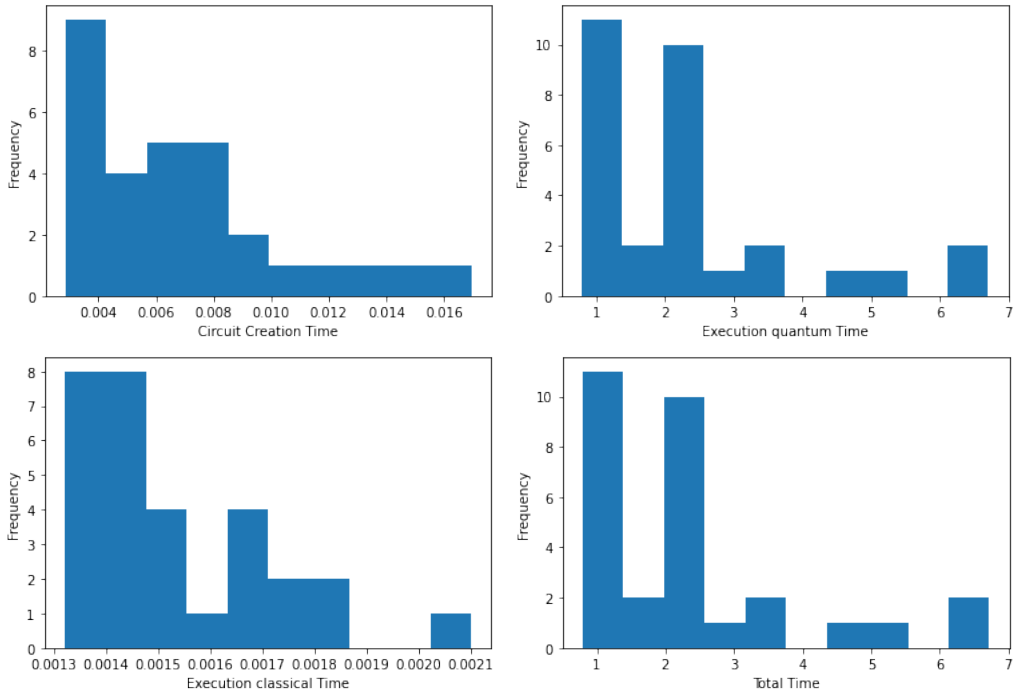


Figure 6: Histograms for circuit times for 30 iterations for n = 3 with noise

- First we observe that, the deterministic part which is circuit creation, has minimum variability in times taken, and almost all times are centred in range 0.002-0.01.

- Similarly, the classical circuit has more variability than circuit creation. This is mainly due to fact that everytime different sets of independent linear equations need to be solved and depending on the algorithm, the time taken would vary to some extent to solve them. Its observed that almost all times are centered in range 0.0012-0.002.
- The quantum circuit times has the maximum outliers and variability, with histogram spread most uniformly. This clearly depicts the randomness associated with the quantum circuit. We see in case of noiseless model, the times range from 0.05-0.4 and in case of noisy model we have more variability of 0.5-7. One of the major reasons is that everytime we have to find n-1 linearly independent equations and its by uniform probability, as computed in Simon's solution, for any of orthogonal vectors to pop up in measurement. Hence we cant guarantee, when will we achieve n-1 linearly independent equations with certainty. In case of noisy model, this further increases since in addition to above problem, we are also not guaranteed when we will get the n-1 orthogonal vectors to s.

1.2.9 Variation of number of iterations used with n

In this experiment we varied n and tried to find the number of iterations it takes for quantum circuit to generate n-1 linearly independent bit strings.

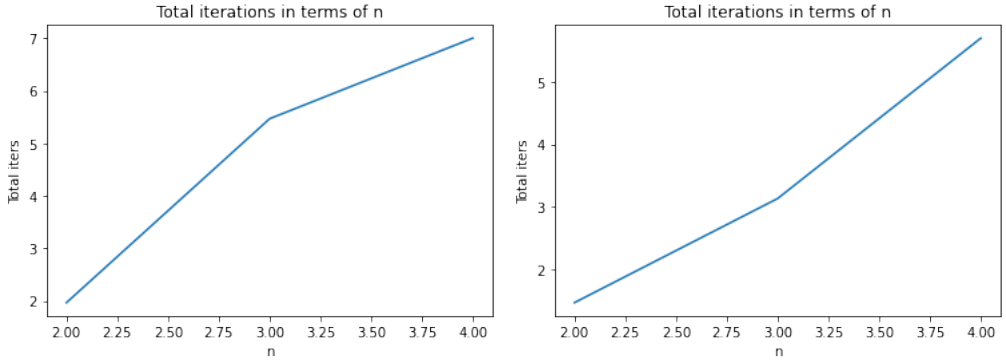


Figure 7: Variation of number of iterations with n. Left: Noiseless model, Right: Noisy model

We ran for each case of n for 30 iteration and averaged out. As we can see, the curve is linear. The major reason behind this is that there are 2 product multipliers while computing number of iterations. As we increase n, the number of iterations to run the simons circuit is proportional to (n-1), to generate n-1 linear equations. At the same time, as we increase n, the probability to find the n-1 linearly independent equations decreases. Hence, we need more of iterations to get those n-1 independent equations. Hence for both noisy and noiseless model, we see increase in number of iterations with n.

1.2.10 Variation of accuracy of circuit output with n

In this experiment we tried to see the effect of noise on accuracy. Accuracy here is defined by whether the quantum circuit generates output which are orthogonal to s. In case of simulator, we get accuracy 1 as its perfect simulations.

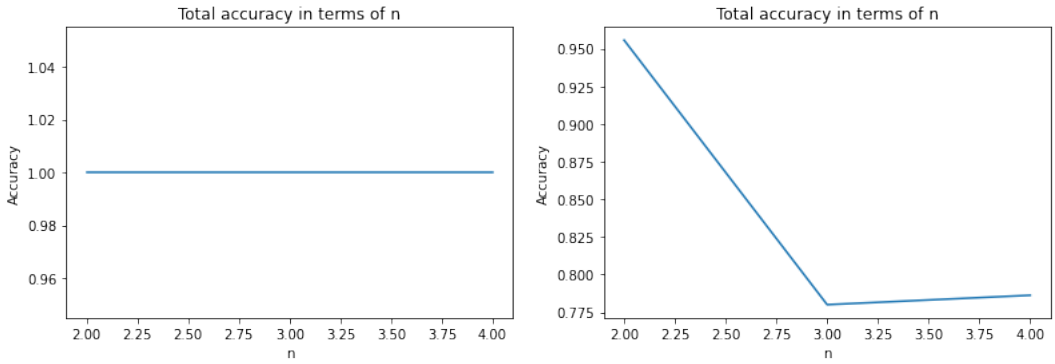


Figure 8: Variation of accuracy of circuit output with n. Left: Noiseless model, Right: Noisy model

We clearly observe that for qiskit simulator simulations, noiseless model, we get accuracy 1 as expected. This is because thats how the simons algorithm is devised. It always produces orthogonal vectors to s. For noisy model, we see that accuracy drops by a big margin as we increase n. Since we tried only for n=2,3,4, hence an exact curve cant be deciphered from this set. But general observation is accuracy decreases as we increase n. The reason behind is that as we increase n, it starts to get difficult to produce n-1 orthogonal vectors to s and hence every run we get non-desired output. And we know as we increase n, the number of iterations also increase, hence number of non-desired outputs also increase and hence dip in accuracy

1.2.11 Variation of accuracy of circuit output with increasing noise

In this experiment we varied different depolarization noise levels and checked the accuracy for 2 cases: adding noise to single qubit case and adding noise to multi qubit case.

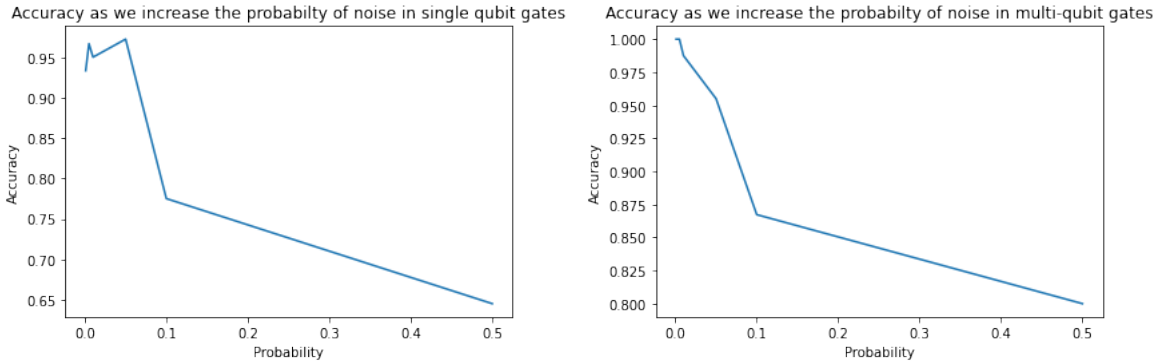


Figure 9: Variation of accuracy with increasing noise (depolarization probabilities. Left: adding noise to single qubit gates, Right: adding noise to multi-qubit gates)

- We clearly observe that as we increase noise probabilities, the accuracy drops in both case of multi and single qubit cases. This is because, increasing noise makes it difficult for the circuit to produce orthogonal vectors to s
- Adding noise to single qubit gates, produces a bigger dip in accuracy than multi qubit case. This is because the number of single qubit gates is 2n (all hadamard gates) and there is only 1 multi qubit gate (custom unitary matrix). Hence more gates imply more noise to these gates and hence greater dip in accuracy.

1.2.12 Code well designed for improved usability or ease of understanding

The code is divided into clear functions. Apart from the basic features we have added the functionality to enable or disable verbosity. Moreover, we have added the feature to pass the n or get it from the user at the run time to enable better usability. To make the code easy to understand, clear and precise comments are added throughout the solution.

1.3 Grover's search algorithm

First, we summarize the problem statement:
Given a function f : $\{0,1\}^n \to \{0,1\}$ *that returns* $f(x) = 1$ *for exactly* $|x| = a < n$ *inputs, find one of the inputs x that make f return 1.*

Solving this problem on a classical computer requires 2^n queries to f, but with Grover's algorithm we can solve it with high probability with $O(\sqrt{2^n})$ operations of the oracle Z_f. The algorithm involves the so-called Grover's operator G, defined as

$$G \coloneqq -H^{\otimes n} Z_0 H^{\otimes n} Z_f \left|x\right\rangle, \tag{1}$$

with Z_f and Z_0 defined through their action on computational basis vectors $|z\rangle$ as follows:

$$Z_f \left|x\right\rangle = (-1)^{f(x)} \left|x\right\rangle, \tag{2}$$

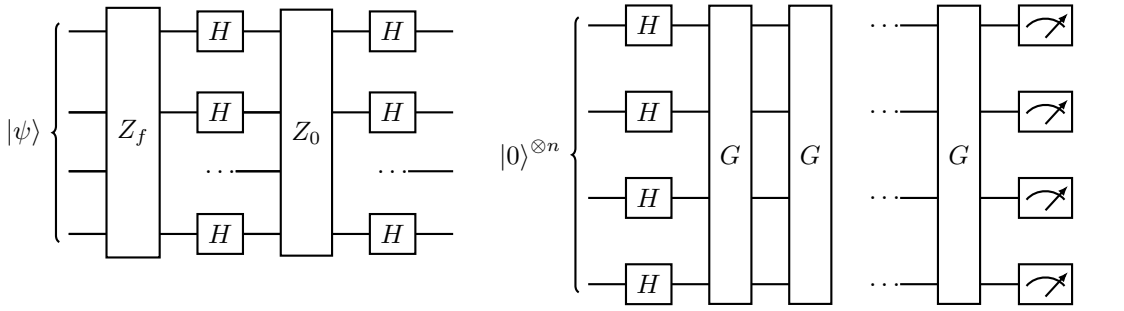
$$\text{and } Z_0 \left|x\right\rangle = \begin{cases} -\left|x\right\rangle & \text{, if } \left|x\right\rangle = 0^n \\ \left|x\right\rangle & \text{, if } \left|x\right\rangle \neq 0^n. \end{cases} \tag{3}$$

Grover's operator, G, is shown in Fig. 10a. In order to maximize the probability of success, the operator G is applied k number of times, with k being

$$\begin{aligned} k &= round\left(\frac{\pi}{4\theta} - \frac{1}{2}\right) \\ \text{with } \theta &\coloneqq \arcsin\frac{a}{N}. \end{aligned} \tag{4}$$

Here, $round(y)$ denotes the integer closest to y. The complete circuit is shown in figure 10b. The probability of success (i.e. the probability of the final measurement resulting in a bitstring satisfying $f(x) = 1$) is given by

$$P_{success} = \sin^2((2k+1)\theta). \tag{5}$$



(a) Grover's iteration G.

(b) Grover's circuit: Grover's operator G occures $k = \mathcal{O}(\sqrt{N})$ times.

Figure 10: Grover's search algorithm

Now we discuss various aspects of our code design:

1.3.1 Implementation of U_f, i.e. G

To implement G, we need to implement Z_f and Z_0.
Implement Z_0:

1. As Z_0 reverses the sign for only 0*n, we can make of the CZ gate. We know that CZ gate changes the sign for 1*n.

2. Apply X gate to all qubits. So 0*n becomes 1*n and every other state is not 1*n.

3. Apply CZ gate now to make 1*n negative.
4. Apply X to all qubits. Now the only negative one is 0*n.

Implement Z_f:

1. We pass the list of x that have f(x) = 1. We need to negate all these values.
2. We use a similar logic as Z_0. If any qubit of the given x is 0, apply X gate to it. Now the given x alone is 1*n.
3. Apply CZ gate now to make 1*n negative.
4. Apply X to the qubits in x that were originally 0 to bring them back to 0. Now the only negative one is the original x.

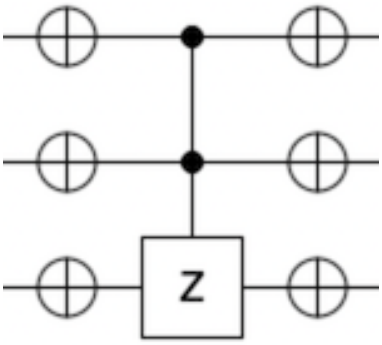


Figure 11: Grover Z0 implementation for n = 3.

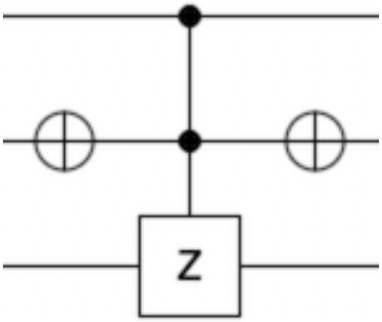


Figure 12: Grover Zf implementation for n = 3 and f(101) = 1.

1.3.2 Parametrizing the solution in n

The circuit is different for every value of n. In the function `runMainCircuit(n, num_ans, verbose)`, we pass in a dynamically created Z_0 *and* Z_f by calling `createZf(qubits, fnlist)` and `createZ0(qubits)` with an arbitrary value of n. We generate number of iterations dynamically as well. These solutions help to parameterize the code.

1.3.3 Code Reusability

We split our code for Grover's algorithm into multiple small functions, each of which executes a simple task.
We will briefly describe the role of each function in our code below:

1. `getFx(n,verbose,num_ans)`: Generates a random function to be implemented. It creates a function of 'n' bits with 'num_ans' as the count of x that satisfy f(x) = 1. We also pass a verbose parameter to decide if the print statements need to be displayed. This was added to avoid large verbosity when re-running the algorithm for measure the efficiency.

2. `createZ0(n)`: Creates Z_0 gate using the method described above.

3. `createZf(n, fnlist)`: Creates Z_f gate using the method described above. fnlist contains the list of x such that f(x) = 1.

4. `bitstring(bits)`: Converts number to bit string, so that the histogram is more readable.

5. `runMainCircuit(n, num_ans, verbose)`: This is the main code that implements the entire algorithm. We first create the the number of iterations using the formula mentioned above. We then create n qubits and generate a random function to implement. Now we move forward to implement the circuit by creating z_0 and z_f. We then use the simulator to run the code 100 times and plot the results using a histogram. We then run the code in the IBM Quantum Computer (ibmq_quito) for 100 times. We generate 3 times - time needed to create circuit, time required for simulation and time required for execution on the quantum computer (This is extracted from the properties returned as part of the results).

6. `expectedProbability(n, num_ans, numIterations)`: This measures the expected probability of success.

By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code more readable and reduce code replication.

1.3.4 Code testing

Code Correctness: Grover's algorithm is probabilistic, succeeding with a high probability, but with a small probability of failure. In order to test that our implementation succeeds at a rate similar to the theoretical success rate, we implemented our code for a variety of combinations of n and a. For each combination, we ran our code over 100 times, recording the fraction of times the algorithm succeeded. The theoretical results match the numerical results obtained from simulation very well. But from the table below, we can observe that the results from a quantum computer are way different. This is because of the immense noise present. We can also observe how the success rates fall as we try for larger qubits. More the qubits and gates involved, much larger the noise and hence the success rates are pretty low. It is quite staggering to notice that even for n = 4, the success rate falls to a mere 10% from the expected 96%.

