THE CLASSICAL BOIS

QUANTUM PROGRAMMING

Madhav Sankar Krishnakumar madhavsankar@ucla.edu

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

February 17, 2022

Contents

1 Design and Evaluation **3**
1.1 Simon's algorithm . . . 3
1.1.1 Problem Statement: . . . 3
1.1.2 Implementation of U_f: . . . 4
1.1.3 Design considerations: . . . 4
1.1.4 Code readability: . . . 4
1.1.5 Parametrizing the solution in n . . . 5
1.1.6 Testing . . . 5
1.1.7 Variation of circuit Time with n . . . 5
1.1.8 Variation of Histogram curves for different parts of circuit . . . 6
1.1.9 Variation of number of iterations used with n . . . 7
1.2 Grover's search algorithm . . . 7
1.2.1 Implementation of U_f, i.e. G . . . 8
1.2.2 Parametrizing the solution in n . . . 9
1.2.3 Code Reusability . . . 9
1.2.4 Code testing . . . 9
1.2.5 Dependence of execution time on U_f . . . 10
1.2.6 Scalability . . . 13
1.2.7 Comprehensive evaluation/testing . . . 15
1.2.8 More optimized circuit construction . . . 15
1.2.9 Code well designed for improved usability or ease of understanding . . . 15
1.3 Deutsch-Josza algorithm . . . 15
1.3.1 Implementation of U_f: . . . 16
1.3.2 Code readability: . . . 16
1.3.3 Parametrizing the solution in n: . . . 17
1.3.4 Code testing: . . . 17
1.3.5 Dependence of execution time on U_f: . . . 17
1.3.6 Scalability: . . . 18
1.3.7 Comprehensive evaluation/testing: . . . 19
1.3.8 More optimized circuit construction: . . . 20
1.3.9 Code well designed for improved usability or ease of understanding: . . . 20
1.4 Bernstein-Vazirani algorithm . . . 20
1.4.1 Implementation of U_f . . . 20
1.4.2 Code readability: . . . 20
1.4.3 Parametrizing the solution in n . . . 20
1.4.4 Code testing . . . 20
1.4.5 Similarities in our codes for the BV and DJ implementations . . . 21
1.4.6 Dependence of execution time on U_f . . . 21
1.4.7 Scalability . . . 22
1.4.8 Comprehensive evaluation/testing . . . 22
1.4.9 More optimized circuit construction . . . 22
1.4.10 Code well designed for improved usability or ease of understanding . . . 23

2 Outline: The Language & Documentation **23**

ABSTRACT

We use Cirq to simulate four popular quantum algorithms (Deutsch-Jozsa, Bernstein-Vazirani, Simon's and Grover's search algorithms) on a classical computer. For each of the algorithms, we benchmark the performance of our simulations, 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. Finally, we provide some comments about Cirq and its documentation.

1 Design and Evaluation

1.1 Simon's algorithm

1.1.1 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.

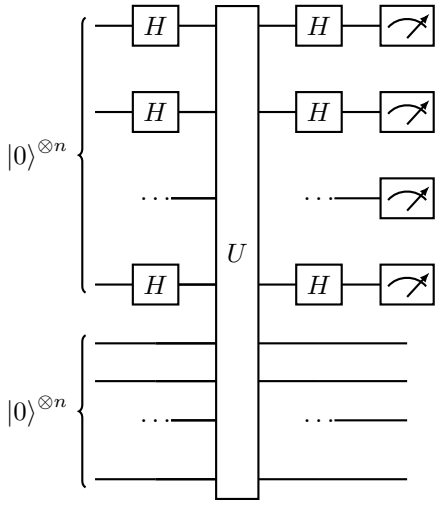


Figure 1: Simon algorithm quantum circuit

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 $y \cdot s = 0$.

1.1.2 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.

1.1.3 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 cirq 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.
- 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.

1.1.4 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.
- `class Oracle(cirq.Gate)`: Creates the actual quantum gate by taking the matrix U_f generated before, which is then fed into our circuit.

- `runMainCircuit(n,verbose = True)`: 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.
- `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).

1.1.5 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.

1.1.6 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 100 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 $n = 5$. Following are the results:

1.1.7 Variation of circuit Time with n

:

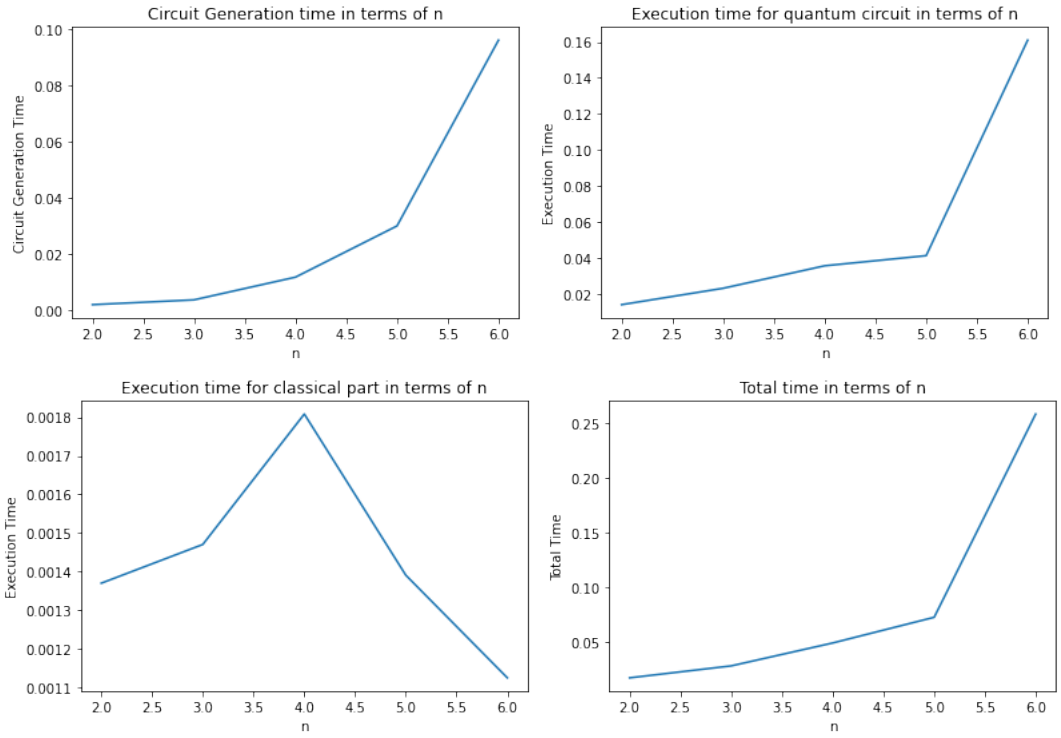


Figure 2: Execution Times for different parts of circuit

- We can see clearly that matrix generation and quantum circuit execution time rises exponentially 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.
- Interestingly, the SAT solver time does not depend much on n, and we cant observe any specific trend with n. The reason is simple, first we observe that scales of times taken by SAT solver to produce any solution is much much lesser than quantum circuit times. Hence any variation in times taken might be due to the current notebook compiler load, since such times become comparable to the time taken by SAT solver to solve the system of equations. For very large n, we might see the time taken to increase, but for small values of n, there is no specific trend.
- Lastly, the total time taken as we can see is exponential with n mainly due to quantum circuit times.

1.1.8 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 = 5 across 100 iterations.

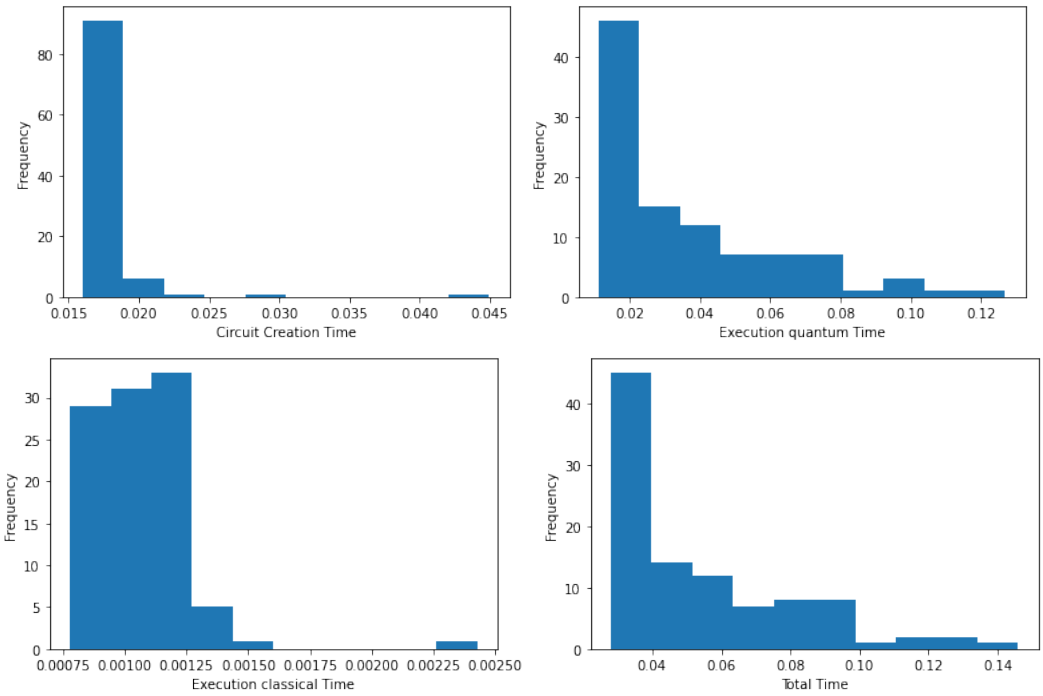


Figure 3: Histograms for circuit times for 100 iterations for n = 5

- 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 0.015-0.02.
- 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 0.00075-0.00125.
- 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. 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.

