QISKIT - THE ENGINEERING GROUP

QUANTUM PROGRAMMING

Attiano Purpura-Pontoniere attiano@cs.ucla.edu **Nishant Sabharwal** nsabharwal@ucla.edu

Pratik Sathe psathe@physics.ucla.edu

June 14, 2020

ABSTRACT

We use Qiskit (version = 0.14.1) to run four popular quantum algorithms (Deutsch-Jozsa, Bernstein-Vazirani, Simon's algorithm and Grover's search algorithms) on IBM's quantum computers. For each of the algorithms, we benchmark the performance, measure 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. Finally, we compare running on a quantum computer to a quantum simulator.

1 Design and Evaluation

1.1 Deutsch-Josza algorithm

First, we summarize 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.*

While a classical algorithm requires $2^{n-1}+1$ calls to the function f, the Deutsch-Jozsa algorithm finds a solution with just one call to the oracle U_f, which is defined by its action on the computational basis as follows:

$$U_f \left|x\right\rangle \otimes \left|b\right\rangle = \left|x\right\rangle \otimes \left|b \oplus f(x)\right\rangle, \tag{1}$$

for all $x \in \{0,1\}^n$ and $y \in \{0,1\}$. The Deutsch-Jozsa circuit is shown in Fig. 1. If the measured state is $\left|0\right\rangle^{\otimes n}$, then the algorithm concludes that f is constant, and balanced otherwise.

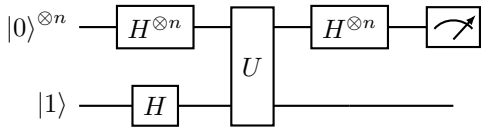
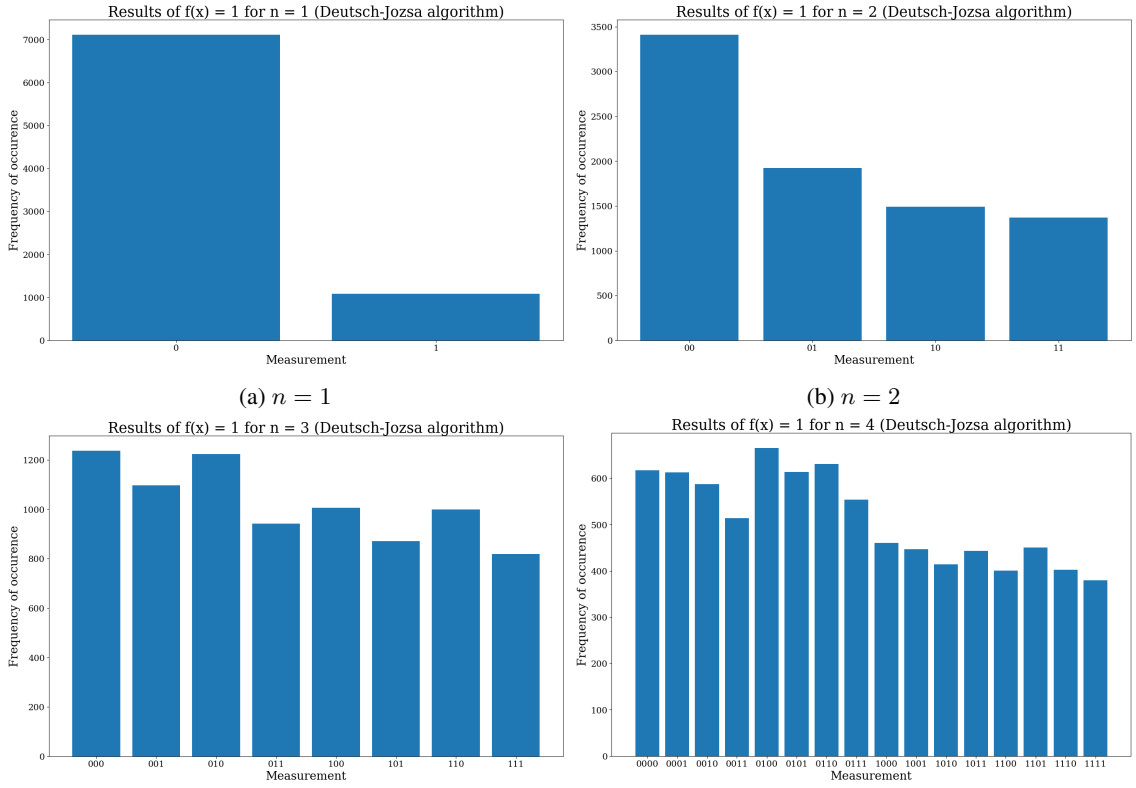


Figure 1: Deutsch-Jozsa algorithm

Now we discuss various aspects of the run time and reliability:

We faced a problem with the built in transpiler and assembler. On the highest optimization level, the circuit depth and size were too large when running with more than 5 qubits. The library would throw an error at runtime that the circuit runtime was greater than the backend's refresh rate. As a result, we limit our analysis to 5 qubits and under for traditional methods and then using alternative methods to extend the range.



(a) $n = 1$ (b) $n = 2$

(c) $n = 3$ (d) $n = 4$

Figure 2: Deutsch-Jozsa algorithm: Results of 8192 trials of a constant function for various values of n.

- **Statistics of Results of Running on a Quantum Computer**: To demonstrate the effect the number of qubits has on the error rate, we ran the Deutsch-Jozsa circuit on IBM's London quantum computer. London is a 5 qubit quantum computer. We noticed that choice is important. The 5 qubit Burlington machine would throw the same error of the circuit being too large for balanced functions. We did not run it on a larger machine, because as mentioned previously, it would throw an error for larger number of qubits. We chose a constant function because under perfect conditions, we would measure 0^n every time. Another error we faced was a bug in the transpiler, which would throw an error if we tried to add a gate that was built from the identity matrix. As a workaround, we simply do not add the gate if it's built from the identity matrix, since the identity matrix does not change the circuit. This occurs when $f(x) = 0$. This however means that this circuit would be much shorter and unusually reliable so we took care to avoid this f in benchmarking.
  Figure 2 shows the result of running 8192 trials of the Deutsch-Jozsa circuit on the IBM London quantum computer for the constant function, $f(x) = 1$, for $n = 1$ through $n = 4$. Note how the noise increases with n. For reference, on a simulator, 0^n is measured with 100% probability. For $n = 1$, we measure 0 with close to 90% probability. For $n = 2$, the probability of measuring 00 drops below 50%. Though not ideal, both these cases are still manageable since a simple majority voting algorithm would still select the right answer. For $n = 3$ and $n = 4$, the probability of measuring 0^n approaches $\frac{1}{2^n}$. This means there is so much noise, the results are close to random. For $n = 3$, 000 is the majority by less than 1% and in $n = 4$, 0000 isn't even measured the most often. Clearly for $n > 2$, different trials would result in different results. We should point out that some of the balanced functions are more complex, so these results would be amplified for different U_f's. To highlight how troublesome this error rate is, we run an experiment where we select 10 functions at random and select the majority element as the final answer. Figure 3 shows the percentage of times that it was correct. For $n = 1$, this method is always correct. However, as n grows larger, this method is increasingly unreliable because it is incorrect a larger percentage of the time. Interestingly enough, we would expect a noisy circuit to be correct a greater portion of the time. Assuming a truly random circuit, a balanced function has a probability of $\frac{2^n-1}{2^n}$ of being right. This is because any measurement that is not 0^n would be correct. At $n = 4$ though, 0^n starts to get measured very frequently. Perhaps when the IBM computers have a large error, they tend to report 0.

In order to examine why the error grows for larger values of n, Qiskit provides two useful measurements. Circuit depth is the length, or number of gates on the critical path and size is the total number of gates. Quantum gates have much higher error rates than classical gates. The more gates there are, and the more gates a calculation passes through, results in a higher error rate overall. Figure 4 shows the results of these values for $n = 1$ through $n = 4$ for $f(x) = 1$ at the highest optimization level. The number of gates grows exponentially which explains why the total error rate grows so rapidly.

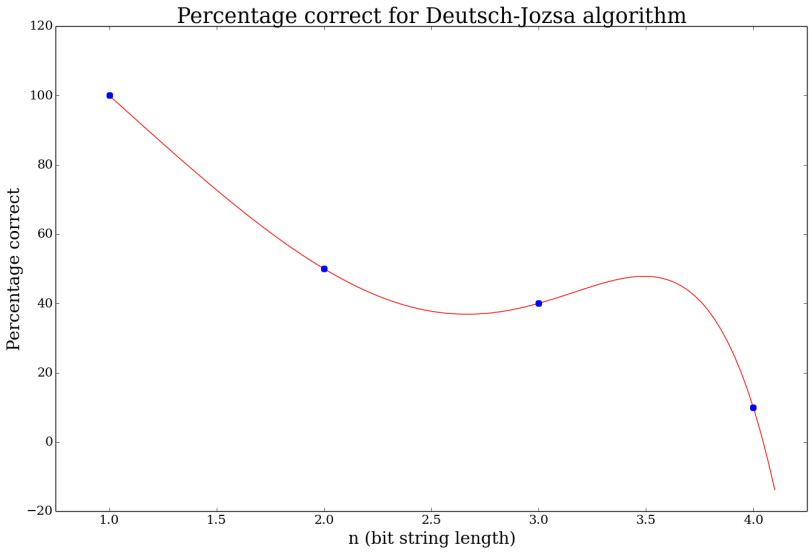
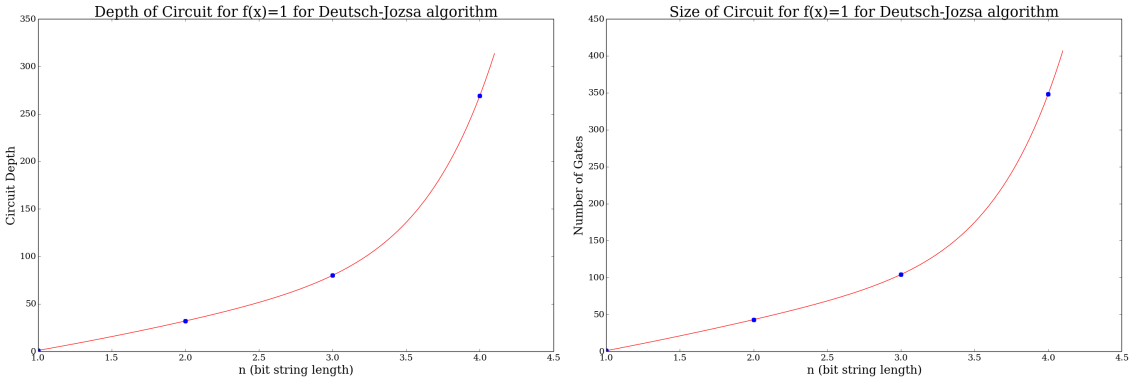


Figure 3: Deutsch-Jozsa Algorithm: Percentage correct for different values of n.

- **Dependence of U_f on execution time** In order to measure the oracle function's impact on execution time, we select 100 input functions of $n = 4$ at random and measure the execution time. The results are presented in Figure 5. Note there is a little variance in execution time. The time is divided by the number of trials so it is the average execution time for a single measurement. Qiskit's simulator saw a range that was about 3% of the median value where as PyQuil saw a much larger range. The quantum computer execution time had a median of 3.1 ms and the values were within 0.1 ms, except for a few outliers, so the variance is similar to the Qiskit quantum simulator. This is expected especially considering the transpiler tries to optimize the circuit so some circuits will be longer than others.