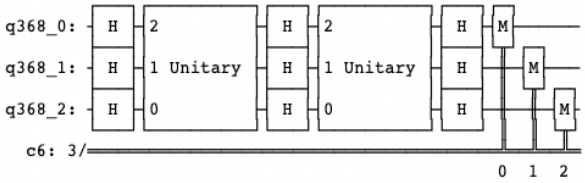
n	N	a	Theoretical success rate	Simulation success rate	Quantum Computer success rate
2	4	1	100%	100%	91%
3	8	2	100%	100%	41%
4	16	1	96%	97%	10%

Table 1: Grover's Algorithm: Success rates compared to theoretical expectations.

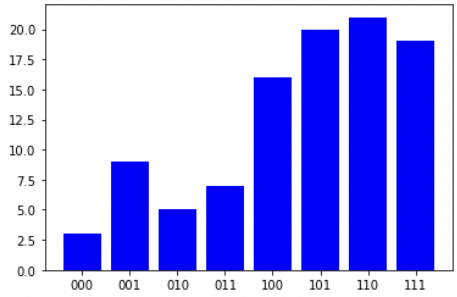
```
Input Length:  3
Number of iterations:  1
Number of x such that f(x) = 1:  2
1 th one is:  101
2 th one is:  110
Iteration no:  0
```

```
ibmqfactory.load_account:WARNING:2022-03-06 00:03:24,569: Credentials are already in use. The existing account in the
session will be replaced.
```

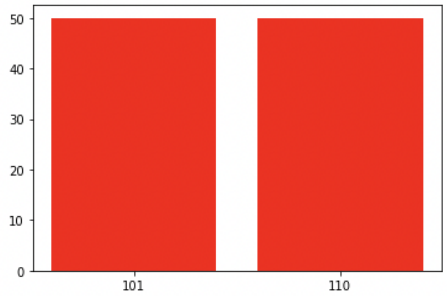
```
Circuit used to solve problem:
```




```
IBM Results:
{'000': 3, '001': 9, '010': 5, '011': 7, '100': 16, '101': 20, '110': 21, '111': 19}
```




```
Simulated results:
{'101': 50, '110': 50}
```



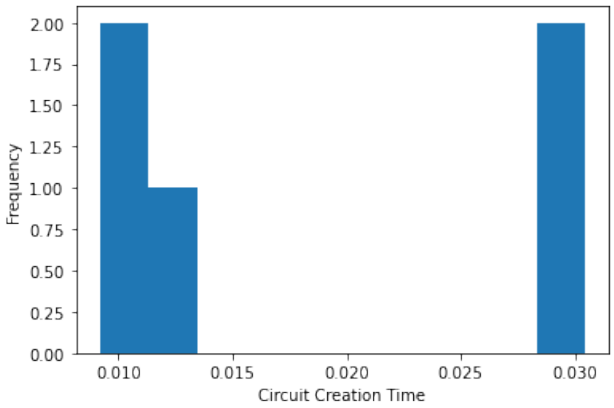

```
Expected Success Rate:  100.0 %
Time taken to create circuit:  0.001683950424194336
Time taken for simulation:  0.024838924407958984
Time taken by execution:  4.287785530090332
```

Figure 13: Grover's algorithm: Sample run for $n = 3$ and $a = 2$.

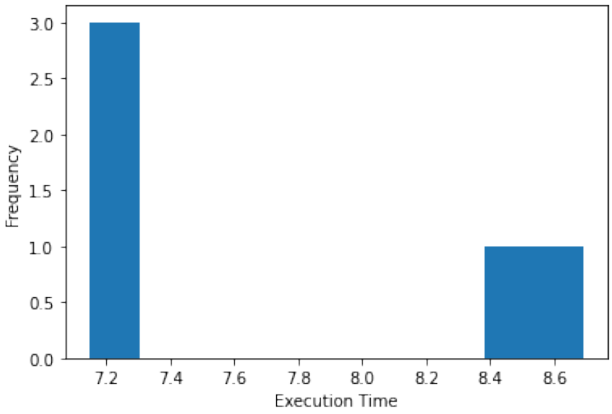
1.3.5 Dependence of execution time on U_f

In order to analyze the variance of the execution time with U_f, we implemented the following:

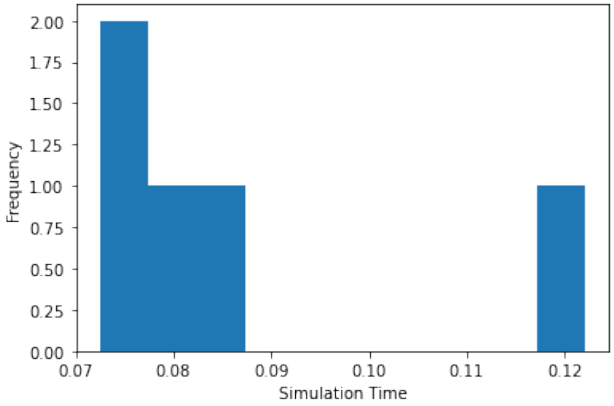
1. We ran the implementation for $n = 5$ and $a = 1$ five times, each time generating a random function. The fastest U_f circuit creation time was close to twice as fast as the slowest. We divided the time into three: time required to create the circuit, the simulation run time and the quantum computer execution time. (see 37a, 37b, 37c).



(a) Circuit Generation time for different U_f.



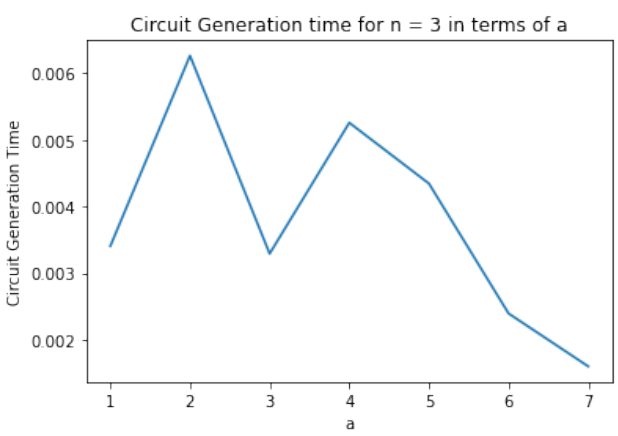
(b) Quantum Computer Execution time for different U_f.



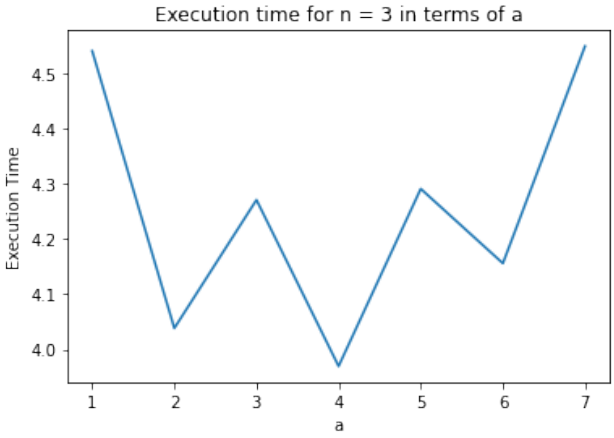
(c) Simulation time for different U_f.

Figure 14: Grover's algorithm: Effect of Uf on time for $n = 5$ and $a = 1$.

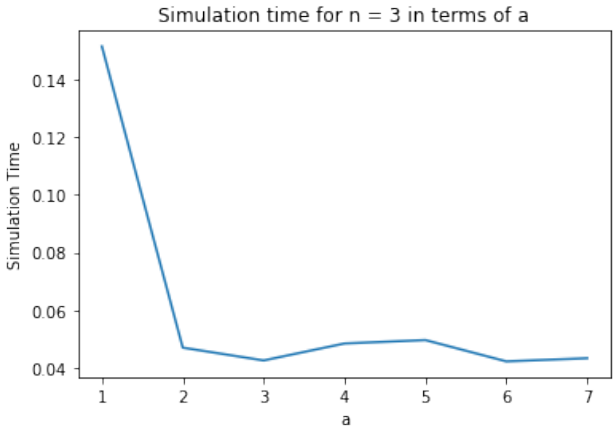
2. We expect a non-monotonic function when we try to analyze the impact of 'a' on execution time given a fixed n. The deviation of G from the identity matrix defines the iterations needed and this can vary for different 'a'. We might observe that for smaller 'a', a large number of iterations might be needed. Increasing 'a' a little bit might decrease this but increasing it more might overshoot and again require a large number of iterations. As expected we do not see a monotonic trend here. (see 15a, 15b, 15c). a = 1 alone has a higher value. This makes sense as a = 1 requires 2 iterations instead of 1.



(a) Circuit Generation time for different 'a'.



(b) Quantum Computer Execution time for different 'a'.

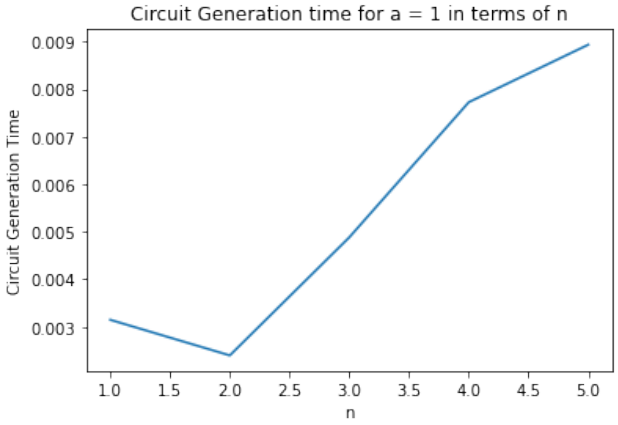


(c) Simulation time for different 'a'.

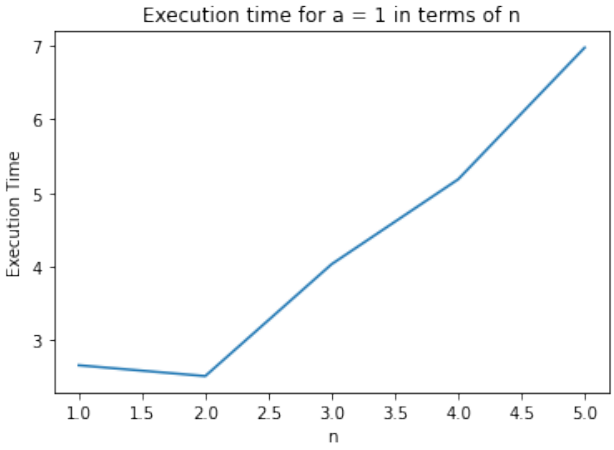
Figure 15: Grover's algorithm: Effect of 'a' on time for $n = 3$.

1.3.6 Scalability

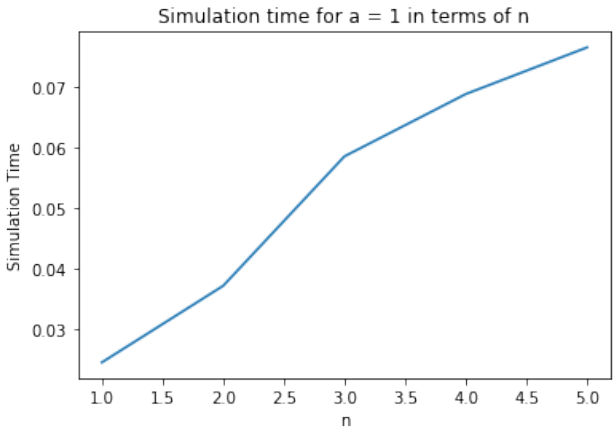
We benchmarked the circuit generation, simulation and execution times for every $n \in \{1, 2, 3, 4, 5\}$ and $a = 1$. We observed that the circuit generation, simulation and execution times for the quantum circuit grows exponentially with n. As can be seen in figures 16a and 16b the circuit generation and execution times both grow with increasing n and this is expected as with larger n we are searching in a larger space. When we increase n to 5 as shown in 16c, we can clearly notice the exponential growth. For a = 1, number of iterations are directly proportional to $\sqrt{N}$, which is $2^{n/2}$. Therefore, it makes sense that the time is increasing exponentially.



(a) Circuit Generation time for different 'n'.



(b) Quantum Computer Execution time for different 'n'.



(c) Simulation time for different 'n'.

Figure 16: Grover's algorithm: Effect of 'n' on time for $a = 1$.

1.3.7 Comprehensive evaluation/testing

We have tested the solution against expected probabilities for a number of different n and a values. We have provided clear histograms highlighting the majority element among multiple runs as well. In the evaluation front, we have provided explanations to cover the performance of the solution for various different values of n and a. We have plotted graphs to understand the impact of varying Zf's and also the effect of a and n on the final outputs.

1.3.8 More optimized circuit construction

We have gone over and above the matrix - custom circuit construction. We have used CZ gates to implement the various functions. We have enabled this circuit for any number of 'a' as well. Moreover, we have calculated the number of iterations accurately without using the a ¡¡ n assumption. Therefore the calculation extends well to all values of 'a' and 'n'.

1.3.9 Code well designed for improved usability or ease of understanding

The code is divided into clear functions. Apart from the basic features we have added the functionality to enable or disable verbosity. Moreover, we have added the feature to pass the n or get it from the user at the run time to enable better usability. Similarly, the value of 'a' could be provided while running or let the program choose randomly. To make the code easy to understand, clear and precise comments are added throughout the solution.

1.4 Deutsch-Josza algorithm

This algorithm was, in many ways, an onset to the computational efficiencies demonstrated by quantum computing over classical computing. We first go through the problem statement:

Given a function $f : \{0,1\}^n \rightarrow \{0,1\}$ *which is either balanced or constant, determine whether it is balanced or constant.*

The traditional classical version goes through at least $2^{n-1}+1$ calls to the function f. However, the Deutsch-Josza algorithm is equipped with finding an encoding of function f which it calls, as an oracle U_f, and defined as below:

$$U_f \left|x\right\rangle \otimes \left|b\right\rangle = \left|x\right\rangle \otimes \left|b \oplus f(x)\right\rangle, \tag{6}$$

It makes a single call to the black-box function, or the oracle as defined above, specific to the problem and with the ingenious circuit below, it is able to ascertain whether the function is balanced or constant. for all $x \in \{0,1\}^n$ and $b \in \{0,1\}$. The Deutsch-Jozsa circuit is shown in Fig. 17. If the measured state is $\left|0\right\rangle^{\otimes n}$, then the algorithm concludes that f is constant, and balanced otherwise.

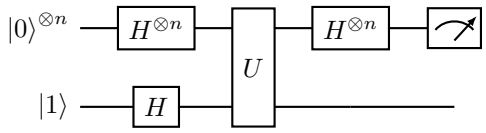


Figure 17: Deutsch-Jozsa algorithm

Now we discuss various aspects of our code design:

1.4.1 Implementation of U_f:

Before implementing the unitary matrix U_f, it is first necessary to obtain the matrix representation in the computational basis. To that end, we implement the following:

1. If it is a constant 0 function, the matrix is an identity matrix. b xor f(x) is just b. So the qubits remain unchanged.
2. If it is a constant 1 function, we need reverse the last bit for every x. So if x is even, we increase 1 and if odd we decrease 1. This way we can reverse the last bit.
3. If it is balanced. Then we choose half of the possible x values. Half the x have f(x) = 1 and hence have the 1's along the diagonals (unchanged like constant 0 case). For remaining half we follow the constant 1 logic of reversing the last bit.

Once the matrix is defined, we use `Operator` in Qiskit in order to get the Gate object.

1.4.2 Code readability:

We split our code into multiple small functions, each of which executes a simple task.
We will briefly describe the role of each function in our code below:

1. `createUfDJ(n, fntype, verbose)`: Creates the Uf matrix using the logic mentioned above.
2. `bitstring(bits)`: Converts bits of number to a string.
3. `runAndPrint(n, uf_gate, verbose)`: This creates the circuit and runs the code. As the circuit is common between DJ and BV, it is common for both the algorithms.

4. `runMainCircuitDJ(n,typeOfFn, verbose)`: This is the function to be called to run the code. Provide the value of n or get it from user at run time. typeOfFn tells if we want to generate uniform or constant function. Pass nothing for this to be randomly generated. We have added this feature so that we can compare execution times for various n value for same type of functions. verbose helps to set if we need to print all details or not.

By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code readable.

1.4.3 Parametrizing the solution in n:

The circuit is different for every value of n. We dynamically create the circuit corresponding to any given value of n, run the circuit, and interpret the result of the measurement to determine whether the function is balanced or constant.

1.4.4 Code testing

We first performed correctness check by running over various values. We tested for various values of n and various different types of Uf and it has always provided accurate results 100% of the time, as expected in simulation. In quantum computer, we observed lesser success due to noise. For any given function it returns all 0s if it is constant and anything other than that when it is uniform. A sample circuit and a sample histogram are shown below.

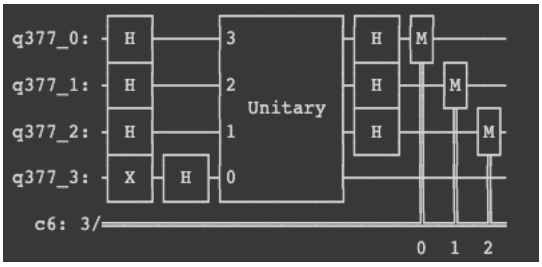


Figure 18: Sample Circuit

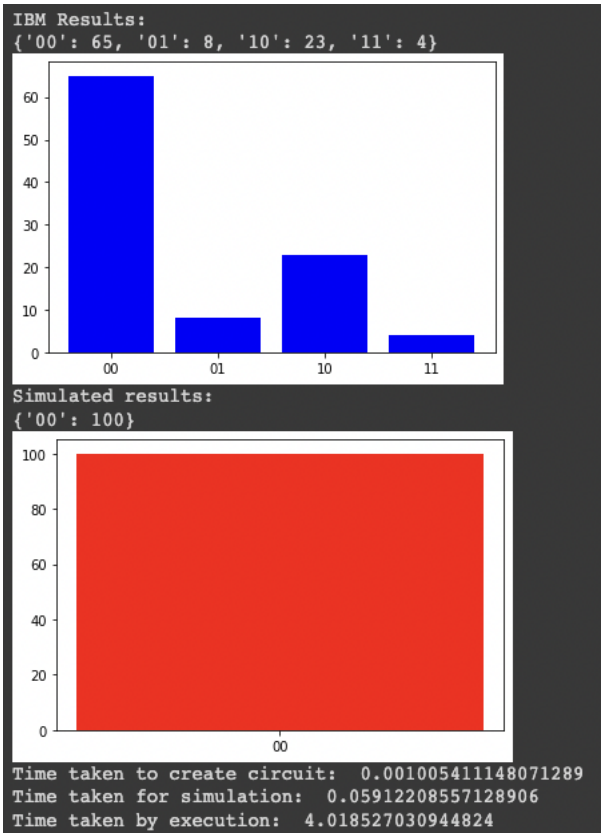
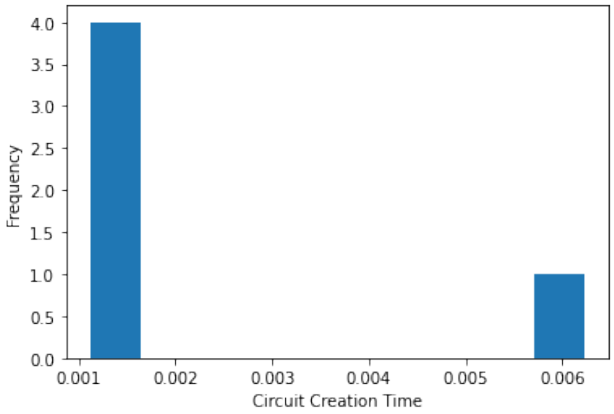


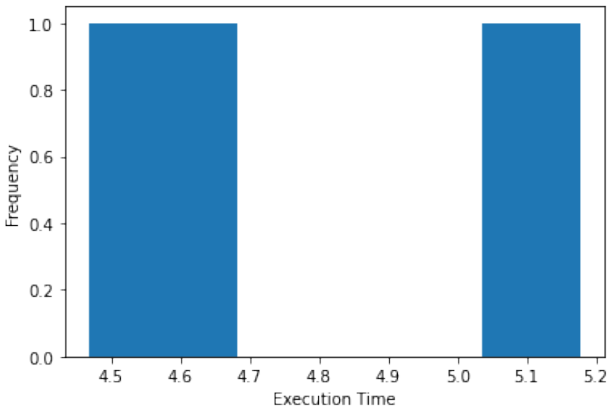
Figure 19: Sample Results.

1.4.5 Dependence of execution time on U_f:

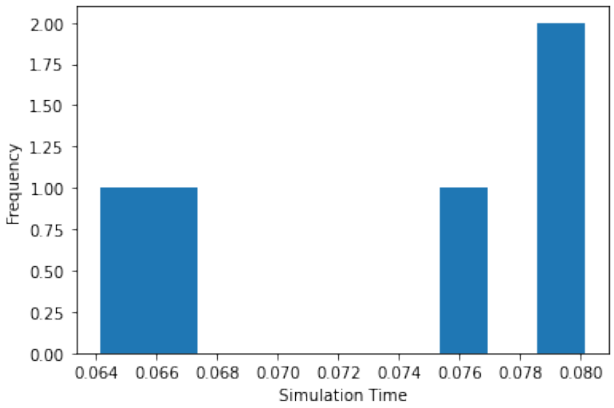
1. We ran the implementation for $n = 4$, 5 times, each time generating a random uniform function. We divided the time into two: time required to create the circuit and the execution run time. (see 20a, 20b, 20c). We do not see much range in the values. The range of times seem to be fairly constant.



(a) Circuit Generation time for different U_f when $n = 4$



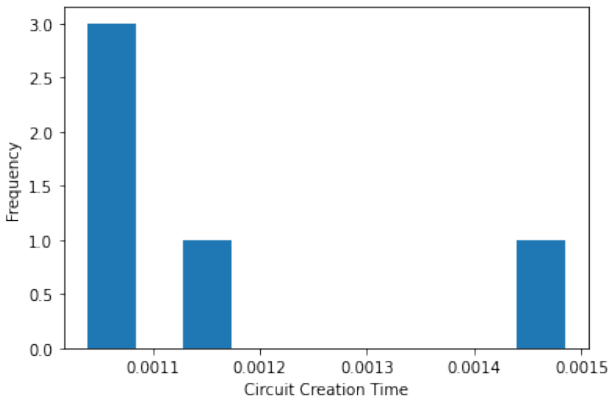
(b) Execution time for different U_f when $n = 4$



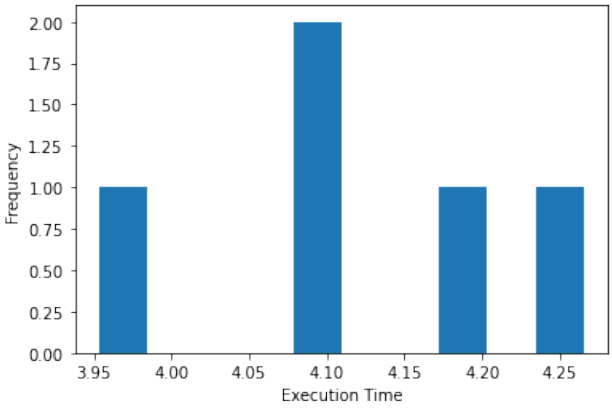
(c) Simulation time taken for different U_f when $n = 4$

Figure 20: DJ algorithm: Effect of Uf on time (Uniform case)

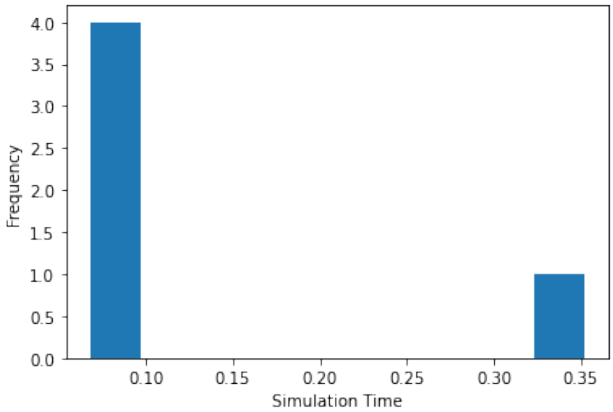
2. Specific to the constant function, we repeated the entire process of finding U_f for the two cases possible: all 0's or all 1's. Our observations are plotted and shown in the figures(see 21a, 21b, 21c). We see that in this case the times are much better than the uniform one.



(a) Circuit Generation Time range for n=4 for different Uf.



(b) Execution Time range for n=4 for different Uf.

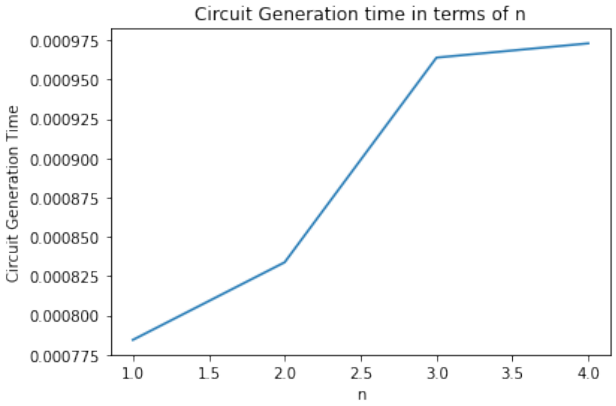


(c) Simulation Time range for n=4 for different Uf.

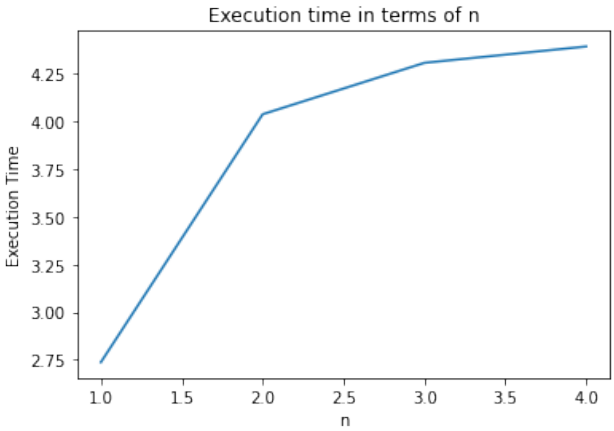
Figure 21: DJ algorithm: Effect of Uf on time

1.4.6 Scalability:

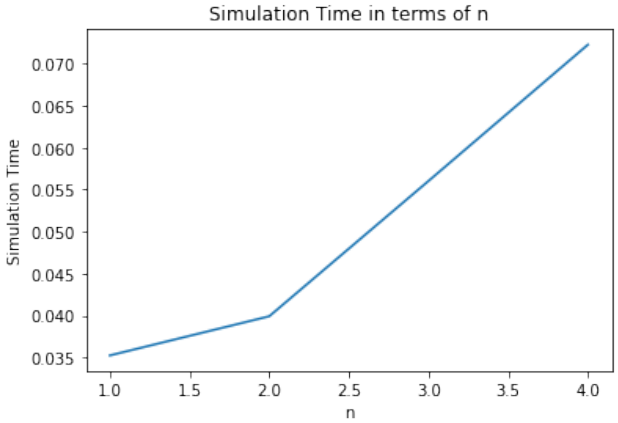
We observed that the execution time for the circuit simulation grows exponentially with the number of qubits involved, i.e. $n + 1$. For each $n \in \{1, 2, 3, 4\}$, we executed the Deutsch-Jozsa algorithm and obtained the average execution time for each value of n. As can be seen in figure 21, the execution time grows exponentially. The exponential growth of execution time with n is in line with the fact that the matrix sizes grow exponentially with the number of qubits.



(a) Circuit Generation time as a function of n.



(b) Execution time as a function of n.

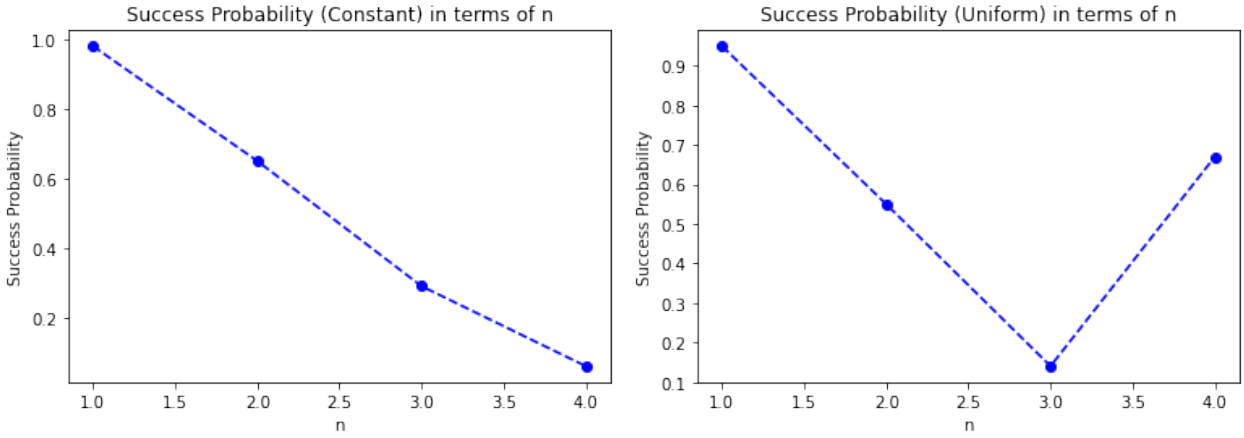


(c) Simulation time as a function of n.

Figure 22: DJ algorithm: Effect of n on time

1.4.7 Success Rate in Quantum Computer as a function of n:

We compared the success rates of the algorithm as a function of the 'n', which is a measure of the number of qubits. We tried for n values of 1,2,3 and 4 as only 5 qubits can be run in the given quantum computer. We have run for both Constant and Uniform functions. As expected, with more qubits and more gate operations, the errors are much higher and hence the success rate is lower. The exception to the trend was n = 4 for uniform case. The circuit generated had 10 out of 16 possible states in the output with varying probabilities. So even if the noise results in uniform distribution across all 16 probabilities, this will give high success rates. Another way of looking at it, is to consider all 0 cases as failures. In this case, we see a consistent failure rate, irrespective of n.



(a) Constant Functions.

(b) Uniform Functions.

Figure 23: Success Rate as a function of n

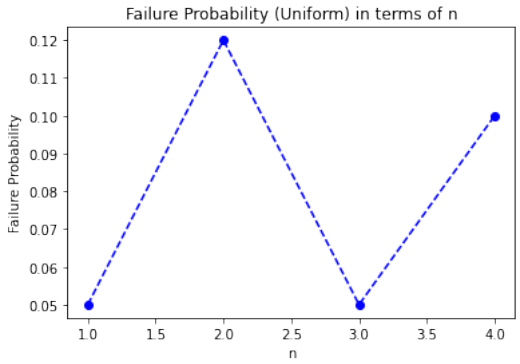


Figure 24: Failure Rate of Uniform Functions

1.4.8 Comprehensive evaluation/testing:

We have tested the solution against expected results for a number of different n, a and b values. In the evaluation front, we have provided explanations to cover the performance of the solution for various different values of n. We have plotted graphs to understand the impact of varying Uf's and also the effect of n on the final outputs.

1.4.9 More optimized circuit construction:

We have gone over and above just implementing a generic matrix. We have made use of particular properties to make the implementation faster for particular functions. Like Uniform functions can be implemented faster.

1.4.10 Code well designed for improved usability or ease of understanding:

The code is divided into clear functions. Apart from the basic features we have added the functionality to enable or disable verbosity. Moreover, we have added the feature to pass the n or get it from the user at the run time to enable better usability. To make the code easy to understand, clear and precise comments are added throughout the solution. We have also added support to pass the type of function to be generated, so that we can compare execution times over various values of n.

1.5 Bernstein-Vazirani algorithm

First, we summarize the problem statement:

Given a function $f : \{0,1\}^n \rightarrow \{0,1\}$ *of the form* $f(x) = a.x + b$*, with* $a \in \{0,1\}^n$ *and* $b \in \{0,1\}$ *are unknown bit strings, obtain the value of* a*.*

While a classical algorithm requires $\mathcal{O}(n)$ calls to the function f, the Bernstein-Vazirani (BV) algorithm obtains the bit string a with just one call to the oracle U_f, which satisfies equation (6). The quantum circuit for the BV algorithm is the same as that of the DJ algorithm (Fig. 17). The BV algorithm determines a to be the state of the n qubits measured at the end of the circuit.

Now we discuss various aspects of our code design

1.5.1 Implementation of U_f

We implement Uf using this simple logic. We find f(x) for all values of x for given a and b. If this value is 0, we know that (b xor f(x)) would remain b and hence qubits remain unchanged. If not, then we need to reverse the last qubit. Reversing last bit is same as reducing by one for odd and increasing by one for even numbers.

1.5.2 Code readability:

We split our code into multiple small functions, each of which executes a simple task. We use `Oracle`, `bitstring`, `runAndPrint` functions from DJ without repeating. The only new functions used are:

1. `runMainCircuitBV(n, verbose)` for higher level access to the implementation of the BV algorithm call this function and pass the n value required. It creates the circuit and the Uf and then calls runAndPrint function to get the output. It finds the value of b by checking f(0*n). This returns b value. Then we run the quantum program using `runAndPrint` to generate the value of 'a'.
2. `decimalToBinary(x,n)`: This converts a decimal value to binary.
3. `getFx(x, a, b, n)`: This generates f(x) = x.a xor b value and returns it.
4. `createUfBV(n, a, b, verbose)`: This creates the Uf matrix using the logic stated above.

By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code readable.

1.5.3 Parametrizing the solution in n

The circuit is different for every value of n. We dynamically create the circuit corresponding to any given value of n, run the circuit, and interpret the result of the measurement to determine whether the right 'a' was found.

1.5.4 Code testing

We first performed correctness check by running over various values. We tested for various values of n and various different types of Uf and it has always provided accurate results 100% of the time, as expected in simulation. In quantum computer, we observed lesser success due to noise. For any given function it returned the exact value of 'b' through classical means and then returns 'a' through quantum means. A sample circuit and a sample histogram are shown below.