(a) Circuit Depth

(b) Circuit Size

Figure 4: Deutsch-Jozsa algorithm: Circuit size and depth for $f(x) = 1$

- **Scalability with n** In order to measure the scalability, we select 10 functions at random, always including $f(x) = 1$ and measure the time. There are two portions the consider. The first is the assembly time, which is done locally. This is measured using a timer. The next component is the actual execution time on the quantum computer. In order to do this we count on Qiskit to report the `time_taken` and we divide by the number of

trials to measure the average time of a single run. We take the average this for each f and present the results in Figure 6. Note the assembly time on the left increases exponentially with n. Since a classical computer is compiling a quantum circuit, there are scaling issues. On the right though, the actual run time of the circuit scales much better. The average execution time for $n = 4$ is only 10% larger than $n = 1$.

- **Comparison with Simulator** The largest difference between a quantum computer and a quantum simulator is our results are no longer deterministic. Due to error rates in gates and decoherence, we measure with noise so we must account for that. Unfortunately, the noise becomes quite large for $n > 2$, and it becomes difficult to get the correct answer. On the quantum simulator, we can run arbitrarily large circuits up to a certain amount before the exponential computation time becomes a problem. On a quantum computer, the hardware must be taken into account and some circuits may be too lengthy for the computer. The simulator has an exponential execution time so it scales very poorly. The quantum computer appears to scale better with n. However, assembly time is still exponential and the length of the critical path is exponential as well. However since gates are fast, the impact on the total run time is mitigated, at least for small values of n.
- **Modifications from Simulator to Quantum Computer** Very little code needs to be changed to go from a simulator to a really quantum computer. IBM designed it's backend abstraction so a simulator or a remote machine can be used. Unfortunately when doing the simulator, I misunderstood the abstraction and thought all that needed to be passed in was the name of the machine. I had originally passed the name of the machine as a function parameter so this needed to be changed to pass in the `backend` object. We also had to make some minor code changes such as the api token to account for the calls to be remote and require authorization. Other than that though, the jobs can be run in the same way with the `execute` method. We did need to make some changes to manually transpile so we could pass in the optimization level and benchmark the circuit size and assembly time. However, these are not required changes.
  We also had to deal with a bug based on the identity matrix. When running on a quantum computer, the machine would throw an error at run time if the custom gate was built from the identity matrix. This was a little frustrating because the simulator had no such problem. In an ideal world, we would want the two to throw the same errors. As a result, we have to check if the oracle is built from the identity matrix, and if so, we don't add the gate. This is technically correct since the identity matrix doesn't affect the circuit and its a large optimization, but some could argue it violates the oracle principle since you must inspect it.
  In practical terms though, changes needed to be made to account for the noise. Whereas on the simulator, we could make simple decisions based on the deterministic measurements, on the quantum computer we needed to run numerous trials, and find the majority element in the results to decide what was the measured value. Also, to account for the `run` method being asynchronous, we need to continuously poll the backend to get the status of the job.

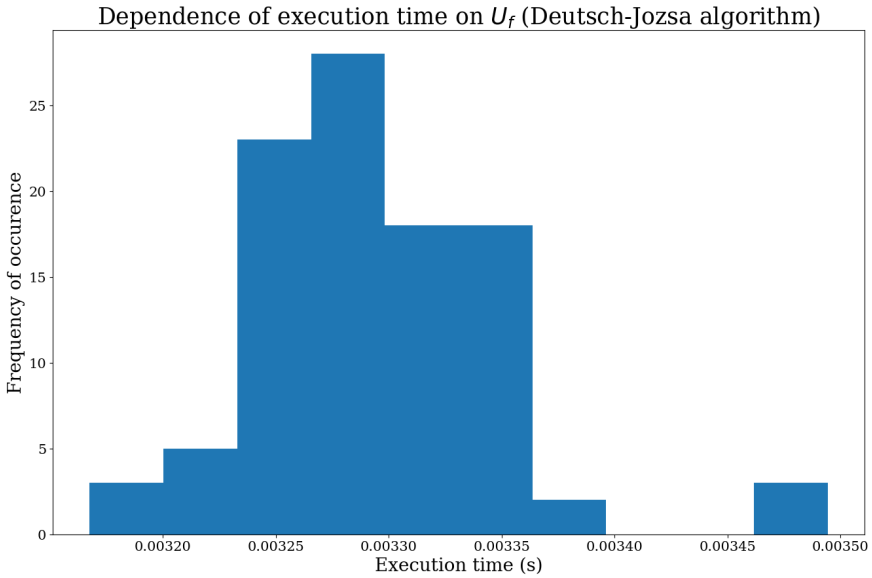
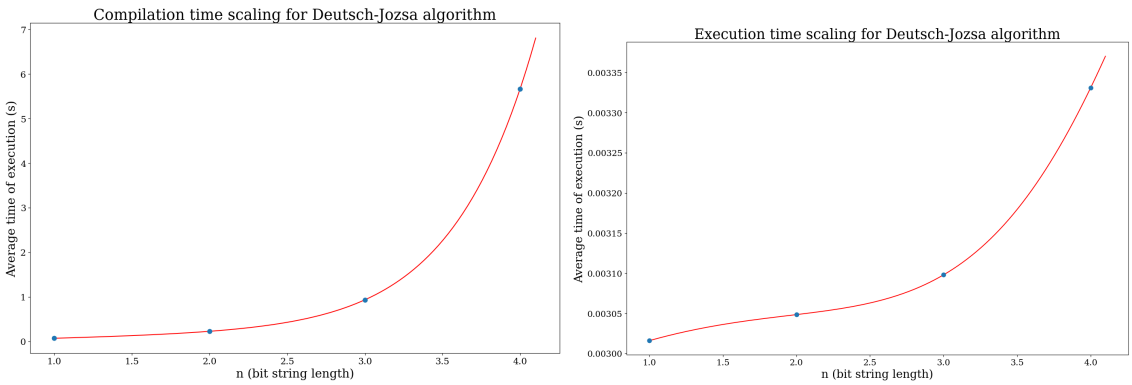


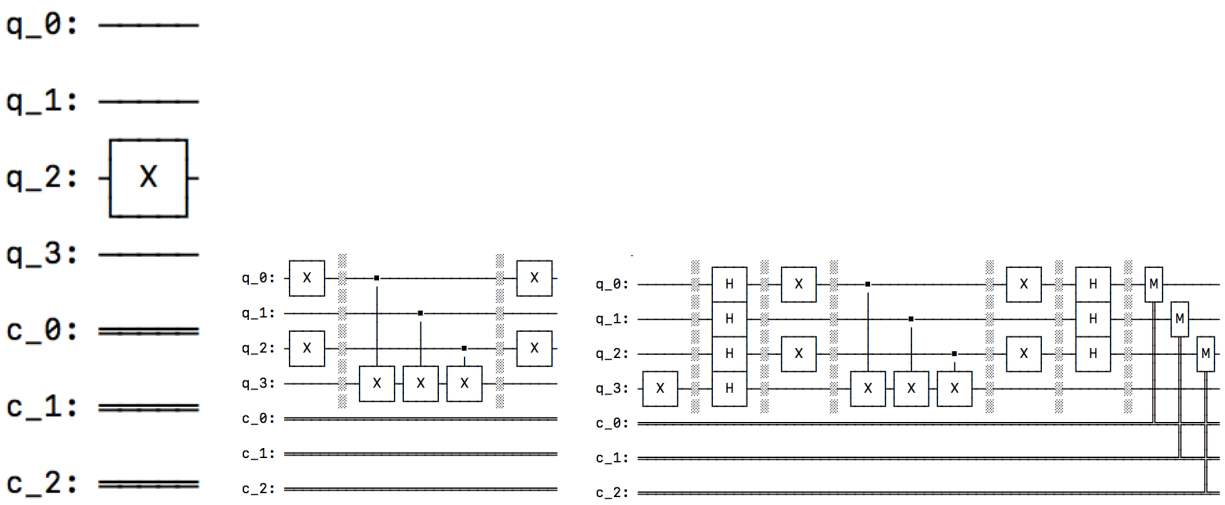
Figure 5: Deutsch-Jozsa: Execution time for different U_f

- **Hacking the Oracle** Unfortunately, finding out that we couldn't run on a computer with more than 5 qubits was an unsatisfying conclusion. Though building a circuit from a unitary matrix for any possible input function



(a) Assembly time for Deutsch-Josza. (b) Execution time scaling for Deutsch-Jozsa.

Figure 6: Deutsch-Jozsa: Benchmarking.

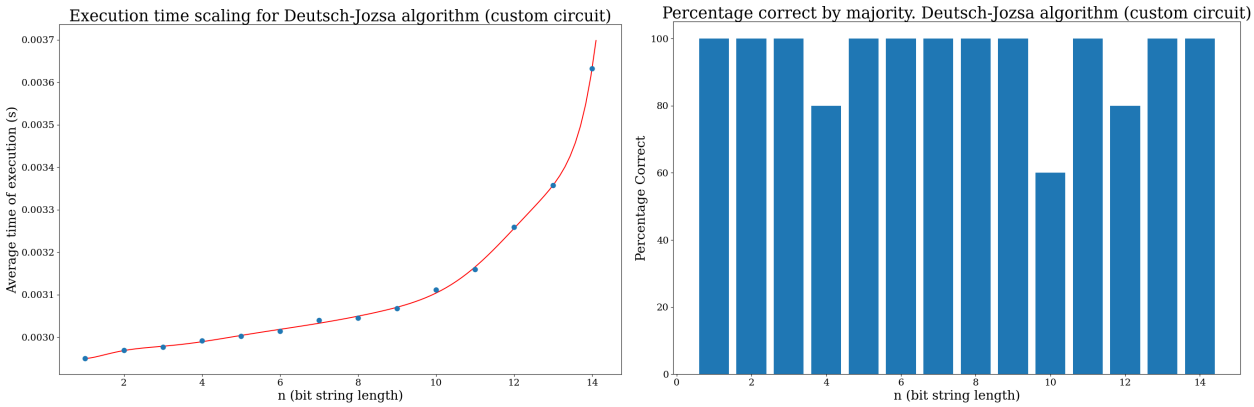


(a) Oracle: $f(x) = 1$ (b) Oracle: balanced (c) Complete circuit

Figure 7: Deutsch-Jozsa: Custom oracle gates.