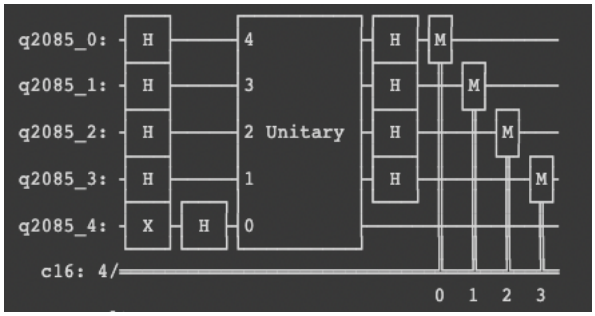


Figure 25: Sample Circuit

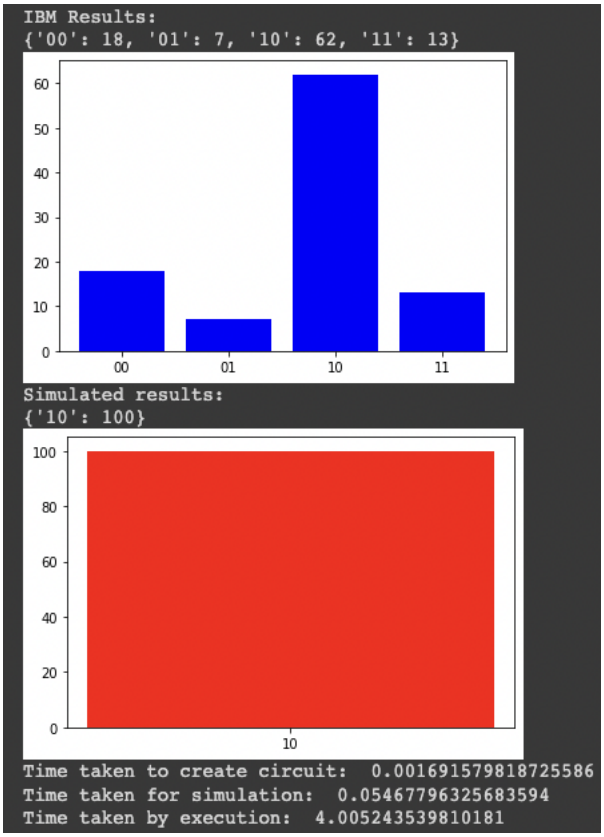


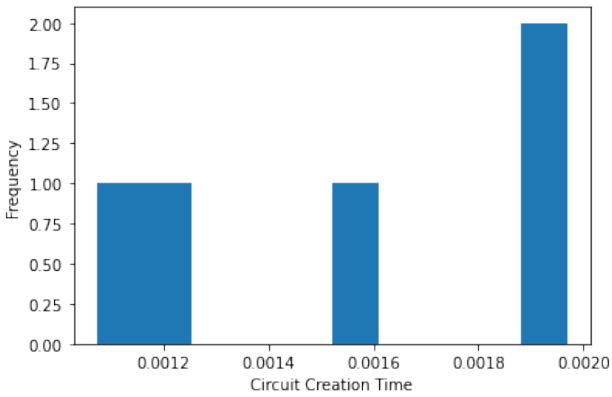
Figure 26: Sample Results.

1.5.5 Similarities in our codes for the BV and DJ implementations

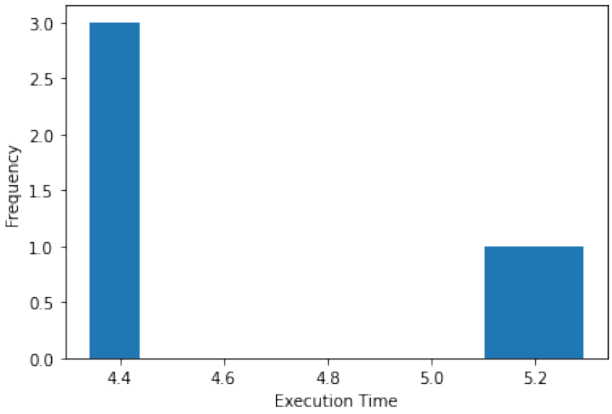
We estimate about 60% code similarity between our BV and DJ algorithms. In particular, the functions verb—Oracle—, `bitstring` and `runAndPrint` are identically defined in both the implementations. Moreover, the only difference between the `BV` and `DJ` functions is in the interpretation of the outputs, with the `DJ` function returning 0 or 1, and the `BV` function returning a bit string of length n. The similarity in the codes is due to the fact that the circuits in both these algorithms are identical.

1.5.6 Dependence of execution time on U_f

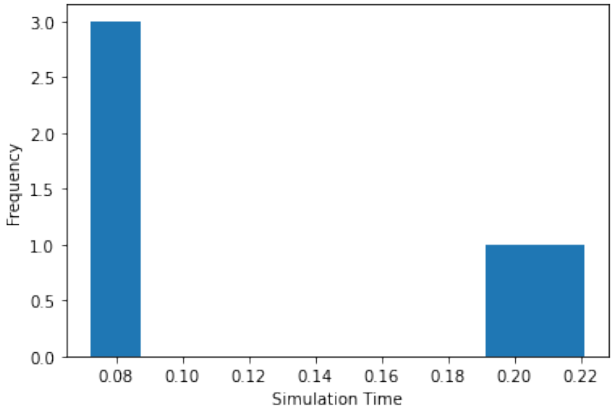
Similar to the DJ algorithm, we observe a variance in the execution time with U_f. For $n = 4$, we simulated 5 randomly chosen U_f's, and plotted the histogram in figure 27. Clearly, there is a higher variance in the simulation times but the quantum computer execution time is fairly consistent. Circuit Generation time is very similar as expected.



(a) Circuit Generation Time range for n=4 for different Uf.



(b) Execution Time range for n=4 for different Uf.

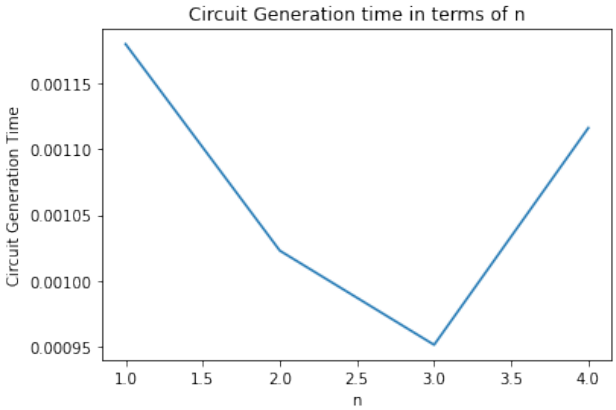


(c) Simulation Time range for n=4 for different Uf.

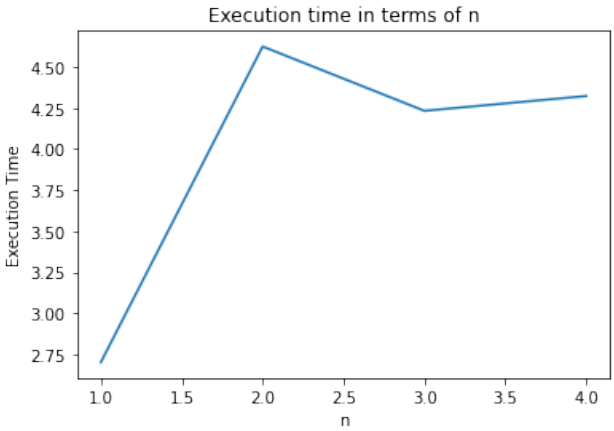
Figure 27: Bernstein-Vazirani algorithm: Effect of Uf on time

1.5.7 Scalability

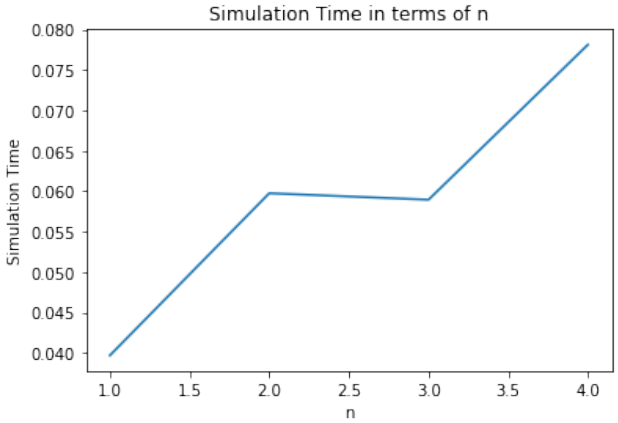
Similar to the DJ algorithm, we observed an exponential increase in the simulation time with n. Let us observe the trend for execution time. Due to the quantum computer qubit limitation, we run for each $n \in \{1, 2, 3, 4\}$ and obtained the execution time. As can be seen in figure 28, the execution time grows with increasing n. We do observe that at a few places, there is not much increase. As we are running for lesser counts, the type of Uf generated also play a part here and this explains the trend.



(a) Circuit Generation Time for different n.



(b) Execution Time range for different n.



(c) Simulation Time range for different n.

Figure 28: Bernstein-Vazirani algorithm: Effect of n on time

1.5.8 Success Rate in Quantum Computer as a function of n:

We compared the success rates of the algorithm as a function of the 'n', which is a measure of the number of qubits. We tried for n values of 1,2,3 and 4 as only 5 qubits can be run in the given quantum computer. As expected, with more qubits and more gate operations, the errors are much higher and hence the success rate is lower. It falls down from over 90% to a mere 4% in the gap of 3 extra qubits.

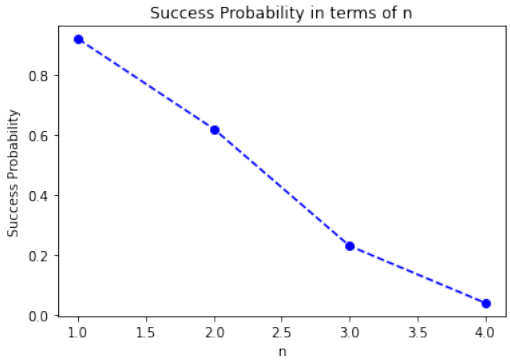


Figure 29: Success Rate as a function of n

1.5.9 Comprehensive evaluation/testing

We have tested the solution against expected results for a number of different n, a and b values. In the evaluation front, we have provided explanations to cover the performance of the solution for various different values of n. We have plotted graphs to understand the impact of varying Uf's and also the effect of n on the final outputs.

1.5.10 More optimized circuit construction

We have gone over and above just implementing a generic matrix. We have made use of particular properties to make the implementation faster for particular functions.

1.5.11 Code well designed for improved usability or ease of understanding

The code is divided into clear functions. Apart from the basic features we have added the functionality to enable or disable verbosity. Moreover, we have added the feature to pass the n or get it from the user at the run time to enable better usability. To make the code easy to understand, clear and precise comments are added throughout the solution.

1.6 Shor's algorithm

1.6.1 Low Level Design / Code Walkthrough:

Given a number n, we return the prime factorization of n. Shor's algorithm factors integers in polynomial time with high probability. The idea of Shor's algorithm is to guess a random integer that is less than the input and then check whether the guess leads to a factoring. The guess-and-check may well fail but it will succeed with probability greater than 1/2. So if we repeat the guess-and-check a few times, we are highly likely to succeed. The basic idea is given by:

```
Algorithm: Run Main Circuit
Input: A positive integer N >= 2.
Output: A prime factorization N = p1 ** k1 * ...  pk ** k_m.
Method:
while(1):
    (factor, power) = find_factor(N)

    if factor is None:
        break

    factors_with_power.append((factor, power))

    if (factor ** power) == N:
        answer_found = True
        break

    N = N // (factor ** power)

return factors_with_power
```

The process can be summarized as:

- First find a non-trivial factor of N. This code is explained below. Assume we do get back a non-trivial factor and the power of this factor in the prime factorization. If this return None, it means our algorithm failed.
- Add the factor to the prime factorization.
- If the value is this prime's power, then we have completed the prime factorization. Else continue factorizing $(N/factor^{power})$.

As can be seen, we have to find a non-trivial factor of N. Let us find how this is done:

```
Algorithm: Integer Factorization (find_factor).
Input: A positive integer N >= 2.
Output: A non-trivial factor of N, with its power -> (factor, power)
Method:
if (N is a prime) {
    return (N, 1)
}

if (N is even) {
    return (2, num_occurences(2))
}
else if (N = p ** k where p is prime and k >= 1) {
    return (p, k)
}
else {
    int d = Shor(N)
    return (d, num_occurences(d))
}
```

The process can be summarized as:

- First check if the number is prime, if yes just return it.
- First check if the number is even, then we know 2 is a non-trivial factor.
- Check if the given number is a power of prime, then we can solve it directly.
- If both the above cases do not satisfy, we try to find a non-trivial factor using Shor's Algorithm.
- We use a function $num\_occurrences(x)$ to find the power of x in the prime factorization of N.

Now let us look at the Shor's algorithm part of the above factorization code:

```
Algorithm: Shor's Algorithm.
Input: An odd, composite integer N that is not the power of a prime.
Output: A nontrivial factor of N, that is, an integer d such that 1 < d < N and d divides N.
Method:
repeat
    int a = random choice from {2, ..., N-1}
    int d = gcd(a, N)
    if (d > 1) {
        return d
    }
    else {
        int r = Find_Order(a, N)
        if (r is even) {
            int x = a ** (r // 2) - 1 (mod N)
            int d = gcd(x,N)
            if (d > 1) {
                return d
            }
        }
```

```
    }
until we give up
```

The process can be summarized as:

- First we take a random value from 2 to N-1 and if this happens to be a factor, then we are lucky and we can return it.
- Shor's algorithm calls the subroutine $Find\_Order$. The idea is that $Find\_Order(a, N)$ returns the smallest integer $r > 0$ such that a^r is 1(mod N). This number r is known as the order of a in Z_N^*.
- As n does not divide x $(x = a^{r/2} - 1(modN))$, we can hope that there might be some common factor between the two values that is non-trivial. So if the gcd between x and N is greater than 1, then we have found a non-trivial factor.

Now let us try to understand how $Find\_Order$ is implemented:

```
Algorithm: Order Finding.
Input: An integer N > 1 and an element x which belongs to Z*N.
Output: The smallest integer r > 0 such that a ** r = 1 (mod N).

Method:
if a < 2 or N <= a or math.gcd(a, N) > 1:
    raise ValueError()

circuit = create_orderfind_circuit(a, N)
backend = Aer.get_backend('aer_simulator')
circ = transpile(circuit, backend)
qobj = assemble(circ)
measurement = backend.run(qobj, shots = 1).result()
frequencies = measurement.get_counts()
return continued_fraction_algorithm(frequencies, a, N)
```

The process can be summarized as:

- First ensure that x is an element of the multiplicative group modulo N.
- Now create the order finding circuit. This function computes an eigenvalue of the unitary function using Phase Estimation, which internally using Quantum Fourier Transform.
- Sample from the created circuit.
- Each run produces k/r, where we don't know either of k and r. This is where the Continued Fraction Algorithm comes in. The Continued Fraction Algorithm maps a list of samples of k/r to r. The Continued Fraction Algorithm runs in $\mathcal{O}((logN)^3)$ time, which is a major contributor to the time complexity of Shor's algorithm.

1.6.2 Code readability:

The whole code was divided into multiple functions. Have used the parameter of `verbose`, which prints all the intermediate steps of the program if set to True. The following are the main functions and their purpose:

- `power_mod_N(n, N, a)`: This function defines modular exponential operation used in Shor's algorithm. It represents the unitary which multiplies the target with the base raised to exponent and applies modulo operator over it. More precisely, it represents the unitary which computes:

```
ME|target>|exponent> = |target * (base ** exponent MOD modulus)> |exponent>
else
ME|target>|exponent> = |target> |exponent>
```

- `create_orderfind_circuit(a, N, verbose)`: This function returns quantum circuit which computes the order, r which is that smallest integer such that a^r mod N = 1. This function computes an eigenvalue of the unitary using Phase Estimation, which internally using Inverse Quantum Fourier Transform.

```
U|y> = |xy mod n> if  0 <= y < n
U|y> = |y> if n <= y
```

- `continued_fraction_algorithm(frequencies, a, N)`: The function interprets the output of the order finding circuit. Specifically, it determines r from the k/r fractiosn that are passed.
- `find_order(a, N, verbose)`: This function finds the smallest positive r such that $a^r mod N == 1$.
- `find_factor_of_prime_power(N)`: This function returns non-trivial factor of n if n is a prime power, else returns None.
- `num_occurrences(N, f)`: Returns the number of occurrences of the factor f of N, that is the power of the factor, f in the prime factorization of N.
- `find_factor(N, verbose, max_attempts)`: This function calls all the previous functions to find a non-trivial factor of N and returns this factor with its occurrence count.
- `find_angles(a, n)`: This function calculates the array of angles to be used in the addition in Fourier Space.
- `double_controlled_phi_add_mod_N(angles, c_phi_add_N, iphi_add_N, qft, iqft)`: This function creates a circuit which implements double-controlled modular addition by a.
- `controlled_multiple_mod_N(n, N, a, c_phi_add_N, iphi_add_N, qft, iqft)`: This function implements modular multiplication by a as an instruction.
- `phi_add_gate(angles)`: This function creates the gate that performs addition by a in Fourier Space.
- `runMainCircuit(N, verbose, max_attempts)`: This is the main function that runs the entire factorization algorithm. It repeatedly calls the $find\_factor$ function to get one factor at a time and creates the prime factorization of N.