is not possible for larger values of n, we could build an oracle gate which we know to be balanced or constant and run the circuit to benchmark performance and correctness. We note there are three possible input functions for dj. Either $f(x) = 0$, $f(x) = 1$, or f is balanced. If $f(x) = 0$, the oracle gate should simply be the identity matrix, or nothing. If $f(x) = 1$, the oracle gate is simply a `NOT` gate applied to last non-helper qubit. We build a balanced oracle gate by appying `cnot` gates from each qubit to the helper qubit. Now we can apply `not` gates before and after the `cnot` to any wire to generate arbitrary balanced oracles. Figure 7 shows the different oracles. We leave out $f(x) = 0$ since it is the identity. Figures 7a and 7b show the circuit diagram of U_f for $f(x) = 1$ and a balanced function. Figure 7c shows the custom oracle plugged in to the dj circuit. This method allows us to capture both constant functions. Unfortunately, it only allows us to capture 2^n of the possible ${}^{N}C_{N/2}$, where $N = 2^n$, possible balanced functions. We can hypothesize that there also exists simpler circuits for those remaining functions, but for now at least, we can study these simplified oracles. These oracles allow us to create shorter circuits which can run on larger computers to measure the performance and correctness.
To perform benchmarking, we create 5 custom oracles at random for $n = 1$ through $n = 14$. We always create the two constants and we create three random balanced oracles. For $n = 1$, there is some repetition. We then transpile the circuit and run on the 15 qubit Melbourne quantum computer. We measure the time taken, averaged over 8192 trials, and the percentage correct. We measure the percentage correct by taking the most frequent measurement as the measurement and reporting correct if the oracle is constant and the measurement is 0^n or if the oracle is balanced and the measurement is not 0^n. Figure 8 shows the results. Note the scaling is indeed much better than a simulator. There is only a 20% increase in run time from 1 to 14. The percentage

correct is much better than in our previous attempts where we built U_f by creating a unitary matrix. The percentage correct is a little misleading as it doesn't take into account noise. This is simply whether or not we would be right if we took the most frequent measurement of the 8192 shots. Again we note that given a lot of noise, the odds are very likely of being correct for a balanced function since only $\frac{1}{2^n}$ of the values is incorrect. The reason we are able to run for larger values of n is the oracle is greatly simplified. When we generate an operator by building a unitary matrix from f, the transpiler cannot optimize it well. Since we generate a short U_f, it can run and it has less errors. We note in the previous discussion for $n = 4$, the circuit depth is over 300. Generating an oracle using this method for a balanced function with $n = 14$ results in a circuit depth of 70. Even with such an improvement, there are still some incorrect measurements.



(a) Run time for Deutsch-Jozsa with Custom Oracles.

(b) Percentage Correct for Deutsch-Jozsa with Custom Oracles.

Figure 8: Deutsch-Jozsa: Benchmarking with custom oracles.

1.2 Bernstein-vazirani algorithm

First, we summarize the problem statement:

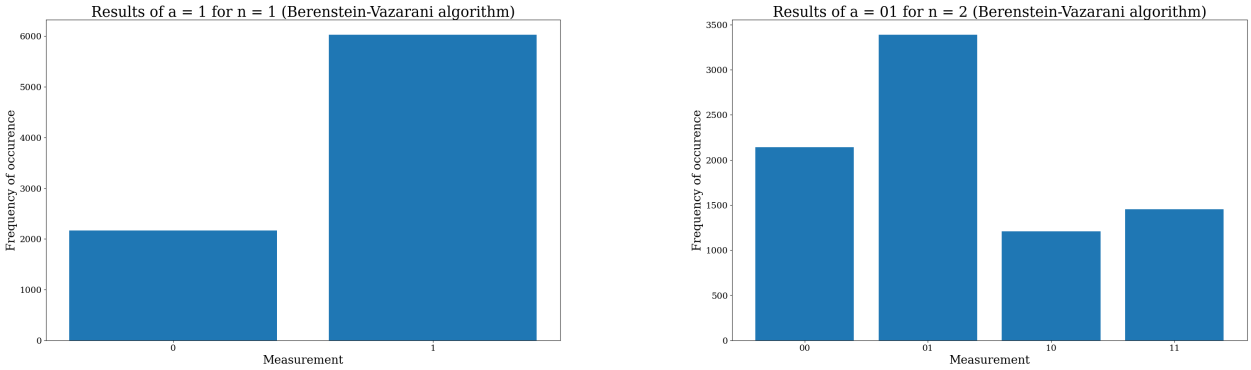
Given a function $f : \{0,1\}^n \rightarrow \{0,1\}$ *of the form* $f(x) = a.x + b$*, with* $a \in \{0,1\}^n$ *and* $b \in \{0,1\}$ *are unknown bit strings, obtain the value of* a*.*

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 (1). The quantum circuit for the BV algorithm is the same as that of the DJ algorithm (Fig. 1). The BV algorithm determines a to be the state of the n qubits measured at the end of the circuit.

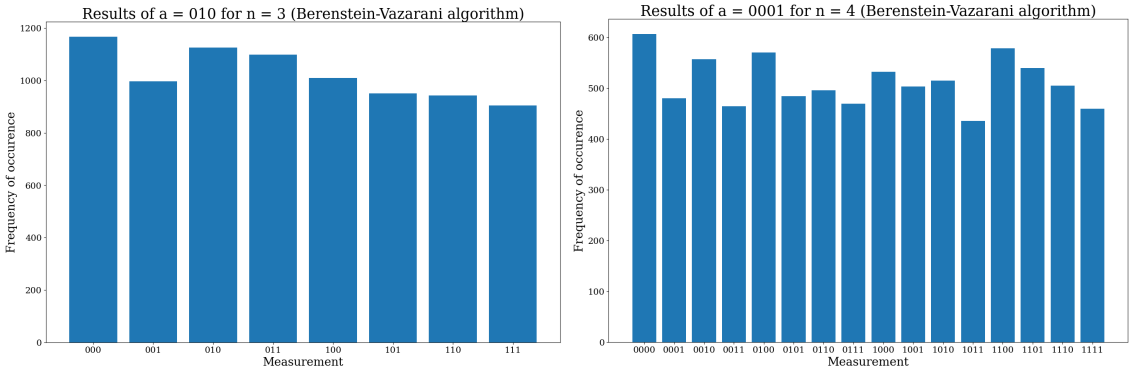
As mentioned in the Deutsch-Jozsa section, we limit our analysis to 5 qubits or less using traditional methods and then explore alternatives for larger values of n.

Now we discuss various aspects of the run time and reliability. The details are very similar to Deutsch-Jozsa since they share the same circuit. We present our results but for a more in depth analysis, see the Deutsch-Jozsa section.

- **Statistics of Results of Running on a Quantum Computer**: To demonstrate the effect the number of qubits has on the error rate, we ran the Berenstein-Vazirani circuit on IBM's London quantum computer. For each n, we select an a at random. We derive b classically, so we ignore it for now. Figure 9 shows the results. For reference, on a simulator with no noise, we would expect a to be measured with 100% probability. Note the results are similar to Deutsch-Jozsa. For $n = 1$, 1 is measured with about 75% probability. For $n = 2$, 01 is the majority. For both $n = 3$ and $n = 4$, there is so much noise that a is not the most frequent measurement. Again, to highlight how problematic this is, we select 10 different values of a and b at random for each n, run 8192 trials, and choose the most frequent measurement as the measured value. We then record how often they are correct. We present the results in Figure 10. As expected, the percent correct decreases as n grows larger. When $n = 1$, each trial is fairly likely to get it right, so overall the majority element is 100% accurate. The error rate increases rapidly because the measurement must be precise. This shows that it would be hard to design a system around this that requires a precise answer. The Berenstein-Vazirani circuit is identical to the Deutsch-Jozsa circuit so we don't discuss the reason for the increasing error rate, or the circuit size since this is the same as in Deutsch-Jozsa.



(a) $n = 1$ (b) $n = 2$



(c) $n = 3$ (d) $n = 4$

Figure 9: Berenstein-Vazirani algorithm: Results of 8192 trials of a constant function for various values of n.

- **Dependence of U_f on execution time** Again, to show the oracle function's impact on execution time, we select 100 input functions of $n = 4$ at random and measure the execution time. The results are presented in Figure 11. Note there is a little variance in execution time. The time is divided by the number of trials so it is the average execution time for a single measurement. This saw a vary similar dependence as the Qiskit simulator. As expected, the histogram is similar to that of Deutsch-Jozsa. Since they are the same circuit, we expected they have the same change in execution time for different oracles. They have a very similar median and Berenstein-Vazirani spreads a little further but not enough to be significant.
- **Scalability with n** In order to measure the scalability, we select 10 values for a and b at random. We then generate an input function and run the Berenstein-Vazirani circuit on it. We measure the time it takes to transpile and assemble the circuit which is done locally. Then we run the circuit on the Essex quantum computer for 8192 runs and divide the time Qiskit reports to us by the number of trials. This gives us an average execution time for a single run. We present the results in Figure 12. Again, the results are very similar to Deutsch-Josza since they are the same circuit. We see the compilation time grows exponentially and could quickly become a problem. The actual execution time scales much better. The 5 qubit circuit takes only 5% longer to run than the 2 qubit circuit and it is in the order of milliseconds. One interesting point is the execution times are nearly identical to the London quantum computer so there is no faster machine between the two.
- **Comparison with Simulator** We discuss this at greater length in the Deutsch-Jozsa section. Overall, the execution time is much faster on a quantum machine but the error rate leads to difficulties.
- **Modifications from Simulator to Quantum Computer** We discuss this at greater length in the Deutsch-Josza section since the code is nearly the same. Most of the changes in code for actually running the circuit had to do with passing around the `backend` object and the api token. Overall though, IBM has designed there API with the `backend` abstraction so your code can go from running on a simulator to a quantum computer with no change. We had to update the identity matrix bug and change the way we evaulated the answer by selecting the most frequent measurement.
- **Hacking the Oracle** Similar to dj, after finding out that our normal method of creating a unitary matrix from f, generating an operator, and building the circuit results in a circuit too large for any quantum computer for

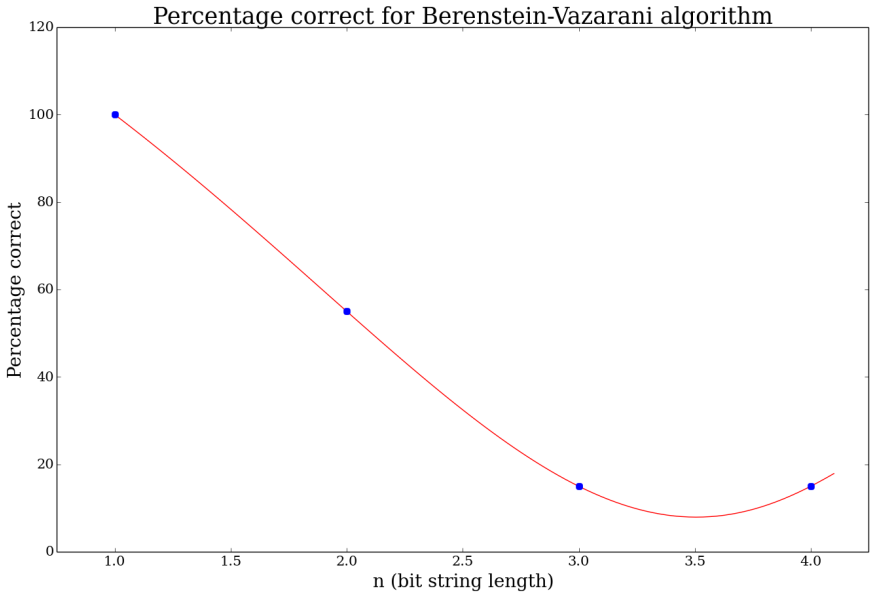


Figure 10: Berenstein-Vazirani Algorithm: Percentage correct for different values of n.

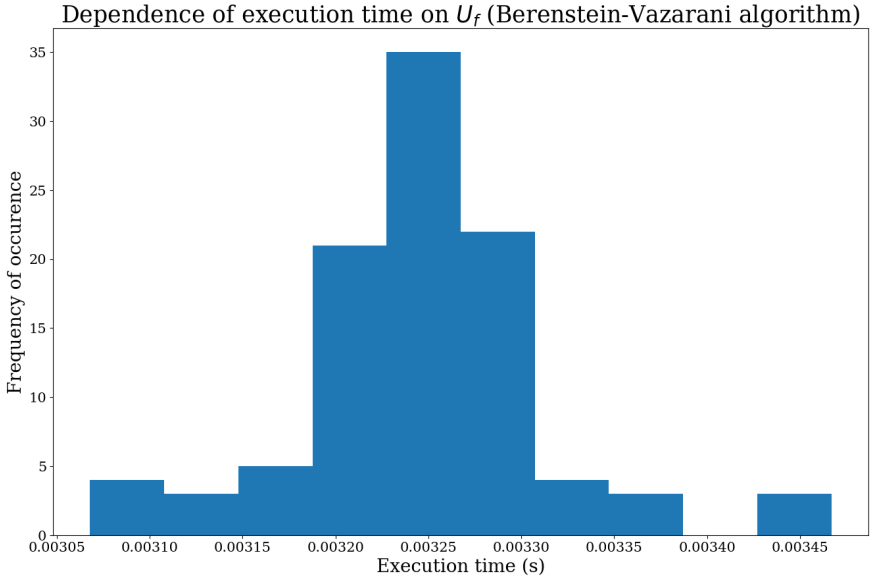
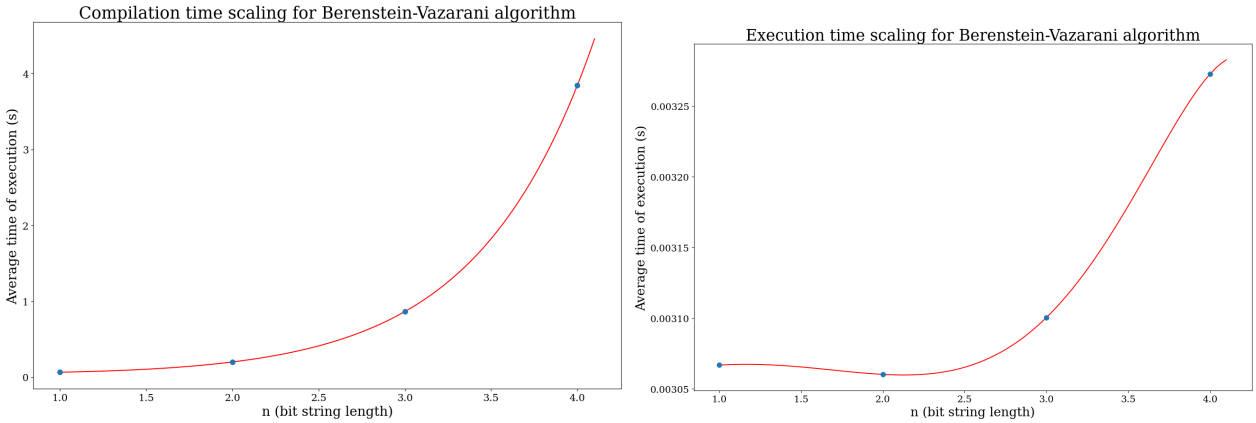


Figure 11: Berenstein-Vazirani Algorithm: Execution time for different U_f.

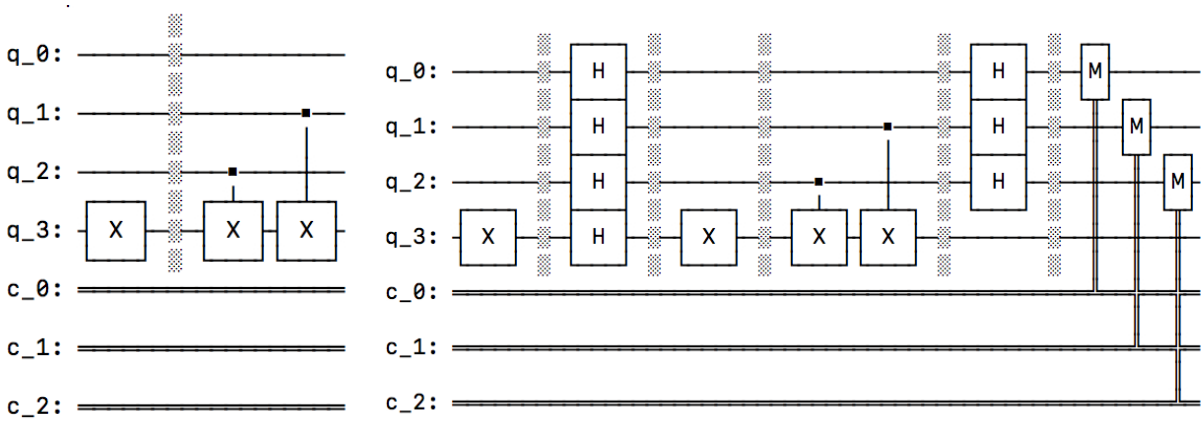
$n > 4$, we look to other methods. For Berenstein-Vazirani, there is a more satisfying solution. We can solve for a and b for a given f classically. Given a and b, we can generate the oracle gate that is much shorter than the method of building the unitary matrix, allowing us to benchmark performance and correctness. We admit this is a bit of hack since we are first solving for f classically and clearly inspecting it. However this method allows us to run for larger values. Also, the quantum problem assumes we're given an oracle. We are just extending this a little to say we are given the hardware specification of an oracle.
Custom oracles can be generated from a and b. If $b = 1$, a not gate is applied to the helper qubit wire. For each $a_i = 1$, the i^{th} wire has a `cnot` gate applied to the helper qubit wire. Figure 13 shows the custom gate and the gate plugged into the full circuit for $a = 110$ and $b = 1$. Note again, the reversal of the `cnot` gates to account for the endian-ness in qiskit.
Similar, to dj, to benchmark performance and correctness, we select 4 values of a and b at random for $n = 1$ through $n = 14$. We selected 4 this time because this is the maximum number of jobs that can be in the queue



(a) Assembly time for Berenstein-Vazirani.

(b) Execution time scaling for Berenstein-Vazirani.

Figure 12: Berenstein-Vazirani: Benchmarking.



(a) Bernestein-Vazirani custom gate.

(b) Complete circuit using custom gate.

Figure 13: Berenstein-Vazirani: Custom gates for a = 110 and b =1.

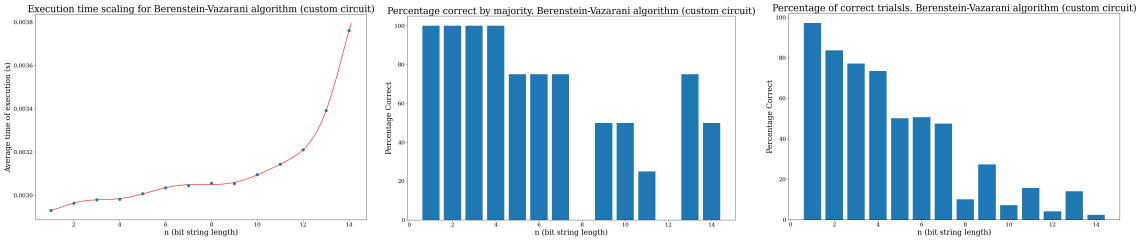
simultaneously. We then measure the run time, averaged out over the 8192 trials, the percentage correct if we used the majority, and the percentage of the 8192 trials that had the correct measurement. Figure 14 shows the results. We see that run time increases with n and starts to take an exponential pattern. However, scalability is still much better than a simulator. $n = 14$ takes only about 20% longer than $n = 1$. We see the error rate is much more manageable using this method than that of the unitary matrix. The percentage correct tends to decrease as n grows larger since longer circuits result in more errors. We note there are some where the percentage correct is 0. This is because we only ran 4 trials since we were using shared machines. Choosing functions at randoms means some will have longer circuits than others. Finally, figure 14c shows the average percentage of the 8192 trials that measured the correct value. This is a better indicator of noise. Note when $n = 1$, there are very few errors and nearly every measurement is correct. For $n = 14$, there is a lot of noise and many measurements are incorrect. As long as the noise is spread out over many values though, we can still get the correct value if the correct measurement is in the majority.

1.3 Simon's algorithm

First, we summarize the problem statement:

Given a function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ *such that* $f(x) = f(y)$ *iff.* $y \oplus x \in \{0, s\}$, *find the value of* s. *If no such string* s *exists, return* 0^n.

The classical implementation requires testing up to $2^{n-1} + 1$ inputs until two inputs with the same output



(a) Run time (b) Percentage correct (c) Percentage of correct trials

Figure 14: Berenstein-Vazirani: Custom oracle gates.

are found. If none are found, s is 0 and f is 1 to 1.

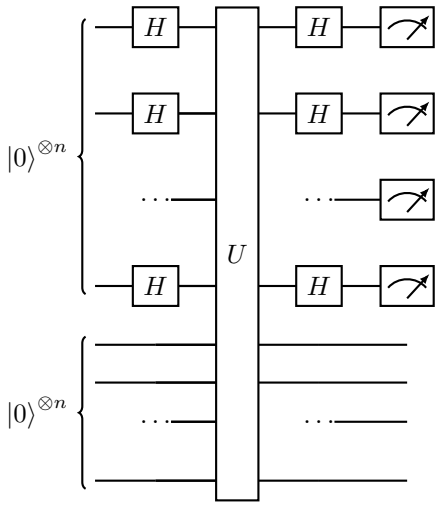


Figure 15: Simon algorithm

Simon's algorithm consists of a quantum circuit, which is shown in figure 15. The circuit involves n helper qubits, instead of just 1, as in the case of DJ and BV algorithms. The oracle U_f satisfies equation (1), except with the modification that b is now a bitstring of length n.

Each time Simon's circuit is run, the result is a y which satisfies $y \cdot s = 0$. If the circuit is measured n-1 times, and all values of y are independent, the system of equations can be solved with 2 values of s, one of them being 0. However, the n-1 values of y may not be all independent so it may need to be measured more times. Though it's possible to keep measuring dependent vectors, there is a 99% probability that $4 * m * (n - 1)$ measurements, where $m = 5$, will result in n-1 independent equations.

Now we discuss various aspects of our code design:

- **Backends used:** We tested two methods for creating oracles. For the first one (which does not work for $n > 2$, we used `ibmq_burlington` (which has 5 qubits), and for the second method, we used the `ibmq_16_melbourne` backend (which has 15 qubits). In order to compare our results with those from a simulator, we used the `ibmq_qasm_simulator` backend.
- **Creating oracle U_f by using the matrix representation- circuit depth issues**: First, we summarize a method which works for all input functions f. As we shall see, this method is problematic, and cannot be used if n values with $n > 2$, because of errors in the quantum hardware. Here is the pseudocode:
  1. Initialize U_f to be a $2^{2n} \times 2^{2n}$ zero matrix.
  2. For every possible bit strings x and b of length n, compute the bit string $y = b \oplus f(x)$. Since $x + b$ maps to $x + y$, we set the corresponding matrix element to 1 (here $+$ denotes string concatenation.)