1.6.3 Parametrizing the solution in n

We create the circuit using $create\_order find\_circuit$ which can take any given value of N as input. The number of bits in the representation of N can be assumed to be n. Then we need 'n' target qubits, '2 * len' exponent qubits and 'n + 2' auxilliary qubits. Modular Exponentiation function is also parameterized in n and all the other gates are applied to the qubits and a circuit is generated dynamically.

1.6.4 Testing

The testing was done rigorously on the correctness of the solution by trying various values of N. The generated circuits were also printed. We tried running up to N of 30. For larger values of N, the memory and time requirement were too large to run multiple times and analyze even on a simulator.

1.6.5 Sample Circuit Generated for Find Order:

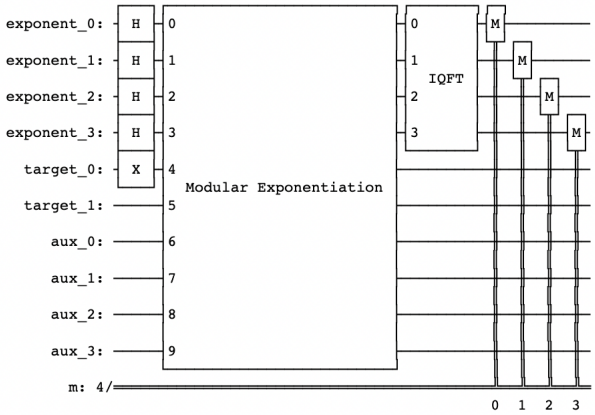


Figure 30: Find Order Circuit

1.6.6 Testing on Qiskit Simulator for correctness:

End to end testing of the Shor's algorithm was initially done on Qiskit Simulator. The tests were done for multiple N including 10, 15 and 21. For larger than 30, the time and memory requirements were too high for the tests to be feasible.

```
N = 21
Factors:
[(3, 1), (7, 1)]
Time taken:  33.851261377334595
```

Then, only the quantum part of the circuit was tested. The find order function was tested for various different values of 'a' and N.

```
a = 7
N = 15
Simulation r:  4
```

1.6.7 Testing on IBM Quantum Computer and Advanced Simulators:

We tried to test the solution on multiple IBM Quantum backends like ibmq_quito and even some noisy backend simulators like simulator_mps, simulator_statevector and ibmq_qasm_simulator. But all these tests failed due to one of these 2 reasons:
1. The backend supports only 5 qubits.
2. The job was too long.

1.6.8 Testing on Qiskit Simulator with custom Noise Model:

Here again, we tried to extract the noise model from an actual backend machine and apply it over the aer_simulator but the noise models were tightly coupled to the number of qubits and hence the 5 qubit limitation carried over.

We therefore ran the code on custom Noise Mode. We applied all qubit depolarizing quantum errors with varying 1-qubit and 2-qubit error probabilties.

```
NoiseModel:
  Basis gates: ['cx', 'id', 'rz', 'sx', 'u1', 'u2', 'u3']
  Instructions with noise: ['u1', 'u2', 'u3', 'cx']
  All-qubits errors: ['u1', 'u2', 'u3', 'cx']
```

We tried to compare the phase estimation and the corresponding 'r' results based on the outputs across 1024 runs. The input values were: a = 2 and N = 3.

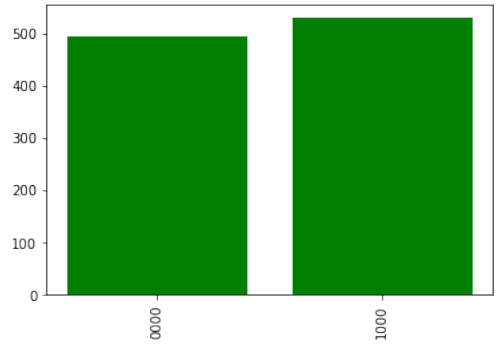


Figure 31: Without Noise: Output Histogram

	Register Output		Phase Fraction	Guess for r	Count
0	0000(bin) =	0(dec)	0/16 = 0.00	0/1	1	481
1	1000(bin) =	8(dec)	8/16 = 0.50	1/2	2	543

Success Probability 0.5302734375

Figure 32: Without Noise: Phase Estimation and corresponding 'r' values.

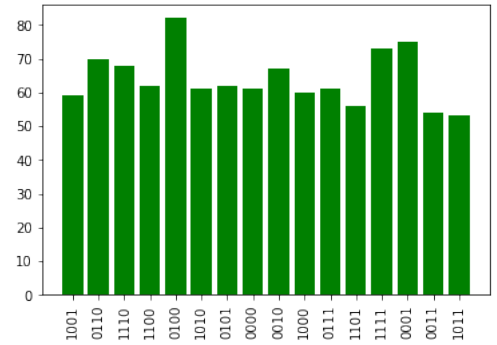


Figure 33: With Noise: Output Histogram

	Register Output		Phase Fraction	Guess for r	Count
0	0000(bin) =	0(dec)	0/16 = 0.00	0/1	1	53
1	0110(bin) =	6(dec)	6/16 = 0.38	1/3	3	64
2	1110(bin) =	14(dec)	14/16 = 0.88	1/1	1	78
3	1011(bin) =	11(dec)	11/16 = 0.69	2/3	3	77
4	0111(bin) =	7(dec)	7/16 = 0.44	1/2	2	58
5	1010(bin) =	10(dec)	10/16 = 0.62	2/3	3	70
6	0010(bin) =	2(dec)	2/16 = 0.12	0/1	1	73
7	0101(bin) =	5(dec)	5/16 = 0.31	1/3	3	47
8	0011(bin) =	3(dec)	3/16 = 0.19	1/3	3	73
9	0001(bin) =	1(dec)	1/16 = 0.06	0/1	1	70
10	0100(bin) =	4(dec)	4/16 = 0.25	1/3	3	67
11	1100(bin) =	12(dec)	12/16 = 0.75	2/3	3	47
12	1000(bin) =	8(dec)	8/16 = 0.50	1/2	2	61
13	1111(bin) =	15(dec)	15/16 = 0.94	1/1	1	64
14	1101(bin) =	13(dec)	13/16 = 0.81	2/3	3	65
15	1001(bin) =	9(dec)	9/16 = 0.56	1/2	2	57

Success Probability 0.171875

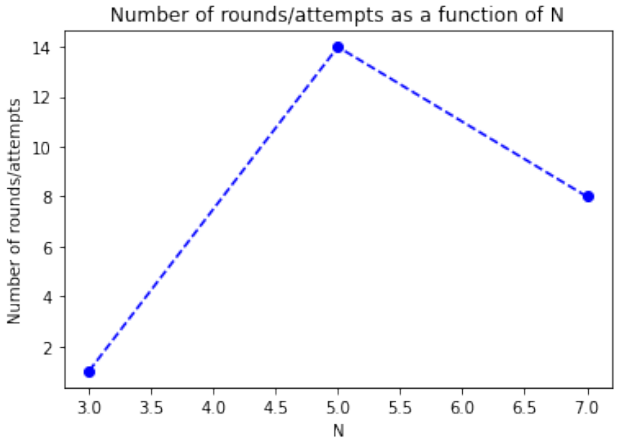
Figure 34: With Noise: Phase Estimation and corresponding 'r' values.

As we can observe, the success probability drops from 53% to a mere 17% on applying noise. This noise simulation is an approximate measure of how an actual Quantum computer works and gives us an idea why noise is a major concern in the field. For a = 3 and N = 5, success rate falls from 48% to 12%.

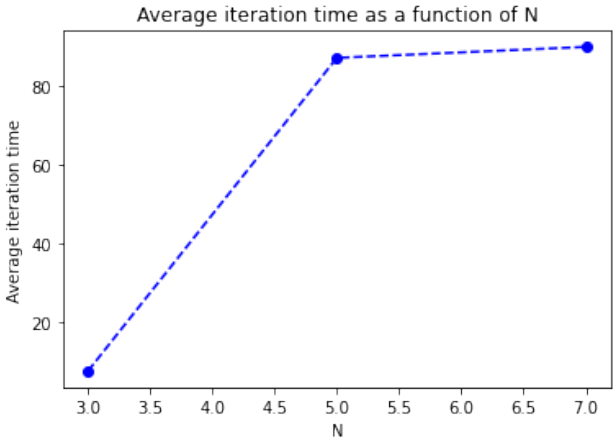
1.6.9 Variation of Execution Time with number of qubits:

As we know that the number of qubits is given by 3*n + 3, where $n = \log N$. Therefore N is an estimate of number of qubits. We have fixed 'a' = 2 and calculated the following parameters for different N values of 3, 5 and 7 using the noisy model:

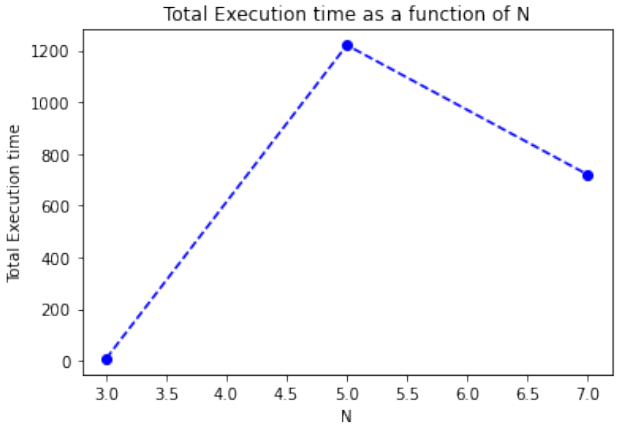
- Number of rounds/attempts: This measures the number of times the circuit was executed. This is a measure of how accurate the random guess is accurate and also on how much reliable the circuit is.
- Average Iteration Time: This measures the average time taken to run the circuit. This helps to understand how the time scales for increasing N.
- Total Execution Time: This is just the product of the previous two parameters.



(a) Number of attempts as a function of N.



(b) Average Iteration Time as a function of N.



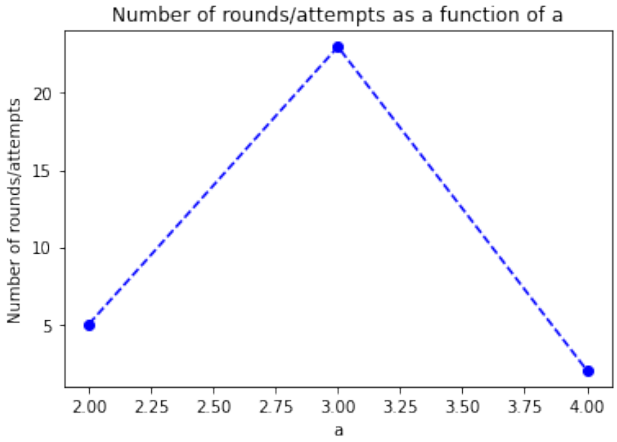
(c) Total Execution time as a function of N.

Figure 35: Shor's algorithm: Effect of N on execution times for $a = 2$.

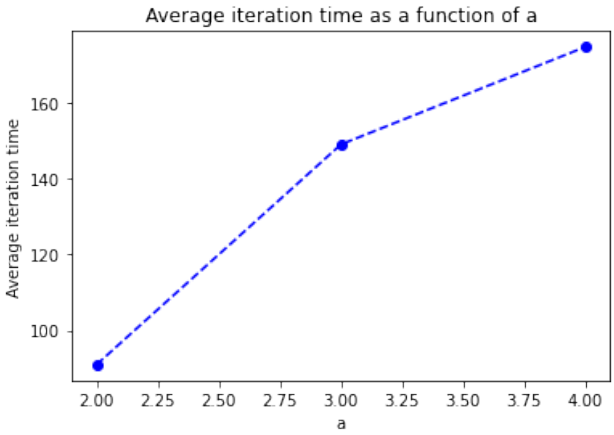
- We observe that the average execution time of a circuit clearly increases with N. This is because larger the N, more the qubits and longer it takes to run the circuit.
- The same is not true for number of attempts and in turn total time as this could be related to the random selection of number and its failure as well.

1.6.10 Variation of Execution Time with number of 'a':

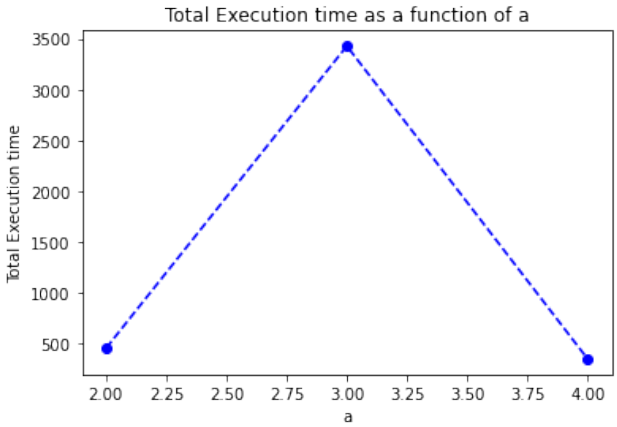
We have fixed 'N' = 5 and calculated the same parameters for different a values of 2, 3 and 4 using the noisy model.



(a) Number of attempts as a function of 'a'.



(b) Average Iteration Time as a function of 'a'.



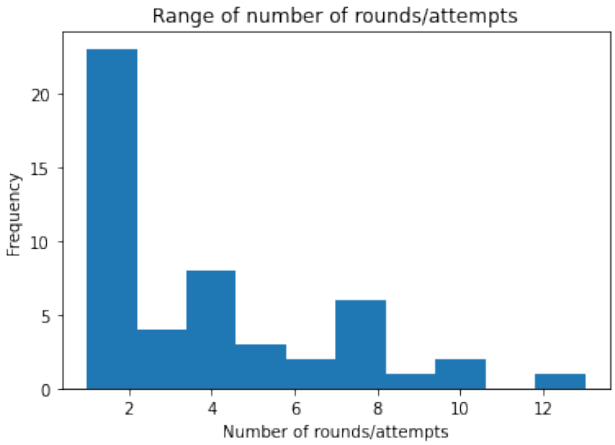
(c) Total Execution time as a function of 'a'.

Figure 36: Shor's algorithm: Effect of 'a' on execution times for $N = 5$.

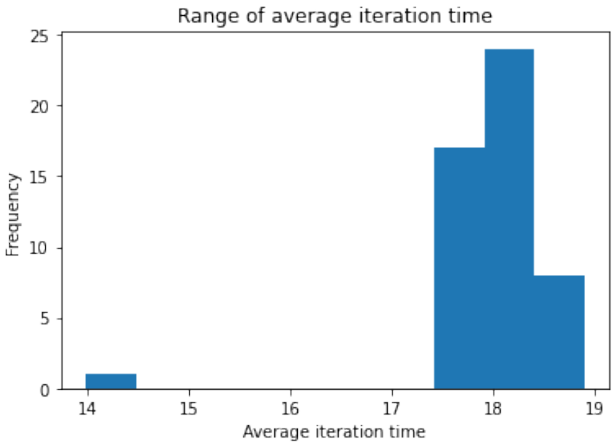
- We observe that the average execution time of a circuit clearly increases with a.
- The same is not true for number of attempts and in turn total time as this could be related to the random selection of number and its failure as well.

1.6.11 Range of time taken for different runs of the same N and a:

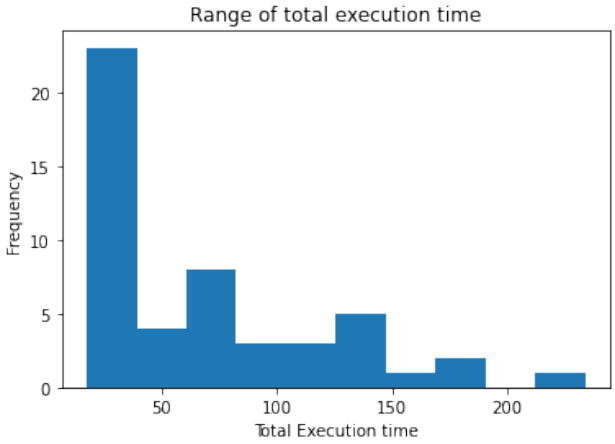
In this experiment, we see the histograms for different runs for the same value of N = 3 and a = 2. We ran the code 25 times using the noisy model.



(a) Range of number of attempts.



(b) Range of Average Iteration Time.



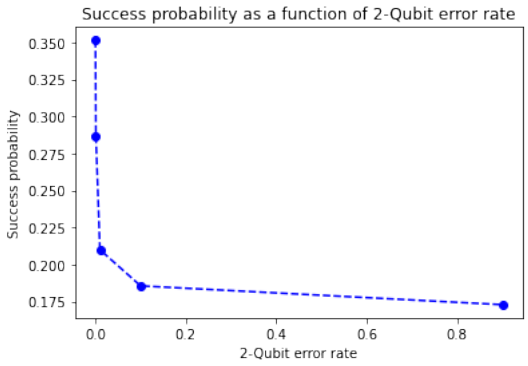
(c) Range of Total Execution time as a function of N.

Figure 37: Shor's algorithm: Range of execution times for $a = 2$ and $N = 3$.