We observed that for any given f, calculating the U_f matrix, and then creating a custom gate corresponding to it and transpiling the circuit is not feasible. This is because the circuit depth becomes too large after transpilation, resulting in an unacceptably high error rate in the quantum circuit. In order to obtain the average transpiled circuit depth, for $n = 1, 2, 3$ each, we created 10 random functions f, and created and transpiled the corresponding circuits for the backend `ibmq_16_melbourne`. The average circuit depth for these n values we obtained were as follows:

n	1	2	3
Average transpiled circuit depth	4	109	3092

Since the average transpiled circuit depth for $n = 1$ was 4, the error rate for the algorithm upon running on the `ibmq_16_melbourne` hardware was relatively low ($\approx 25\%$). However, for $n = 2$, the results of the measurement were effectively random- a circuit depth of about a hundred is too high for NISQs, with the error accumulating with every additional quantum gate required. For $n = 3$, since the circuit depth was in the thousands, with the job being rejected by IBM, with the following message: `Circuit runtime is greater than the device repetition rate`. (Even if the job had run, we can expect the result to be essentially random).

This was not an issue while working on the previous projects- since the quantum simulators we used were error-free, the circuit depth was not important from the point of view of ensuring correctness of the results.

All this calls for a different method in order to construct the U_f gate, so that after transpilation, the circuit depth is relatively small. This is what we discuss next.

- **An alternative: Creating the oracle directly in terms of default gates**: Ideally, given any function f, one should be able to obtain the oracle matrix, explicitly in terms of well-known one and two qubit gates, even before decomposing the matrix using a transpiler. In general, given any function f, it is always possible to obtain a U_f satisfying (1), by first constructing a classical reversible circuit using universal classical gates, and then obtaining the corresponding quantum version, as outlined in section 3.2.5 of Nielsen and Chuang [1].

  However, such a method requires additional number of ancilla or helper qubits. Since we have access to a maximum of 15 qubits (using the `ibmq_16_melbourne` backend at IBM), we instead use a method which allows us to work for functions with as large an n as possible. To that end, we developed an algorithm (inspired by ideas in [2]) in order to create an oracle directly in terms of `NOT`, `CNOT` and `SWAP` gates. Recall that for any given secret s, there exist multiple number of functions f_s which satisfy the condition required in Simon's algorithm. We created an algorithm (see 1) which given any s, randomly chooses a function f_s, and creates a corresponding oracle directly in terms of the usual gates. In other words, we do not create the U_f matrix. Such a procedure results in a circuit depth which is tremendously smaller than that obtained using the matrix construction technique, allowing us to execute Simon's circuit for n as large as 7, (since that corresonds to a total number of required qubits of 14, which just shy of the upper limit of 15 qubits available to us on the `ibmq_16_melbourne` backend.

Algorithm 1: Given a string s, obtaining an oracle directly as a combination of predefined gates

Input : A string s

Procedure: If n is the length of string s, create a circuit with n qubits in the first register, and with the second register consisting of n qubits as well. Do:

1. Copy the first register to the second register, i.e. $|x\rangle \otimes |0\rangle \rightarrow |x\rangle \otimes |x\rangle$, by appending `CNOT(i,n+i)`, for $i = 0, \ldots, n-1$ to the circuit.
2. If the string s is not all zeros, then there exists a least number $j < n$, such that $s_j \neq 0$. In order to execute $|x\rangle \otimes |x\rangle \rightarrow |x\rangle \otimes |x \oplus s\rangle$ if $x_j = 0$, do the following:
   (a) append `NOT(j)`.
   (b) for $i = 0, \ldots, n-1$, if $s_i \neq 0$, append `CNOT(j,n+i)`.
   (c) append `NOT(j)`.
3. Randomly permute and flip qubits in the second register. To that end
   (a) For a random number $k <^n C_2$, obtain k number of distinct combinations (i, j) for $i, j \in \{n, \ldots, 2n-1\}$. For each combination, append `SWAP(i,j)`.
   (b) For a random number $k < n$, choose (without replacement) k number of numbers from $n, \ldots, 2n-1$. For each chosen number j, append $NOT(j)$.

Output : An oracle operator corresponding to string s, consisting only of `CNOT`, `SWAP`, and `NOT` gates.

Some representative examples of the type of oracles we obtain from this algorithm are shown in figure 16. Notably, the circuit shown in 16d has a circuit depth of 9 for $n = 7$, compared to a circuit depth of millions, if one were to construct the U_f matrix first, convert it into a gate, and then transpile it.

For each value $n = 1, \ldots, 7$, we created ten random circuits corresponding to randomly chosen values of `s`, and transpiled those circuits for the `ibmq_16_melbourne` backend. A plot of the circuit depths is shown in figure 17.

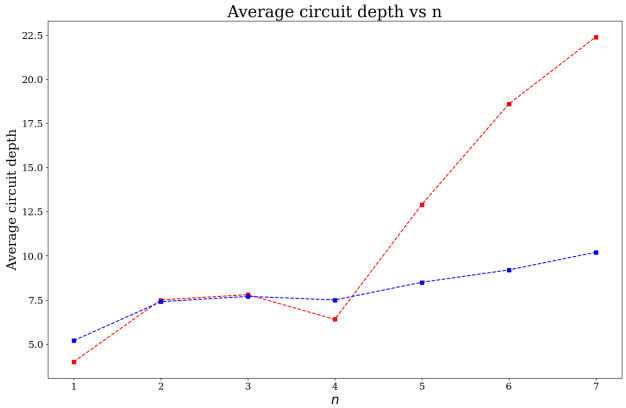
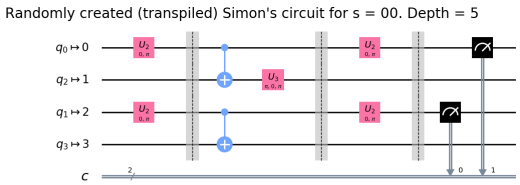


Figure 17: Average circuit depth obtained across different values of n, after following algorithm 1. The red curve is corresponds to the average transpiled circuit depth, while the blue curve denotes the untranspiled average circuit depth.

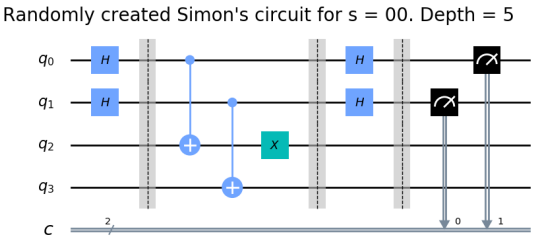
- **Measurement error mitigation:** Because of errors in the quantum hardware, generally, the measured output can differ from what is possible given the wavefunction before measurement. For example, even if the state of a qubit is $|1\rangle$, there is a non-zero (but small) probability that the hardware will report a measured state of 0. Such an error is called a measurement error. In order to reduce such errors, we use the measurement error mitigation procedure available in qiskit `qiskit.ignis`. Specifically, we use `qiskit.ignis.mitigation.measurement.complete_meas_cal` in order to obtain a measurement filter, which can be used to as a classical post-processing tool to mitigate measurement errors. As we will show below, such an error mitigation procedure reduces errors, and takes us closer to the results obtained from the simulator.
- **Results on the `ibmq_burlington` backend - comparison with simulator output**: We present the results obtained after using the algorithm 1 in order to create an oracle. Since the number of qubits available on this backend is 5, the only possible values of n for Simon's algorithm are $n = 1$ and 2. We compare three outputs- the values of y measured by the quantum hardware, the corresponding error mitigated counts, and the counts obtained by running the same circuit on the simulator.

  For example, for $n = 1$, we create a circuit corresponding to $s = 1$ (see figures 18a and 18b). For each experiment, we run this circuit 20 times. We executed 50 such experiments, on the `ibmq_burlington` and the `ibmq_qasm_simulator` simulators. An error-free output would only consist of $y = 0$, since this is the only allowed value which satisfies $y.s = 0$. However, we found that the quantum hardware also measured 1 about a quarter of the times on average, although the simulator was completely error-free (see figure 18d).

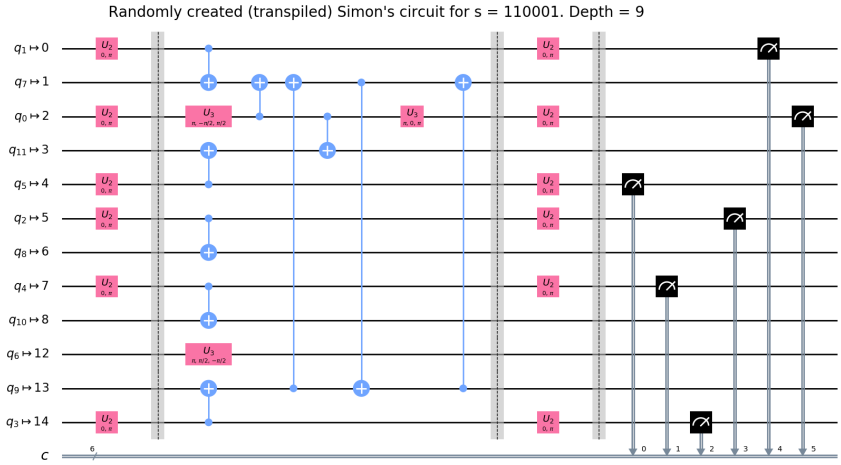
  A similar comparison of the obtained results for $n = 2$, and $s = 10$ is shown in figure 19. The error rate is approximately 23%.



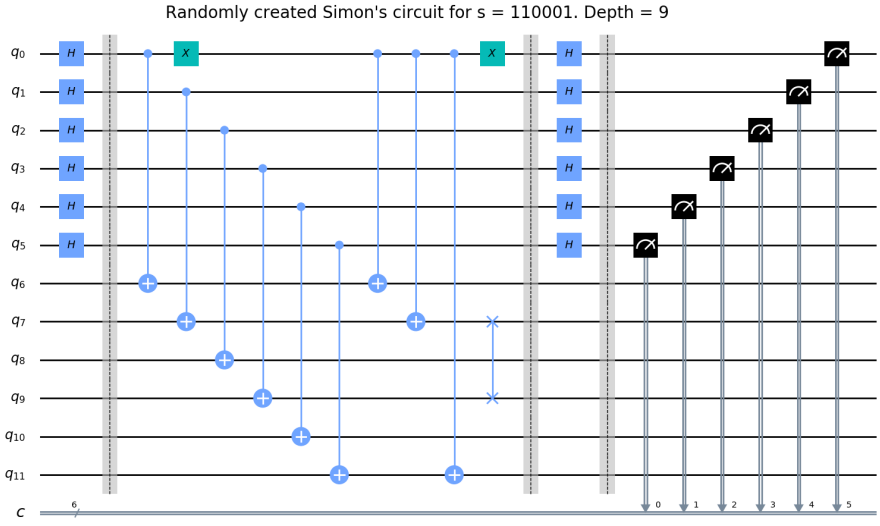
(a) Randomly generated Simon's circuit for $n = 2$.



(b) The circuit from figure 16a after transpilation.



(c) A randomly generated Simon's circuit for $n = 7$.



(d) The circuit from figure 16c after transpilation.