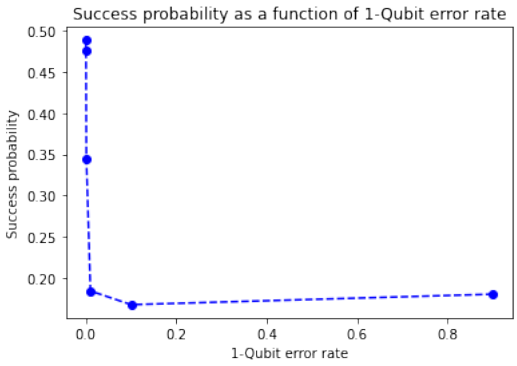
- First we observe that there is a huge range of number of attempts. This has a lot to do the measurements as well and hence could be randomized.
- The average iteration execution time seems to be fairly consistent. This is expected as the number of qubits and number of operations are exactly the same.
- The total execution time is hugely dependent on the number of rounds as the average execution time is fairly constant.

1.6.12 Success Rate as a function of Qubit Error Rate:

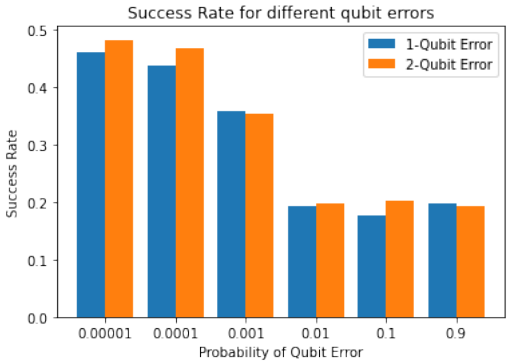
In this experiment, we vary the 2-qubit error rate for a given 1-qubit error rate of 0.001 and a = 2 and N = 3 in the custom noise model. The we vary the 1-qubit error rate for a given 2-qubit error rate of 0.0001.



(a) Success Rate as a function of the 2-Qubit Error Rate.



(b) Success Rate as a function of the 1-Qubit Error Rate.



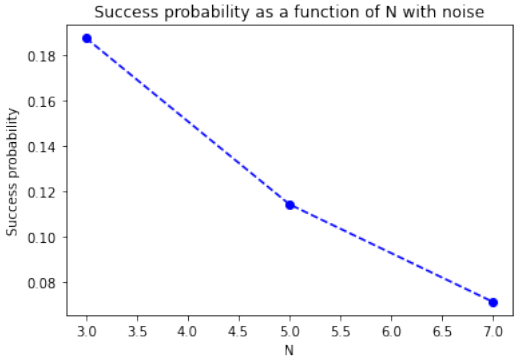
(c) Success Rate 2-Qubit vs 1-Qubit errors.

Figure 38: Shor's algorithm: Success Rate as a function of Qubit error rate.

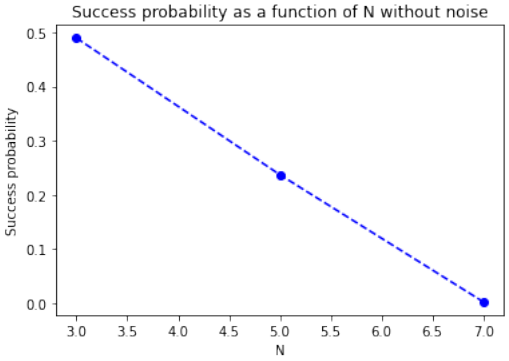
- First, as expected we observe that the success rate exponentially decreases as we increase the 2-Qubit and 1-Qubit error rate.
- We observe that the type of error doesn't affect the success rate much. It is just the probability of error that affects the success rate.

1.6.13 Success Rate as a function of N:

In this experiment, wanted to check the success rates of the Algorithm by varying N. We set a = 2 and ran the code for both without noise and also by using the same noise model as above.



(a) Noisy Simulation: Success Rate as a function of N.



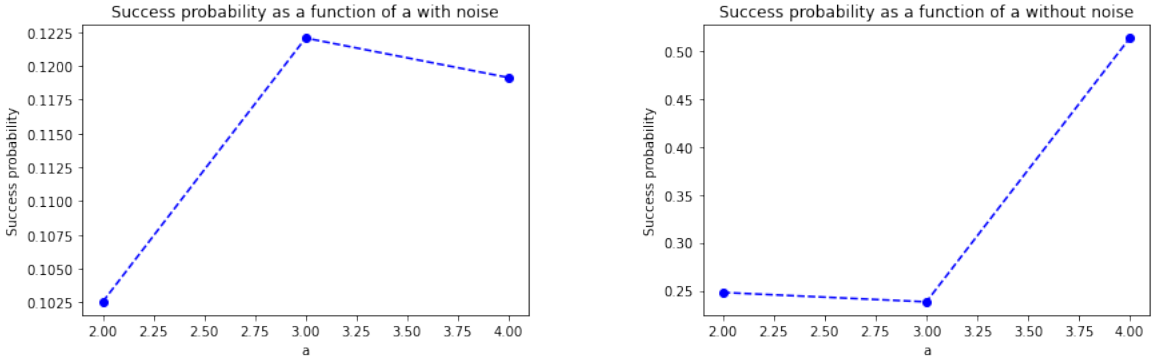
(b) No noise: Success Rate as a function of N.

Figure 39: Shor's algorithm: Success Rate as a function of N for a = 2.

- First, in both cases the success rates decrease with increasing N.
- Interesting observation is that though the success rates for N = 3 and 5 are lesser for noisy simulation, the value for N = 7 is fairly same. This is because in case of without noise simulation also, this accuracy was fairly low.

1.6.14 Success Rate as a function of 'a':

In this experiment, wanted to check the success rates of the Algorithm by varying 'a'. We set N = 5 and ran the code for both without noise and also by using the same noise model as above.



(a) Noisy Simulation: Success Rate as a function of 'a'.

(b) No noise: Success Rate as a function of 'a'.

Figure 40: Shor's algorithm: Success Rate as a function of 'a' for N = 5.

- In this case, we do not observe much of a pattern. We can conclude that there is not much empirical dependency of the success rate on 'a'.

1.7 QAOA Algorithm

1.7.1 Low Level Design / Code Walkthrough:

The objective of a Quantum Approximate Optimization Algorithm (QAOA) is to find approximate solution to a combinatorial optimization algorithm, with n variables and m clauses. [1]

More specifically,
Given fixed n, m, we take sufficiently varied number of $\gamma \in [0, 2\pi]$, $\beta \in [0, \pi]$, *such that when we measure:*
Mix(β*) Sep(*γ*)* $H^{\otimes n}$ $|0^n\rangle$ *we obtain a measurement z that maximizes count(z)*

Constraints:

- We have restricted to Max2Sat problem that comprises of all m clauses with 2 literals, each such literal can be any one of the n variables. Each clause can have repetition or negation of the same literals.
- **Problem statement**: Here n denotes the number of variables, m denotes the number of clauses and t which can be anything from 1,2,....m denotes the number provided by the user of how many clauses the algorithm must satisfy. The user asks if it does satisfy at least t clauses, then output 1, else output 0.
- Here, for current implementaion, p = 1, which denotes application of *Mix(*β*) Sep(*γ*)* $H^{\otimes n}$ in the above statement for 1 time. Theoretically, p can be varied to be sufficiently large to approximate the solution with higher probability.

The key to obtain the set of best solutions — by performing grid search of γ's and β's, is that for each such pair, QAOA satisfies as close to maximum clauses as possible — is dependent on designing apt implementation of the circuit.

Given a circuit with n qubits and 1 helper qubit, where each qubit denotes one variable, we divide our objectives in the following stages: For all different γ's and β's:

- i. Estabilishing Ground Truth using Classical Solver
- ii. Devising and Implementing Separator Circuit
- iii. Desigining Mixer circuit
- iv. Measure z and Evaluate Count(z). If $Count(z) \geq t$, then it's a good solution.

All the information can be summarized in the figure 41:

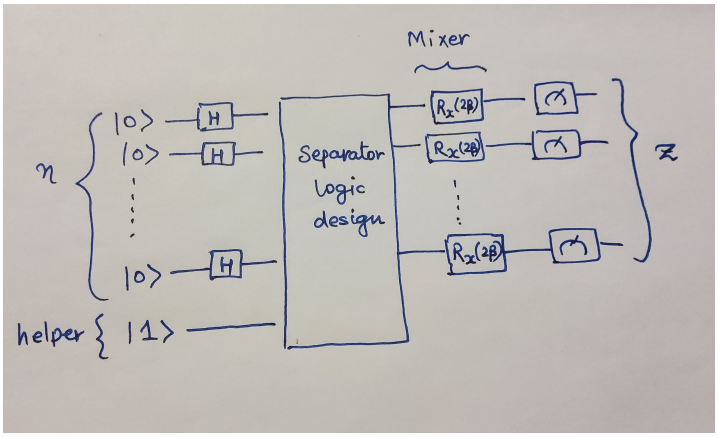


Figure 41: Overall Circuit logic

Each such objective is achieved using the following process:
i. Estabilishing Ground Truth

- **Black box function dependent on n and m:** We obtain a random list of m entries, each defining one clause. Further, each such clause is composed of two literals. The literals are denoted as 1,2,3,...n. In case of negation, we simply denote it as -1, -2,..etc. This gives us a very compact representation.

```
Algorithm: Black Box function for ground truth
Input: n, m.
Output: List of clauses
Method:
def getFx(m,n,verbose = True):
  lis = []
  coun = 0
  all_tups = []
  while(coun<m):
    s = np.random.choice(n, 2, replace=True)
    s = s + 1

    sig = np.random.randint(2, size = 2)*2-1
    s = sorted(s*sig)
    if(tuple(s) not in all_tups):
      all_tups.append(tuple(s))
      lis.append(list(s))
      coun += 1
  return lis
#Sample List:
#[[3, 3], [-4, 4], [-1, 4], [-3, 2], [-4, -4]]
```

For above sample list the underlying 2CNF is:

$$(x_2 \vee x_2) \wedge (\sim x_3 \vee x_3) \wedge (\sim x_0 \vee x_3) \wedge (\sim x_2 \vee x_1) \wedge (\sim x_3 \vee \sim x_3)$$

We have used 0-indexing, hence count starts from 0, and hence for every clause we see the subscript is one less than integer listed in sample list. The negative is used for integer in sample list for negating the literal in clause.

- **Creating classical solver for ground truth:** For checking accuracy and time running later, we need to equip black box function using a solver.

```
Algorithm: Classical Solver for ground truth
Input: List of clauses, n, m, t
Output: 1 if it satisfies at least t clauses, 0 otherwise
Method:

def solve(lis,n):
  dic = {}
  maxx = -1e10
  max_lis = []

  for i in range(2**n):
    val = bin(i)[2:].zfill(n)
    list_val = np.asarray(list(map(int,val)))*2-1
    summ = 0
    for j in range(len(lis)):
      summ += (np.sign(list_val[abs(lis[j][0])-1]) == np.sign(lis[j][0]))
              or
              (np.sign(list_val[abs(lis[j][1])-1]) == np.sign(lis[j][1]))

    dic[val] = summ
  for i in range(2**n):
    val = bin(i)[2:].zfill(n)
    # list_val = np.asarray(list(map(int,val)))*2-1
    if(dic[val]==max(dic.values())):
      max_lis.append(val)
  return max_lis,max(dic.values())
# Give the classical output: 1 or 0 : depends if at least t clauses are satisfied or not
def classical(lis,n,t):
  outs,outs2 = solve(lis,n)
  if(outs2>=t):
    return 1
  else:
    return 0
```

ii. Devising Mechanism for Separator Circuit

We need to repeat the below process for all of the m clauses:

- We first need three gates dependent on γ : R(γ) (1-qubit), CR($-\gamma$) (2-qubit), CCR($-\gamma$) (3-qubit)

```
# Get the Three gates
def gamma_matrix(size, gamma):
  matrix_R = np.identity(size, dtype=complex)
  matrix_R[-1][-1] = np.exp(-1j*gamma)
  return matrix_R

def obtain_gamma_gates(gamma):
  gate1_rgamma = Operator(gamma_matrix(2, gamma))
  gate2_crgamma = Operator(gamma_matrix(4, -1*gamma))
  gate3_ccrgamma = Operator(gamma_matrix(8, -1*gamma))
  return gate1_rgamma, gate2_crgamma, gate3_ccrgamma
```

Here, Oracle is a custom gate implementation.

- We observe that each clause is a union (or disjunction) of two literals. And if for a particular assignment, the clause is satisfied, the separator will multiply the tensor product of qubits by $exp(-j * \gamma)$, where $j = \sqrt{-1}$
- Handling negation: Further, any literal can be negated. So we use a NOT gate for that particular qubit, use it for a clause (as indicated in previous step), then undo this process by using another NOT gate. This is shown in Figure 42

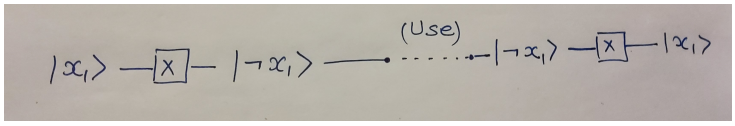


Figure 42: Handling negation

We handle these negation case as shown in below implementation:

```
#Handling negation and literal cases:
def separator_utility(circuit, first, second, qubits, gamma):
  # parsing
  line1 = np.abs(first) - 1
  line2 = np.abs(second) - 1
  gate1_rgamma, gate2_crgamma, gate3_ccrgamma = obtain_gamma_gates(gamma)
  # Negating the qubits
  if(first < 0):
    #meaning the literal is negated
    circuit.x(qubits[line1])
  if(line2!=line1 and second<0):
    circuit.x(qubits[line2])

  circuit = utility_helper(circuit, line1, line2, qubits, gate1_rgamma, gate2_crgamma, gate3_ccrgam

  # Undoing negation of qubits
  if(first < 0):
    #meaning the literal is negated
    circuit.x(qubits[line1])
  if(line2!=line1 and second<0):
    circuit.x(qubits[line2])
  return circuit
```

- Depending on the two literals, our circuit can be one of the two types:
  - Both literals in one clause are same:

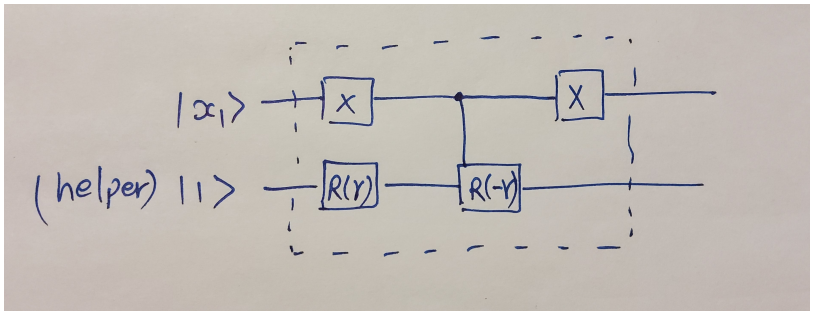


Figure 43: Both literals in one clause are same

– Both literals in one clause are different:

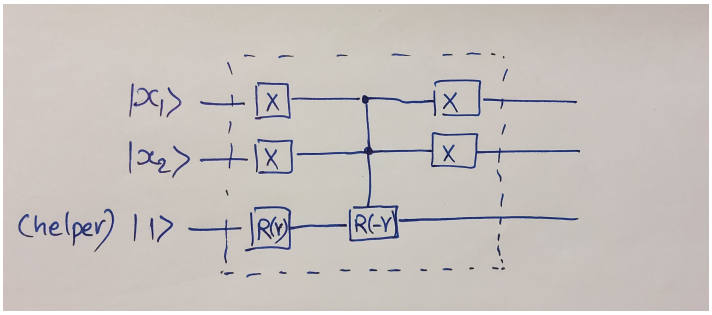


Figure 44: Both literals in one clause are different

We handle these two cases as shown in below implementation:

```
#Separator implementation
def utility_helper(circuit, line1, line2, qubits, gate1_rgamma, gate2_crgamma, gate3_ccrgamma)
  if(line1 == line2):
    circuit.x(qubits[line1])
    circuit.unitary(gate1_rgamma, -1)
    circuit.unitary(gate2_crgamma, [line1, -1])
    circuit.x(qubits[line1])

  else:
    circuit.x(qubits[line1])
    circuit.x(qubits[line2])
    circuit.unitary(gate1_rgamma, -1)
    circuit.unitary(gate3_ccrgamma, [line2, line1, -1])
    circuit.x(qubits[line1])
    circuit.x(qubits[line2])
  return circuit
```

iii. Design *Mix(β)*:
This is a relatively straight forward implementation, as it is dependent on number of qubits, or n, and involves applying Rotation gate to all the qubits (except helper qubit).

```
#Mixer implementation
def createMixer(n,beta):
    R = np.array([[np.cos(beta),-1j*np.sin(beta)],[-1j*np.sin(beta),np.cos(beta)]])
    return Operator(R)
```

iv. Measure z and determining count(z): The bitstring generated at the end will be used to evaluate Count(z). If count(z) $\geq$ t, we output 1, else we output 0. Further, we aim to obtain a reliably better solution by varying the values in (γ, β) pair.

We expect that Separator circuit creation time depends on the no. of clauses it needs to process, as for each such clause, it needs to use either 1- , 2- or 3-qubit gates, which is otherwise constant, independent of n.

This is further illustrated in the experiments and analysis that we have added in the following sections.

1.7.2 Code readability: Special Note on Porting from Cirq to Qiskit

Special Note:
As the entire code has been ported from cirq to qiskit, before porting every line of code, the original cirq code

has been commented, so it clearly illustrates a reader to check the validity of the code.