Figure 16: Given any string s, we can create a corresponding Simon's circuit following algorithm 1, in order to create a circuit of small depth. This allows us to use the quantum computer at IBM

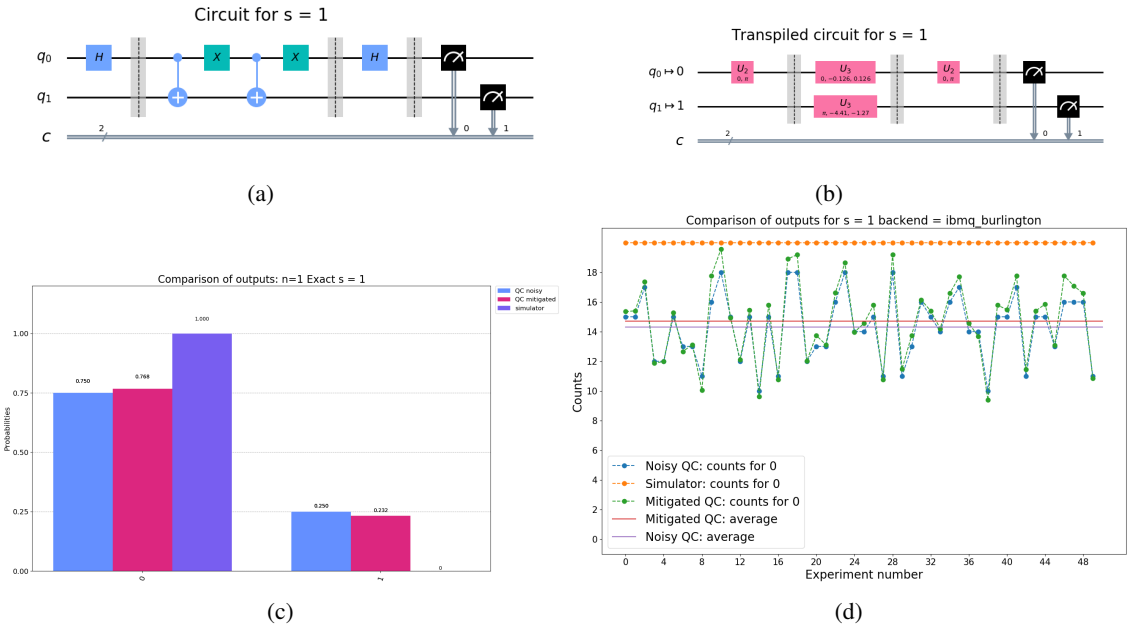
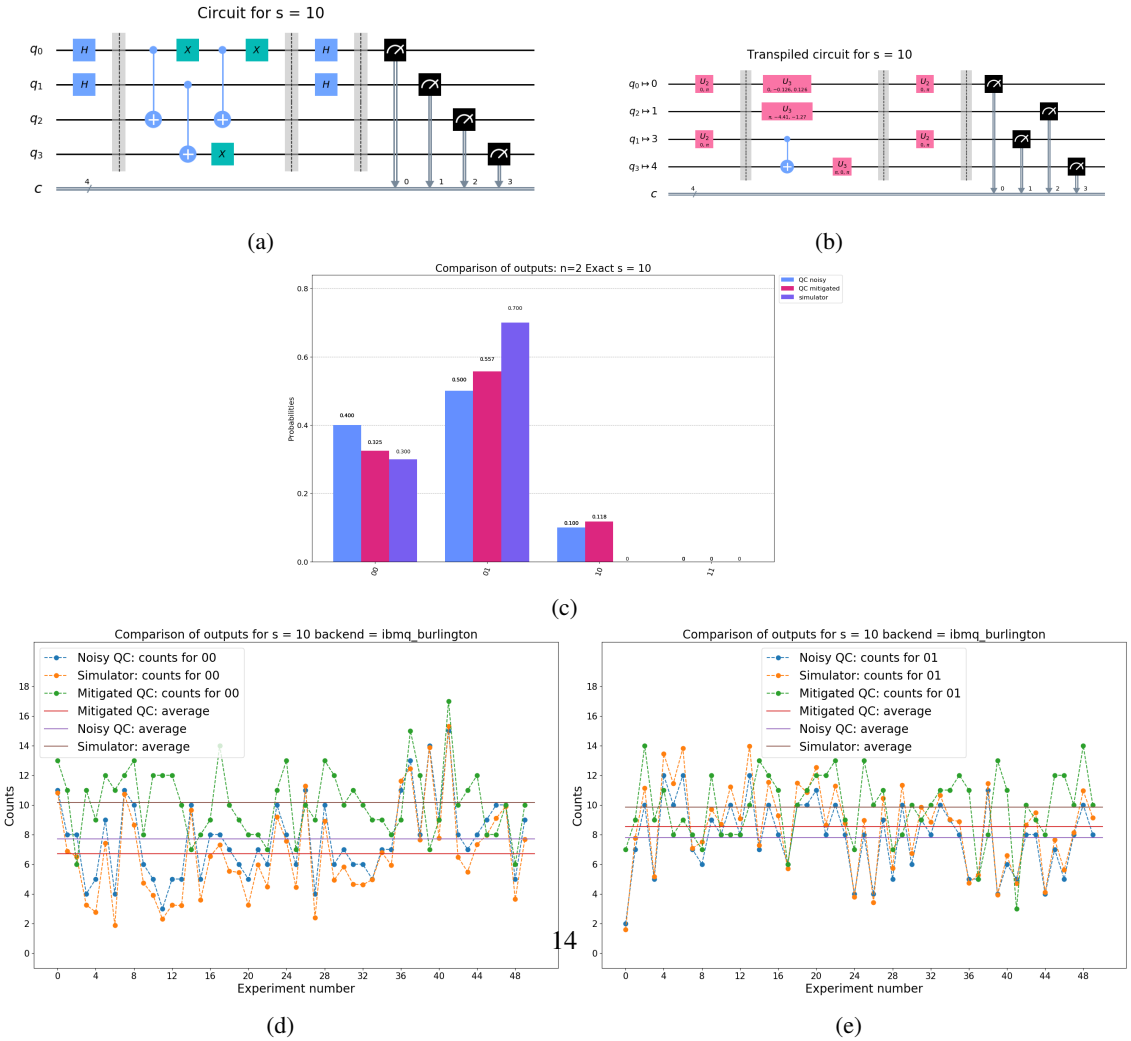
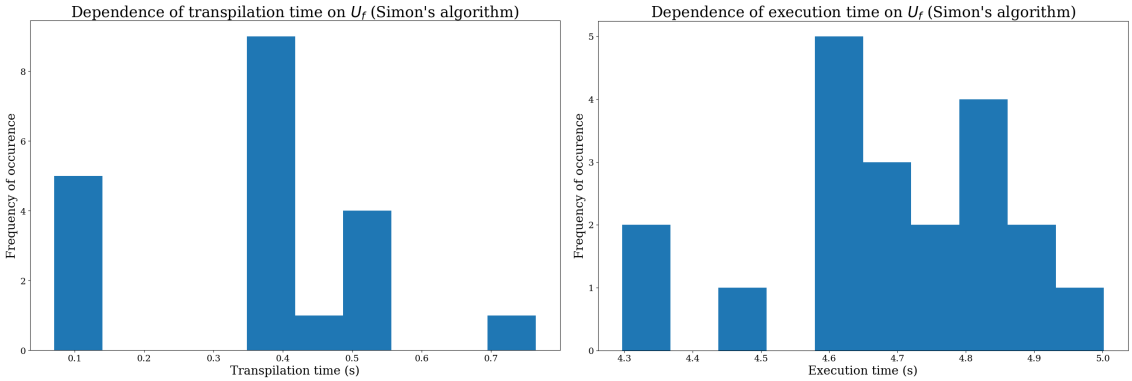


Figure 18: Comparison of results obtained by `ibmq_burlington` and `ibmq_qasm_simulator` for $n = 1$. (a) Circuit used for analysis on `ibmq_burlington` for $n = 1$. (b) The circuit from figure 18a after transpilation. (c) A typical histogram of measured y values. The simulator's output is error-free, but the quantum hardware measures values of y which do not satisfy $y.s = 0$ a small fraction of the times. (d) Since $s = 1$, the measured value of y should ideally be 0. Although the simulator always measures 0 every time, the quantum hardware measures 1 on average, 15 out of 20 times, i.e. with an error rate of about 24%.



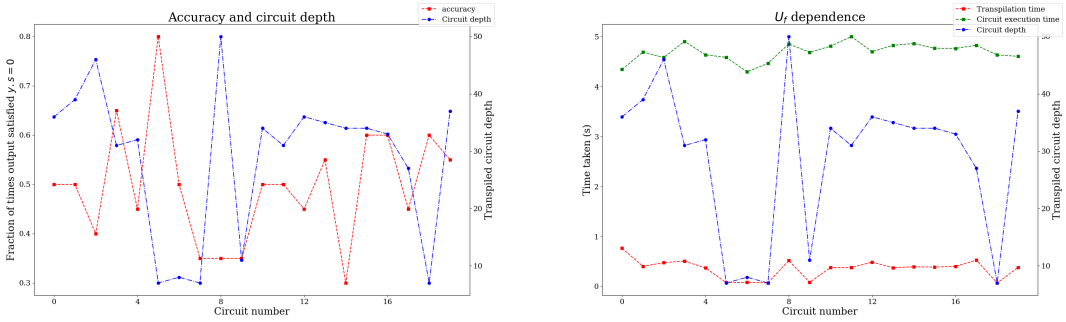
- **Variation with U_f**: We found that the execution time is almost the same across different values of U_f for a chosen value of n. In order to study the variance of execution time with U_f, for $n = 7$, we obtained twenty randomly chosen bitstrings s. Using algorithm 1, we obtained the corresponding oracles and prepared the corresponding Simon's circuits. Thus, we executed twenty experiments (with `num_shots` set to 20 for each one), and obtained the execution time for each experiment using `result.time_taken`. Histograms for the obtained transpilation times and circuit execution times is shown in figure 20



(a) (b)

Figure 20: For $n = 7$ and the `ibmq_16_melbourne` backend, (a) a histogram of obtained circuit transpilation times, and (b) a histogram of execution times for twenty randomly chosen values of s.

- **Accuracy of outputs vs circuit depth**: For the same set of experiments as in 20, we compared the accuracy of the output obtained by the quantum computer, and the corresponding circuit depth. Clearly, the accuracy of the quantum hardware is higher, if the circuit depth is lower. This is expected- the larger the circuit depth, the higher the chance of decoherence. We present the results in figure 21



(a) (b)

Figure 21: For the same set of experiments as in figure 20, we compare (a) the accuracy of the quantum hardware and cirucit depth, and (b) circuit execution time and circuit depth. It can be concluded that a higher circuit depth results in a larger execution time, and a smaller accuracy rate.

- **Scalability:** We observed (see figure 22 that the execution time for Simon's algorithm did not depend on the value of n. This is in stark contrast with the simulator, for which the execution time scales exponentially with increasing n. Clearly, nature takes the same amount of time for calculating matrix products, independent of the sizes of the matrices involved- a remarkable fact!