We have modularized our code and used the name of variables and functions, following the conventions of the problem statement or the names are self indicated of the utility they perform. "Verbose", a parameter when passed as true by the user would generate detailed description of the main circuit An exhaustive description of the functions used for implementation is enlisted below for reference:

- `getFx(m,n,verbose = True)`: Here m, n denote m clauses and n variables. This would output a list (or the boolean formula representation), containing m entries each representing one clause. Each such clause would be having 2 values, out of 1,2,....n, -1,-2,...-n, which denote that particular qubit/variable is involved in the clause. The minus sign denotes negation of a particular variable in the literal.
- `solve(lis,n)`: The classical solver that takes input the boolean formula and exhaustively search for the solution across all possible variables, denoted as n. It returns the list of possible solutions and count.
- `classical(lis,n,t)`: This takes the lis and gives the clear output (analogous to QAOA quantum solver) in classical domain dependent on t. Here t can be from 1,2,....m.
- `createMixer(n,beta)`: This function takes β and n and returns the Rotation - β matrix which is then used as a gate, applied on all the qubits.
- `gamma_matrix(size, gamma)`: Returns the rotation gamma matrix for a given gamma and of a given size
- `obtain_gamma_gates(gamma)`: It generates 3 different gates: R(γ), CR($-\gamma$), CCR(-γ) and returns these gates as output depending on the value of γ.
- `utility_helper(circuit, line1, line2, qubits, gate1_rgamma, gate2_crgamma, gate3_ccrgamma)`: line1 and line2 denotes the indices of particular variable/qubit involved in a particular clause; circuit is the equivalent circuit; qubits denote linequbits passed from maincircuit;
  `gate1_rgamma, gate2_crgamma, gate3_ccrgamma` are all gates passed as input. It applies the separator logic and returns the circuit.
- `separator_utility(circuit, first, second, qubits, gamma)`: first and second denote the list entries (not indices) ranging from 1,2,...n,-1,-2,...-n. qubits are line qubits. It handles the separator logic of negation of a literal and calls `utility_helper(circuit, line1, line2, qubits, gate1_rgamma, gate2_crgamma, gate3_ccrgamma)` to perform the case by case job.
- `runMainCircuit(n,m,t,lis,beta_divs = 5,gamma_divs = 5,repeats=1, verbose = True)`:
  Provide the number of variables:n
  Provide the number of clauses:m
  Provide the number of clauses to be satisfied:t
  Provide the boolean formula in the format:lis (or use `getFx(n,m)` to obtain the randomized lis
  Provide the number of beta divisions the user wants to investigate.
  Provide the number of gamma divisions the user wants to investigate.

All the subsequent case-wise analyses are appended in the later sections.

1.7.3 Parametrizing the solution in n

We have successfully parameterized the solution in terms of n: as the number of qubits that we have are dependent on n (alongwith additional helper qubit). Moreover, the gates that we have added include X, H, R, CR, CCR. The number of gates in the circuit are directly a function of m. In comparison to classical approach, which uses all 2^n possible solutions for satisfying the Max2Sat problem, the implementation of QAOA is linear in the sense that we ran it for O(n*m) for given number of γ-divisions and β-divisions.

Moreover, we didn't use (nxn) custom gate - matrix implementation for separator logic circuit. Instead we devised optimized approach for implementing separator logic through use of 1-qubit, 2-qubit and 3-qubit gates, as entailed in previous sections.

Next section illustrates the example of how we parametrized in greater detail.

1.7.4 Sample Circuit Generated for a particular boolean formula:

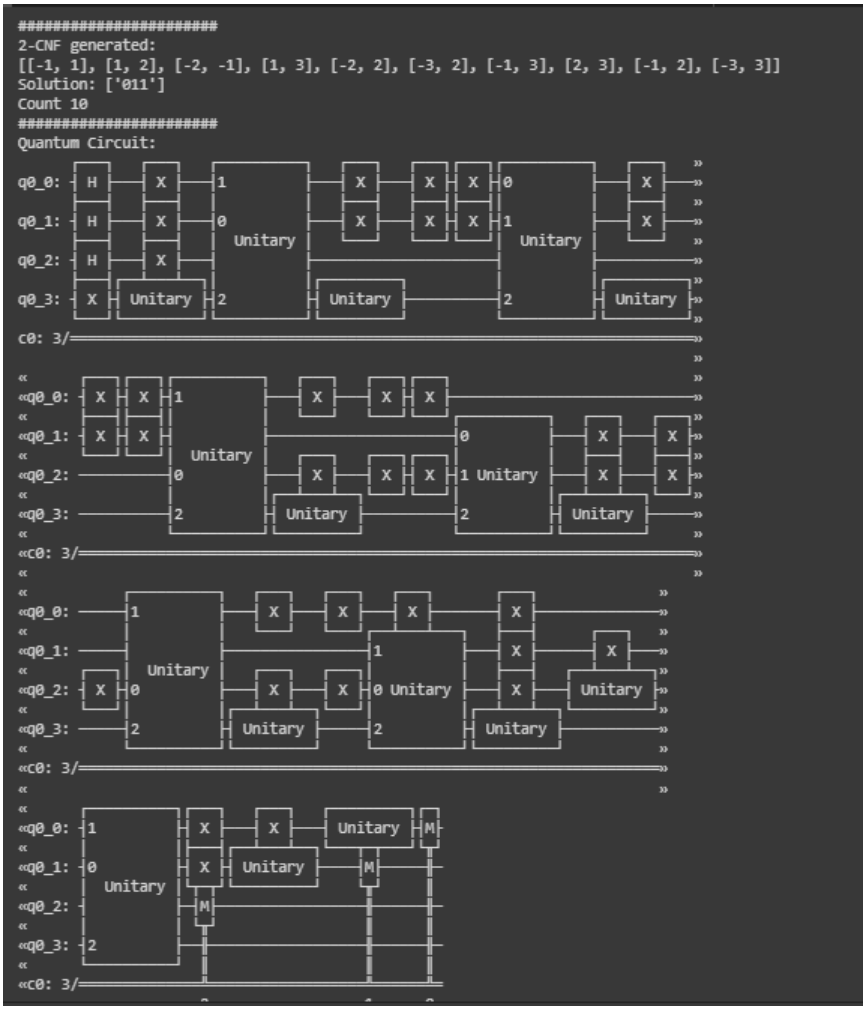


Figure 45: Sample Circuit (B)

As shown in the Figure 45,

2-CNF denotes boolean formula

$$[[-1,1],[1,2],[-2,-1],[1,3],[-2,2],[-3,2],[-1,3],[2,3],[-1,2],[-3,3]]$$

is same as

$$(\neg x_0 \vee x_0) \wedge (x_0 \vee x_1) \wedge (\neg x_1 \vee \neg x_0) \wedge (x_0 \vee x_2)$$
$$\wedge(\neg x_1 \vee x_1) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_0 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2) \wedge (\neg x_2 \vee \neg x_2)$$

solution denotes the classical solution

count denotes the possible no. of satisfied clauses

t=10 is greater than this count

Hence, we don't have solution neither from quantum nor from classical, which matches the ground truth

Output: 0

1.7.5 Histogram of assignment values for variables

Choice of gamma and beta divisions for series of experiments on Quantum Computer

- Firstly, when we had conducted an experiment on single circuit using cirq where we tried to run the circuit on particular β and γ for 1000 repetitions/shots. We had done this for whole search space then and analaysed the histogram of variables assignments outputs from circuit for a particular β and γ which has the highest count of outputs which satisfy more than equal to *t* clauses (as described in problem statement).
- It is important to mention that when we performed the same experiment only using cirq (without QC), we were able to use best possible gamma and beta by searching through the whole space. But here, we had to choose a reasonable value of beta and gamma division space, and eventually decrease these divisions as we increased n.
- The reason we had to do so was: As beta and gamma divisions increase, the number of times we need to call the IBM quantum computer increases, queuing delay in accessibility of quantum computer. Its effect asymptotically increases as the number of qubits increase. Hence, we are not able to retain the best value of gamma and beta divisions.
- Still, from what results we previously obtained, we had used gamma and beta to be 5 as default. We could still reasonably use default value of 3 for n=1 and n=2. As n=3 and n=4, we reach the bottleneck of IBM which is 5 qubits (we are using a helper qubit as well), there we make gamma and beta division as 1. So, the following plots must be interpreted using this reference as the key.

Plotting technique: We increase gradually n by 1 and see the outputs, compare it with the classical for accuracy, and interpret it using a histogram for both the simulator as well as IBM quantum computer.

QAOA suggests that good outputs will be amplified after a reasonably good number of choice of gamma and beta.

Following were the histogram obtained:

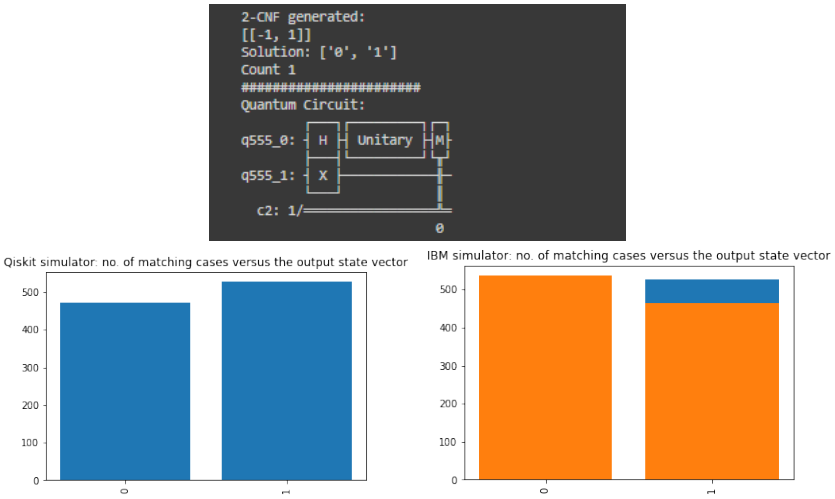


Figure 46: **n = 1** First figure shows the 2CNF (with n = 1 and m = 1 and t = 1) used and corresponding set of solutions which give the maximum count and bottom figure is the histogram of assignment values for 1000 iterations for β = 3 and γ = 3. IBM figure is overlapped for comparison of both qiskit as well as IBM as the two solutions 0 and 1 are both equally likely.

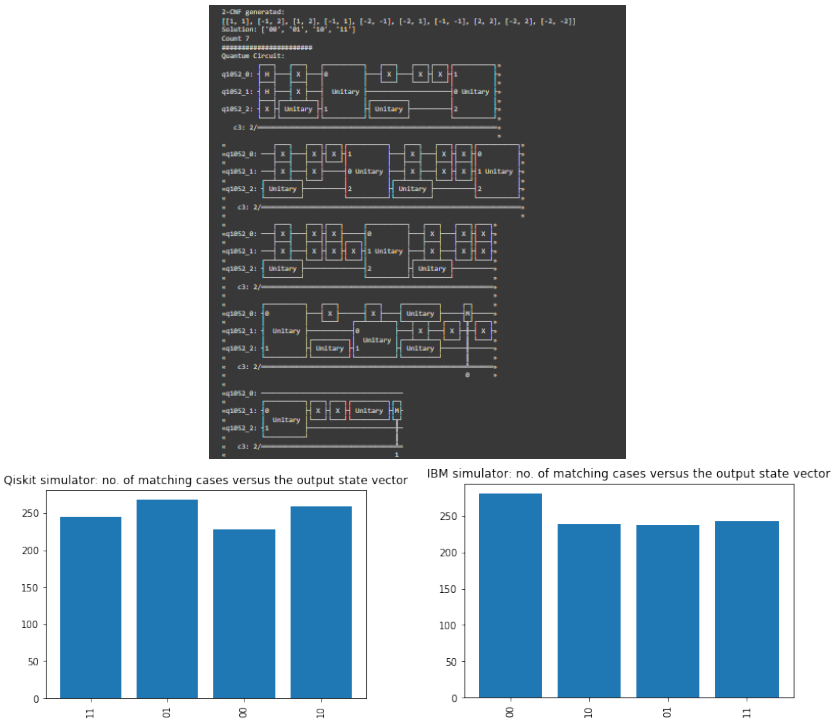


Figure 47: **n = 2** First figure shows the 2CNF (with n = 2 and m = 10) used and corresponding set of solutions which give the maximum count and bottom figure is the histogram of assignment values for 1000 iterations for β=3 and γ=3.

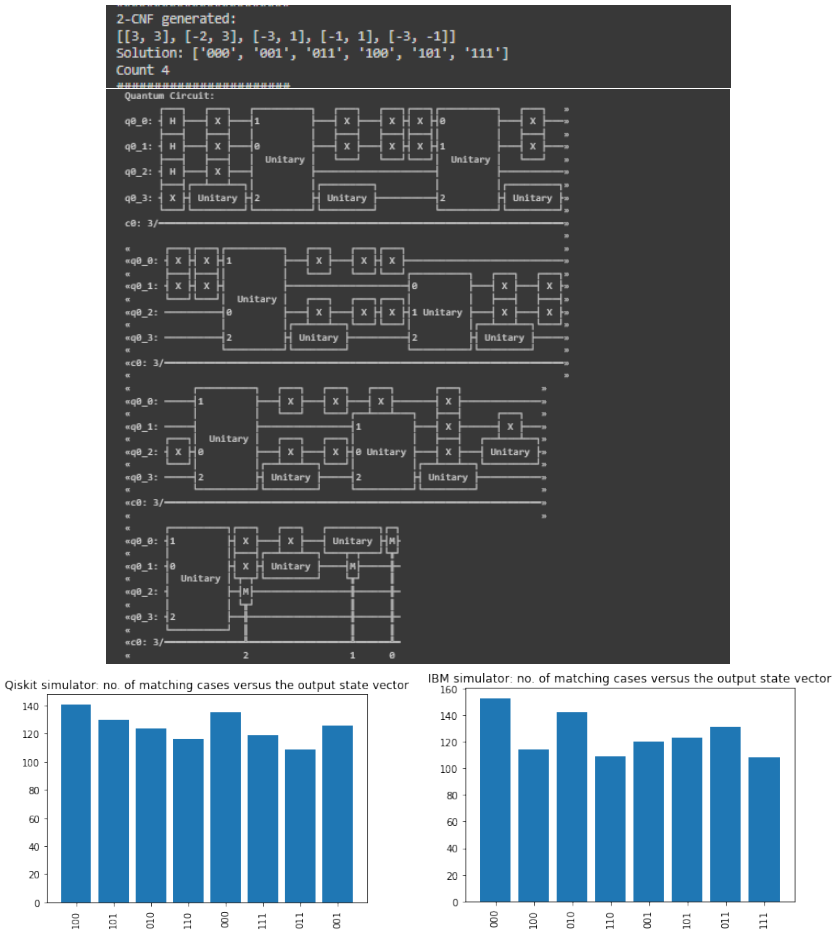


Figure 48: **n = 3** First figure shows the 2CNF (with n = 3 and m = 10) used and corresponding set of solutions which give the maximum count and bottom figure is the histogram of assignment values for 1000 iterations for best β and γ.

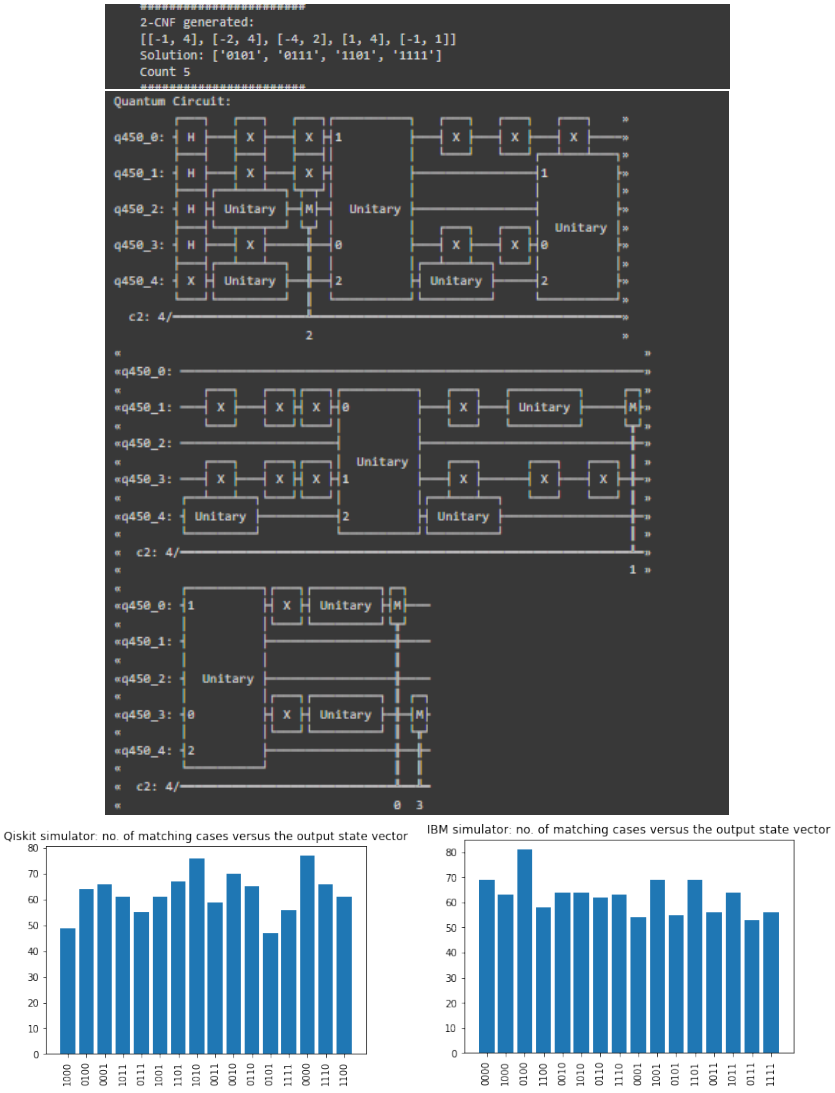


Figure 49: **n = 4** First figure shows the 2CNF (with n = 4 and m = 10) used and corresponding set of solutions which give the maximum count and bottom figure is the histogram of assignment values for 1000 iterations for best β and γ.