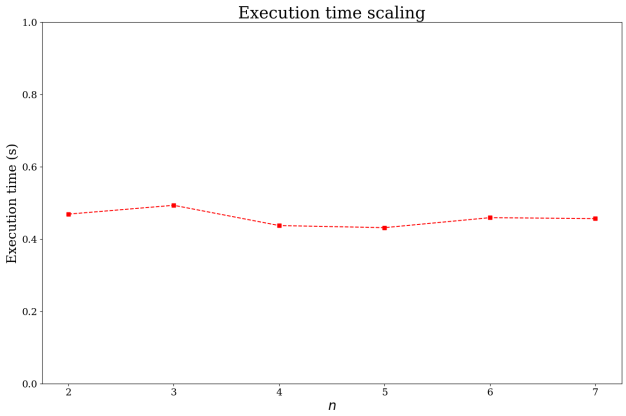


Figure 22: The average execution time stays roughly the same for different values of n. For n ranging from 2 to 7, we created 10 random Simon's circuits using algorithm 1. The average execution times obtained on the `ibmq_16_melbourne` backend are plotted. Note: this does not include transpilation time.

1.4 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

$$G := -H^{\otimes n} Z_0 H^{\otimes n} Z_f \left|x\right\rangle, \tag{2}$$

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

$$Z_f \left|x\right\rangle = (-1)^{f(x)} \left|x\right\rangle, \tag{3}$$

$$\text{and } Z_0 \left|x\right\rangle = \begin{cases} -\left|x\right\rangle & \text{, if } \left|x\right\rangle = 0^n \\ \left|x\right\rangle & \text{, if } \left|x\right\rangle \neq 0^n. \end{cases} \tag{4}$$

Grover's operator, G, is shown in Fig. 23a. In order to maximize the probability of success, the operator G is applied k number of times, with k being

$$\begin{aligned} k &= CI\left(\frac{\pi}{4\theta} - \frac{1}{2}\right) \\ \text{with } \theta &:= \arcsin\frac{a}{N}. \end{aligned} \tag{5}$$

Here, $CI(y)$ denotes the integer closest to y. The complete circuit is shown in figure 23b. The probability of success (i.e. the probability of the final measurement resulting in a bitstring satisfying $f(x) = 1$) is given by

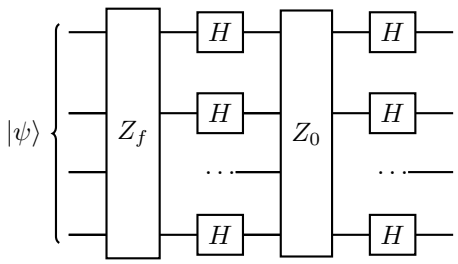
$$P_{success} = \sin^2((2k+1)\theta). \tag{6}$$

k (5) number of Grover's iterations G results in measurement of state $|x\rangle$ which satisfies $f(x) = 1$ with high probability, on the order of $95 - 99\%$.

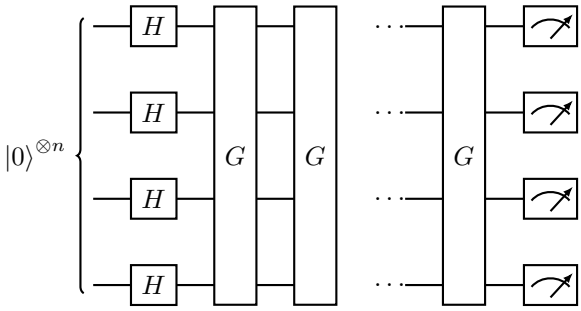
Now we discuss various aspects of the run time and reliability:

We ran tests on the IBMQ London computer with n=1 to n=4. There were problems with the built in transpiler and assembler. On the highest optimization level, the circuit depth and size were too large when running with more than 4 qubits, (> 3000). The library would throw an error at runtime that the circuit runtime was greater than the backend's refresh rate. As a result, we limit our analysis to 4 qubits and under for traditional methods and then consider using alternative methods like building custom gates.

- **Statistics of Results of Running on a Quantum Computer**: To demonstrate the effect varying the number of qubits has on the error rate, we ran the Grover circuit on IBM's London quantum computer, which is a 5

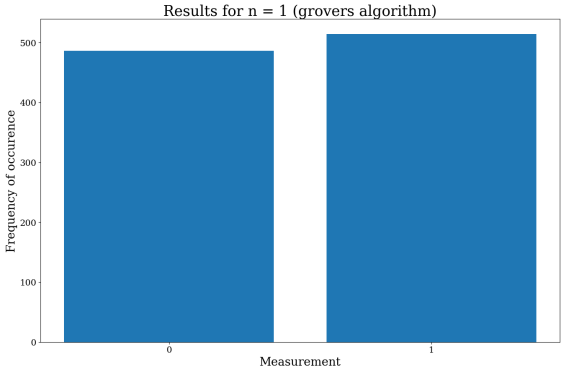


(a) Grover's iteration G.

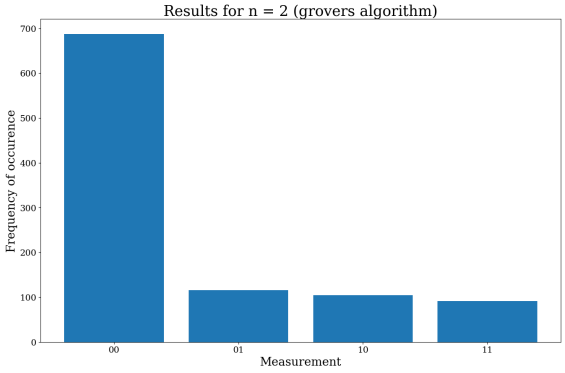


(b) Grover's circuit: Grover's operator G occures $k = \mathcal{O}(\sqrt{N})$ times.

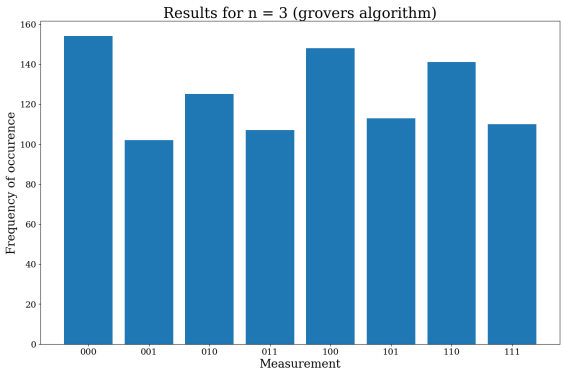
Figure 23: Grover's search algorithm



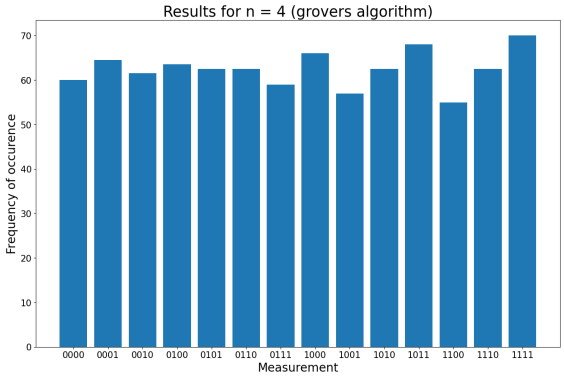
(a) $n = 1$



(b) $n = 2$



(c) $n = 3$



(d) $n = 4$

Figure 24: Grover algorithm: Results of 1000 trials for various values of n.

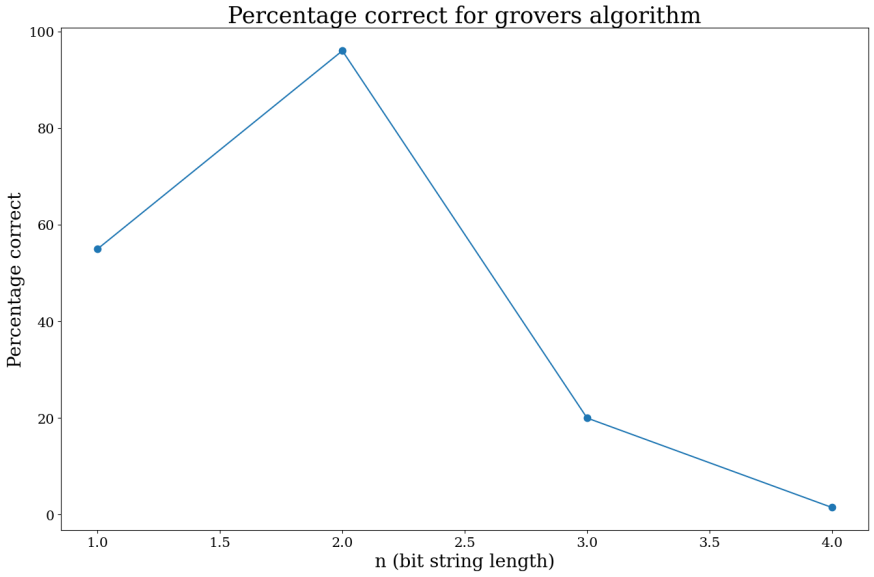


Figure 25: Grover's Algorithm: Percentage correct for different values of n.

qubit quantum computer. Above 4 qubits for Grover's circuit with the regular transpilation and assembling method on based on a matrix representation of the circuit: the quantum computer timed out because the circuit was too big. We did not run it on a larger machine, because as mentioned previously, it would throw an error for larger number of qubits. We chose a random function for Grover's algorithm with a=1. We had approximately matching theoretical error for the n=1 and n=2 cases but the error got considerably worse after $n > 2$. The error for n=3 was approximately 20% and for $n = 4$ was 1.5%. Another error we found was a bug in the transpiler, which would throw an error if we tried to add a gate that was built from the identity matrix. As a workaround, we simply do not add the gate if it's built from the identity matrix, since the identity matrix does not change the circuit.

Figure 24 shows the result of running 1000 trials of the grover's circuit on the IBM London quantum computer for random functions with $a = 1$, the number of items in the domain of x such that $f(x) = 1$, for $n = 1$ through $n = 4$. Note how the noise increases with n. For reference from our earlier report, on a simulator, the theoretical probability matches the simulated probability. For $n = 1$, we measure 0 with close to 50% probability, which is as expected because you couldn't expect to do better than flipping a coin. For $n = 2$, the probability of measuring x such that $f(x) = 1$ also matches the theoretical at 95%. However as mentioned earlier when n is not ideal, both these cases are still manageable since a simple majority voting algorithm would still select the right answer. For $n = 3$ and $n = 4$, the probability of measuring an x s.t. $f(X) = 1$ approaches $\frac{1}{2^n}$. This means there is so much noise, the results are close to random. Clearly for $n > 2$, different trials would result in different results. We should point out that some Grover's functions could be more complex, so these results would be amplified for different U_f's with different number of a values. To highlight how troublesome this error rate is, we run an experiment where we select 10 functions at random and select the majority element as the final answer. Figure 25 shows the percentage of times that it was correct. For $n = 1$, this method matches the theoretical based on the previous report. However, as n grows larger, this method is increasingly unreliable because it is incorrect a larger percentage of the time. Interestingly enough, we would expect a noisy circuit to be correct a greater portion of the time than it was in practice. Assuming a truly random circuit, a random function has a probability of $\frac{1}{2^n}$ of being right. This is because only the measurement that is x s.t $f(X) = 1$ when $a = 1$ would be correct. At $n = 4$ though, the correct value is measured with nearly negligible probability. 0^n starts to get measured very frequently. Perhaps when the IBM computers have a large error, they tend to report 0.

In order to examine why the error grows for larger values of n, Qiskit provides two useful measurements. Circuit depth is the length, or number of gates on the critical path and size is the total number of gates. Quantum gates have much higher error rates than classical gates. The more gates there are, and the more gates a calculation passes through, results in a higher error rate overall. Figure 25 shows the results of these values for $n = 1$ through $n = 4$ for $f(x) = 1$ at the highest optimization level. The number of gates grows exponentially which explains why the total error rate grows so rapidly. This indicates a need for a better decomposition algorithm for transpilation to accepted 2qubit gates on the given quantum hardware.

- **Dependence of U_f on execution time** In order to measure the oracle function's impact on execution time, we select 100 input functions with $n = 3$ at random and measure the execution time. The results are presented in Figure 27. Note there is a little variance in execution time. The time is divided by the number of trials so it is the average execution time for a single measurement. Qiskit's simulator saw a range that was about 3% of the median value where as PyQuil saw a much larger range. The quantum computer execution time had a median of 4.9 s and the values were within 0.1 s, except for a few outliers, so the variance is similar to the Qiskit quantum simulator. This is expected especially considering the transpiler tries to optimize the circuit so some circuits will be longer than others.

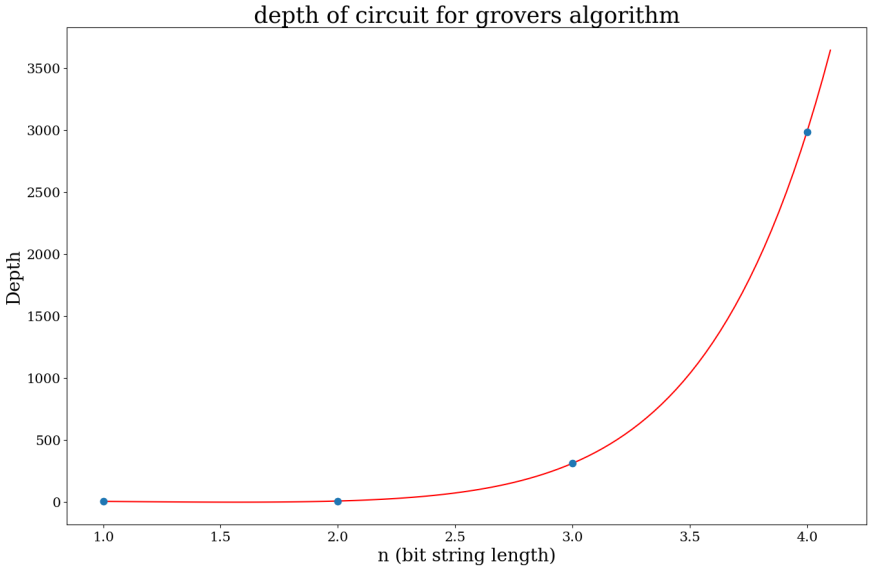


Figure 26: Grover algorithm: Circuit size and depth for $f(x) = 1$

- **Scalability with n** In order to measure the scalability, we select 10 functions at random and measure the time. There are two portions the consider. The first is the assembly time, which is done locally. This is measured using a timer. The next component is the actual execution time on the quantum computer. In order to do this we count on Qiskit to report the `time_taken` and we divide by the number of trials to measure the average time of a single run. We take the average this for each f and present the results in Figure 28. Note the assembly time on the left increases exponentially with n. Since a classical computer is compiling a quantum circuit, there are scaling issues. On the quantum computer scales better though, the actual run time of the circuit scales much better. The average execution time for $n = 4$ is much larger than $n = 1$ because the circuit is much larger, roughly 2 orders of magnitude greater. 26 demonstrates how the size of the circuit grows exponentially with n.
- **Comparison with Simulator** The largest difference between a quantum computer and a quantum simulator is our results are no longer deterministic. Due to error rates in gates and decoherence, we measure with noise so we must account for that with mitigation or doctoring circuits to be easy to transpile. Unfortunately, the noise becomes quite large for $n > 2$, and it becomes difficult to get the correct answer. On the quantum simulator, we can run arbitrarily large circuits up to roughly n=10 before the exponential computation time becomes a problem. On a quantum computer, the hardware must be taken into account and some circuits may be too lengthy for the computer. The simulator has an exponential execution time so it scales very poorly. The quantum computer appears to scale better with n. However, assembly time is still exponential and the length of the critical path is exponential as well. However since gates are fast, the impact on the total run time is mitigated, at least for small values of n.
- **Modifications from Simulator to Quantum Computer** Very little code needs to be changed to go from a simulator to a really quantum computer. IBM designed it's backend abstraction so a simulator or a remote machine can be used. 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. We did need to make some changes to manually transpile the circuits we created such that we could pass in the optimization level and benchmark the circuit size and

assembly time. However, these are not required changes.
We also had to deal with a bug based on the identity matrix. When running on a quantum computer, the machine would throw an error at run time if the custom gate was built from the identity matrix. This was a little frustrating because the simulator had no such problem. In an ideal world, we would want the two to throw the same errors. As a result, we have to check if the oracle is built from the identity matrix, and if so, we don't add the gate. This is technically correct since the identity matrix doesn't affect the circuit and its a large optimization, but some could argue it violates the oracle principle since you must inspect it.
In practical terms though, changes needed to be made to account for the noise. Whereas on the simulator, we could make simple decisions based on the deterministic measurements, on the quantum computer we needed to run numerous trials, and find the majority element in the results to decide what was the measured value. We had to find the majority value for Grover's algorithm on the simulator too, but the proportions for each result were deterministic. Also, to account for the `run` method being asynchronous, we need to continuously poll the backend to get the status of the job.

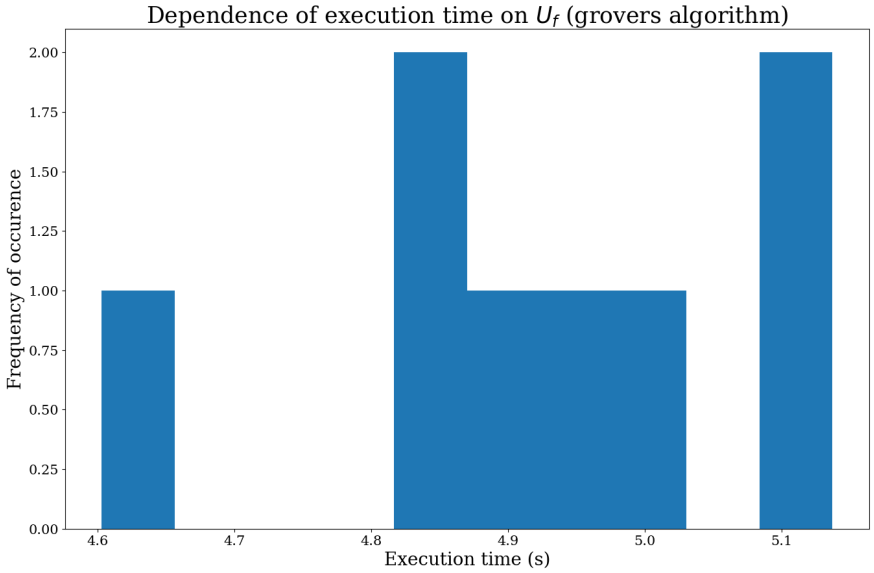
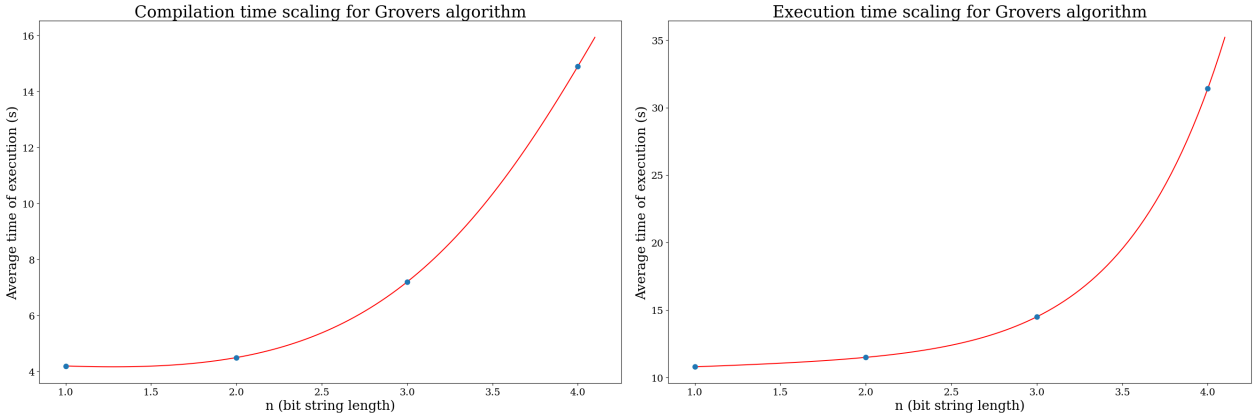


Figure 27: Grover: Execution time for different U_f



(a) Assembly time for Grover.

(b) Execution time scaling for Grover.

Figure 28: Grover: Benchmarking.

2 Evaluation and experience

Most of the assignment prompts have been answered in the sections above. We briefly summarize some of our conclusions below:

1. The quantum hardware is riddled with a significant error issues. Unless the circuit depth is small (≈ 10) the measurement errors are significant, creating a seemingly random output.
2. Measurement error mitigation can be used as a post-processing tool, in order to reduce the impact of measurement errors (see for example, our implementation for Simon's algorithm).
3. The oracle operators are essentially $2^m \times 2^m$ permutation matrices (with m being the number of qubits required). Transpilation of even such simple matrices is not efficient- even with $m = 4$, the transpiled circuit has a depth in the thousands!
4. For these reasons, creating an oracle gate by first obtaining a matrix representation, converting it to a gate, and transpiling, is not feasible. The circuit depth becomes too large for $m > 3$, and IBM rejects such jobs.
5. For three of the four algorithms, we were able to devise and implement an alternative to the matrix procedure- directly expressing the oracle in terms of `NOT`, `CNOT` and `SWAP` gates directly, without obtaining the matrix representation. The benefit of these methods is that they allow us to create circuits of sufficiently small depth, enabling us to run algorithms for the highest allowed values of n supported by the `ibmq_16_melbourne` backend.
6. For BV, given any function f, we devised a procedure for automatically creating the corresponding oracle so that it has a short circuit depth. This allowed us to go all the way up to a bit-string length of $n = 14$.
7. For DJ's algorithm, we devised a procedure which works for a large subset (but not all) of the allowed functions. We can randomly create balanced/constant functions in an automated fashion, with small circuit depths, allowing us to go up to $n = 14$.
8. For Simon's algorithm, we devised an algorithm for creating an oracle directly, given any value of s. This enabled us to go run circuits for a maximum n value of 7, which corresponds to a total of 14 qubits.
9. Although we identified a similar method for Grover's algorithm, we were unable to execute it on the hardware, because of a lack of time.
10. The execution time scaling is essentially constant- we observed it to remain roughly the same even with varying n values. This is in stark contrast with the exponential scaling observed in simulators.
11. There isn't a significant variance in the execution time with varying U_f.
12. The simulator output had a success rate of a hundred percent. Because of all the factors stated above, we had to modify our approach, and our codes significantly, in order to get a reasonable output from the actual quantum hardware.

References

[1] Nielsen, M.A. and Chuang, I., 2002. Quantum computation and quantum information.

[2] Qiskit textbook-Simon's algorithm.