- We observe as expected that histogram's peaks, although not as sharp as for cirq simulator, still seem pretty reliable, atleast for n=1,2. Even though the amplification of good outputs was not that great, it still goes to show that IBM quantum computer appears to be unreliable in this case: since we had used less gamma and beta divisons, as described above the relationship between increase in divisons with n.
- The rest of assignment values get a lesser peak in histogram as the separator and mixer circuits reduced their coefficients and hence the probability of observing
- We further observed that over various runs, sometimes a sub-otimal assignment gets a higher peak over optimal assignment. This sub-optimal assignment was observed to satisfy 1 or 2 less clauses than the optimal clause. The primary reason for this is that QAOA is an approximation algorithm and can give sub-optimal output (even for various iterations). In this process, since the sub-optimal assignment has almost equal clauses satisfied as optimal one, it leads to getting picked up many times during runs.
- Given the times that we were able to run, it shows a comparative trend in comparison to the simulator. Our code is written scalably and if connection establishment/queuing delay in calling to quantum computer is not that much, it is anticipated that this will give us freedom to choose gamma and beta, and accuracy should asymptotically be similar for both.

1.7.6 Variation of various Times with n (number of variables): Comparison of Cirq and IBM

We conducted various experiments where we varied number of variables and observed the various parts of circuit times

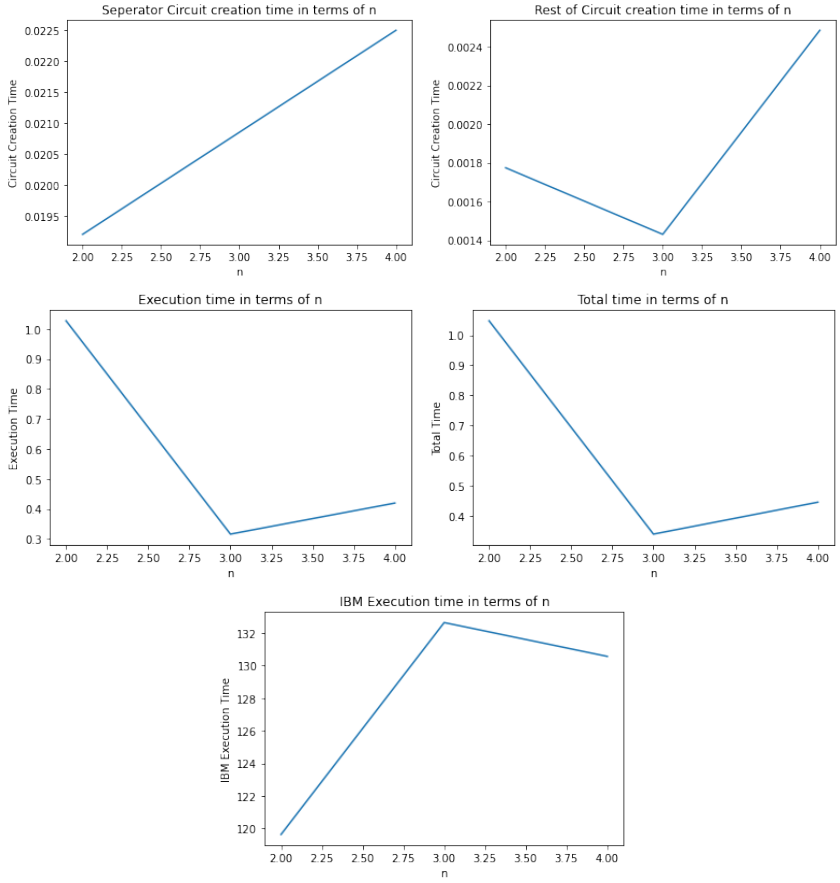


Figure 50: Variation of separator circuit creation time, rest of circuit creation time, Execution and total running time vs number of variables

Following were the observations in general

- The separator circuit was the deciding factor in the overall circuit times. In general separator circuit time heavily dependent on the actual clauses, hence we dont see a clear increase in seperator circuit times as we increase n.
- The seperator circuit time also decided the execution time, since majority of gates were contributed by seperartor circuit. Hence we see a similar, not exact, curve in execution time vs n graph. We see its linear dependence on the number of qubits(n).
- The rest of circuit creation time was pretty small and random and highly dependent on how the compiler and interpreter works during run time, since such small times were comparable to other delays associated with run times of compilers. It may have points where it will decrease for higher n too.
- The total time hence was primarily decided by execution and separator circuit creation time, which indeed actually dependent effectively on seperator circuit creation time, hence we see a similar curve for total time vs n as well.

1.7.7 Variation of various Times with m (number of clauses): : Comparison of Cirq and IBM

We conducted various experiments where we analysed the times taken by various componets of circuit vs number of clauses. For this experiment, we increased the number of clauses in sequential manner, where we increase the number of clauses for our CNF one by one.

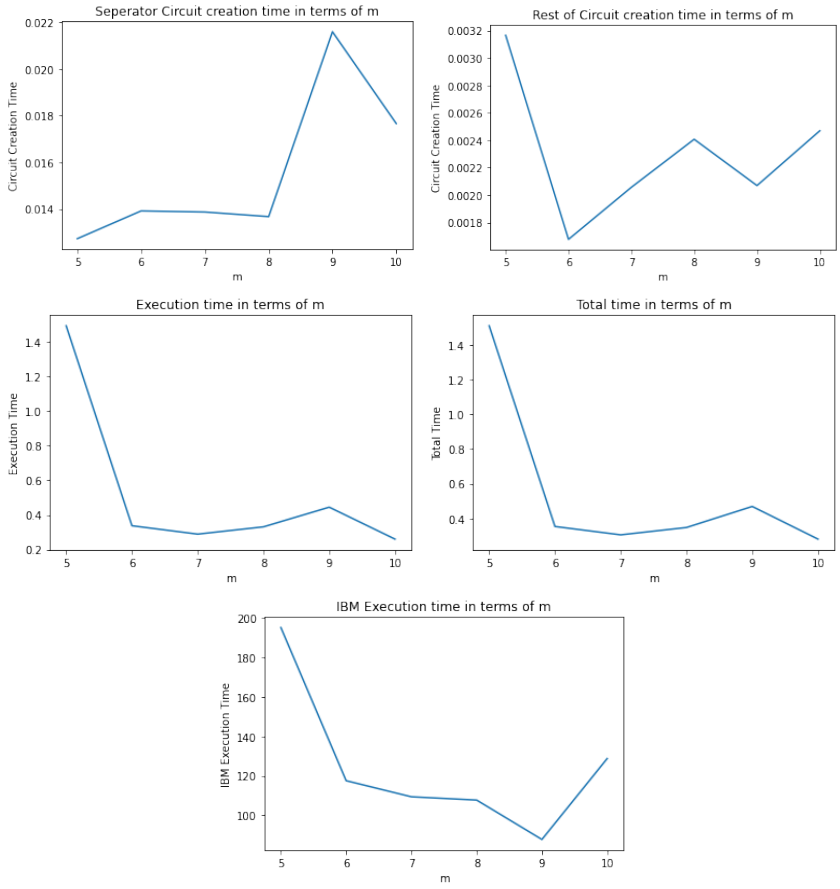


Figure 51: Variation of separator circuit creation time, rest of circuit creation time, Execution and total running time, and IBM execution time(excluding connection establishment) vs number of clauses

Following were the observations in general

- As discussed before, execution time and total time, more or less depend on seperator circuit creation time, and this is observed here too with similar curves in graph.
- The separator circuit creation time this time was observed to increase always, leaving the outlier at m=10, this circuit iterates over the clauses and adds the gates correspondingly. Since, we added the clauses in sequential manner to our CNF, hence as m increased, the circuit creation time also increased.
- The rest of circuit creation time was again very small and had randomness associated with it due to reasons mentioned in previous section. The total time was not affected due to it much since the scale of values for this part were very small.
- **The IBM execution time has been taken into account excluding the initial connection establishment time with the quantum computer, that is the queuing delay.**

1.7.8 Variation of Accuracy versus (γ, β): : Comparison of Cirq and IBM

In this section ,we analyse how our approximation algorithm works in case we increase our search space. Specifically, we increased the number of beta and gamma divisions, keeping other paramter fixed, and observed the accuracy which is achieved by comparing output everytime with classical solver. For both of the experiments, we generated random 20 2CNF and used them to verify the accuracy for each gamma division.

- Accuracy vs Gamma divisions

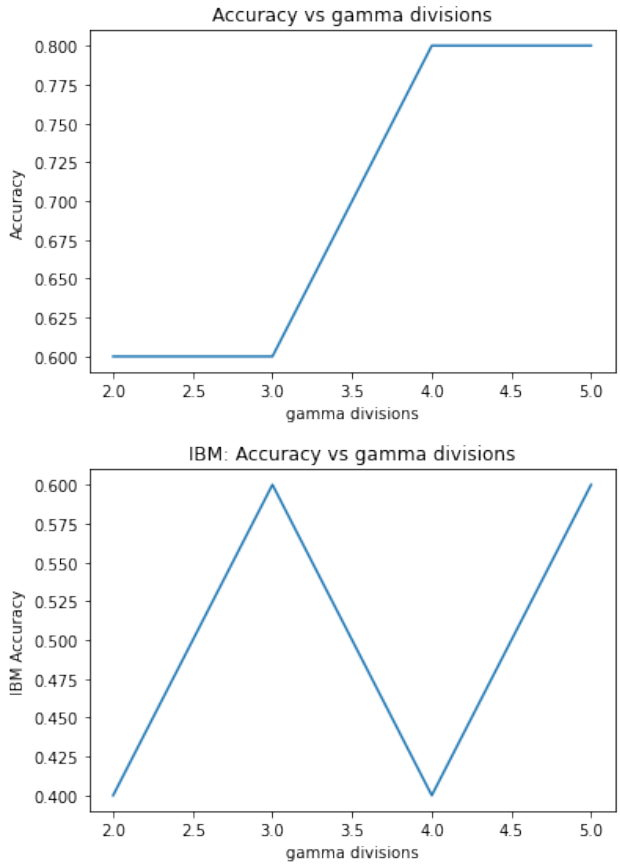


Figure 52: Variation of accuracy of circuit vs number of divisions in γ

Following were observations in general:

- We see that as we increase the number of divisions in γ, the accuracy in general increased.
- This we quite expected since our search space increases and hence we find a better optimal γ which will better separate to get the optimal assignment.
- This trend is not visible in quantum computer results for qubits at higher gamma divisions, where it shows unreliability in output accuracy.

- Accuracy vs Beta divisions

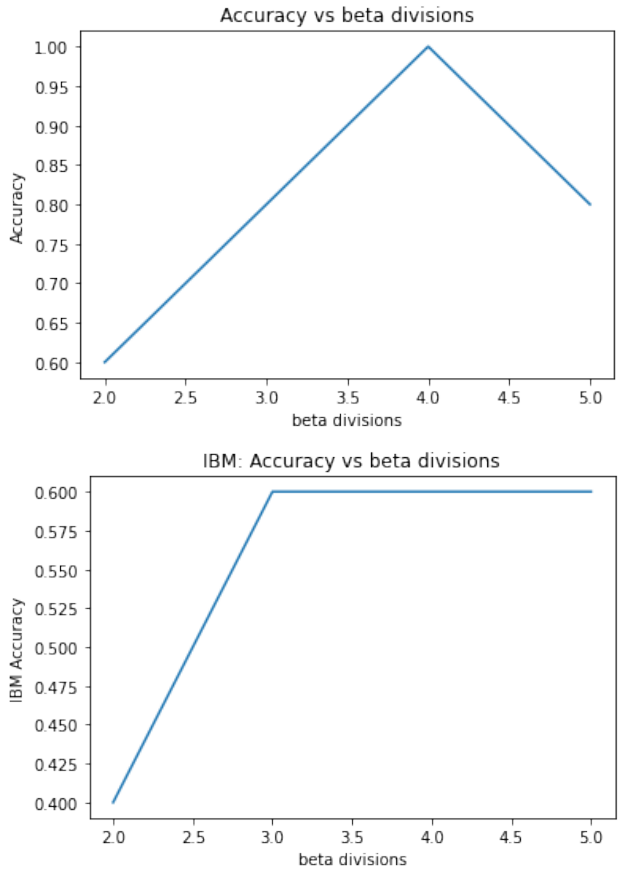


Figure 53: Variation of accuracy of circuit vs number of divisions in β

Following were observations in general:

- We see that as we increase the number of divisions in β, the accuracy in general increased.
- Again this was expected since our search space increases and hence we find a better optimal β which will better separate to get the optimal assignment.
- Though this trend is not so vividly visible in ibm quantum accuracy for higher number of beta divisions, it can atleast be said to be better than the trend we observed for gamma (in previous section), where trend was found to be decreasing at higher divisions.

  Further, as the qiskit simulator itself becomes an outlier at the 4.5 and 5.0, it is reasonable to assess that quantum computer performed relatively reliable in case of beta divisions.

In both of above analysis, its observed that sometimes there is drop in accuracy with increasing divisions. Following are the possible reasons:

- In both of graphs, over many random clauses, we observed that the decrease is not that much.
- One of the reasons for such dip in accuracy by little amount is that, the set of divisions of both β and γ are not exactly the superset of previous iteration set. For example, for β divisions = 3, we will have set $\beta = [0, \frac{\pi}{2}, \pi]$ and for β divisions = 4, we will have set $\beta = [0, \frac{\pi}{3}, \frac{2\pi}{3}, \pi]$. Here we observe that although we increased our search space, the optimal β might be near $\frac{\pi}{2}$ than either of $\frac{\pi}{3}$ or $\frac{2\pi}{3}$ and in that case accuracy will be higher for first case. Similar story for γ too. Hence we observe a dip in accuracy sometimes.
- Second reason for such fluctuations is that eventually this is an approximation algorithm and its very much possible that while measuring we would have measured a lower probable assignment of variables, which is

sub-optimal and hence we observe sometimes lower accuracy. This is completely dependent on that particular run time.

- The dip in accuracy is always not much, which is obvious due to above reasons. In general, once we have increased divisions to great extent, the first cause effects starts to mitigate.

2 Experience

Which modifications of the programs did you have to make when moving from the simulator to the quantum computer?

1. We had to port the entire code from Cirq to qiskit as IBM machine supports that.
2. to_qasm was an option to port but in cases where more complex gates were used, this was an issue.
3. As the backend did not support more than 5 qubits, we had to use noise models to replicate the IBM Quantum Computers.
4. The name of the machine as a function parameter is needed to be passed in as the backend object. We also had to make some minor code changes such as the API token to account for the calls to the IBM quantum computer to be remote and require authorization. Other than that though, the jobs can be run in the same way with the execute method.
5. On creating oracle with identity matrix, a few bugs were observed.
6. To account for the run method being asynchronous, we need to continuously poll the backend to get the status of the job.

3 README

1. Download the python notebook. All the required libraries are installed in case you are running on colab. If you are running locally install numpy, ortools and matplotlib.
2. Follow these links: Install numpy and Install matplotlib.
3. Run the notebook one by one.
4. **DJ**: Call runMainCircuitDJ with parameters n, type of function (optional - 0 means constant and 1 means uniform) and verbose (optional). eg: runMainCircuit(3, 0, True).
5. **BV**: Call runMainCircuitBV with parameters n and verbose (optional). eg: runMainCircuit(3, True).
6. **Simons** : Call runMainCircuit with parameters n and verbose (optional). eg: runMainCircuit(3, True).
7. **Grover's algorithm**: Call runMainCircuit with parameters n, a (optional) and verbose (optional). eg: runMainCircuit(3, 1, True).
8. **Shor**: Call runMainCircuit with parameters n, verbose (optional, default = True) and max_attempts (optional, default = 25). eg: runMainCircuit(21, True, 20).

9. **QAOA**: Call runMainCircuit with parameters n, m , t,
    - lis (The 2CNF in format mentioned before). If kept empty, the function will create a randomized 2CNF during run time using `getFx(m,n)`. The CNF can also be explicitly provided to the function.
    - `gamma_divs` (optional, default = 5)
    - `beta_divs` (optional, default = 5)
    - repeats (optional, can be modified to draw histograms, when we wish to determine probabilistic outputs, default value = 1)
    - verbose (optional, default = True)

References

[1] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. *A Quantum Approximate Optimization Algorithm*. 2014. arXiv: `1411.4028 [quant-ph]`.