1.1.9 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. We ran for each case of n for 100 iteration and averaged out. As we can see, the

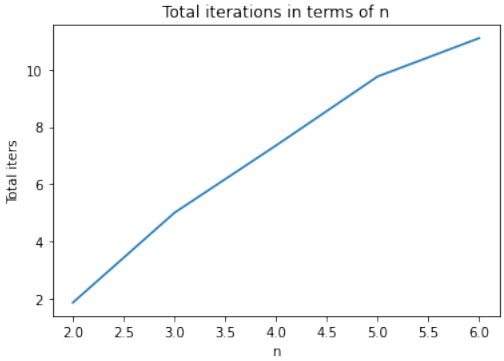


Figure 4: Variation of number of iterations with n

curve is linear, but is tapering in end. The major reason behind this is that there are 2 opposing product multipliers while computing number of iterations. At one hand, the number of iterations is directly proportional to $n-1$ (number of independent equations needed), which is increasing with n, but at other end, the number of iterations to generate n-1 linearly independent bit strings decrease as we increase n. This is mainly due to fact that probability to get (n-1) independent bit strings among 2^n bit strings increases as n increases and hence number of iterations to get (n-1) ranked matrix is less.

1.2 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:

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

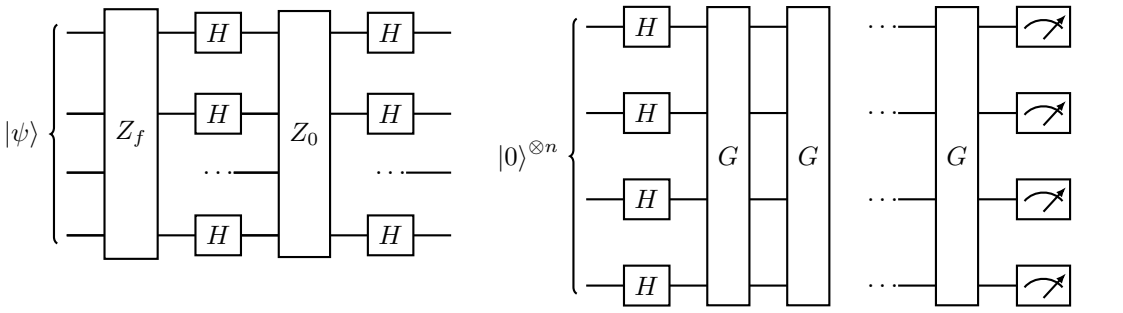
$$\text{and } Z_0 \left|x\right\rangle = \begin{cases} -\left|x\right\rangle & \text{, if } \left|x\right\rangle = 0^n \\ \left|x\right\rangle & \text{, if } \left|x\right\rangle \neq 0^n. \end{cases} \tag{3}$$

Grover's operator, G, is shown in Fig. 5a. 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 5b. 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 5: Grover's search algorithm

Now we discuss various aspects of our code design:

1.2.1 Implementation of U_f, i.e. G

To implement G, we need to implement Z_f and Z_0
Implement Z_0: 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. Apply X gate to all qubits. So 0*n becomes 1*n and every other state is not 1*n. Apply CZ gate now to make 1*n negative. 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.

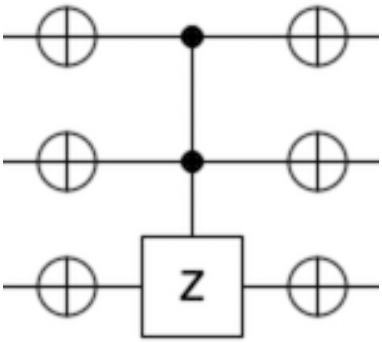


Figure 6: Grover Z0 implementation for n = 3.

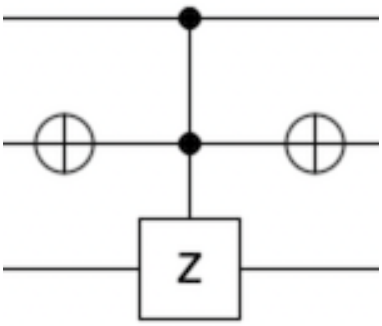


Figure 7: Grover Zf implementation for n = 3 and f(101) = 1.

1.2.2 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.

1.2.3 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(qubits)`: Creates Z_0 gate using the method described above.
3. `createZf(qubits, 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 30 times and plot the results using a histogram. We generate 2 times - time needed to create circuit and time required for execution.
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.

1.2.4 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 very well, as can be seen in the table below:

n	N	a	Theoretical success rate	Experimental success rate
1	2	1	50%	46%
2	4	1	100%	100%
2	4	2	50%	48%
2	4	3	1.5%	0%
3	8	2	100%	100%
4	16	1	96%	49%

Table 1: Grover's Algorithm: Numerical rate of success matches with the theoretical value

1.2.5 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 $a = 1$ 100 times, each time generating a random function. The fastest Z_f was close to twice as fast as the slowest. We divided the time into two: time required to create the circuit and the simulation run time. (see 8, 9, 10).

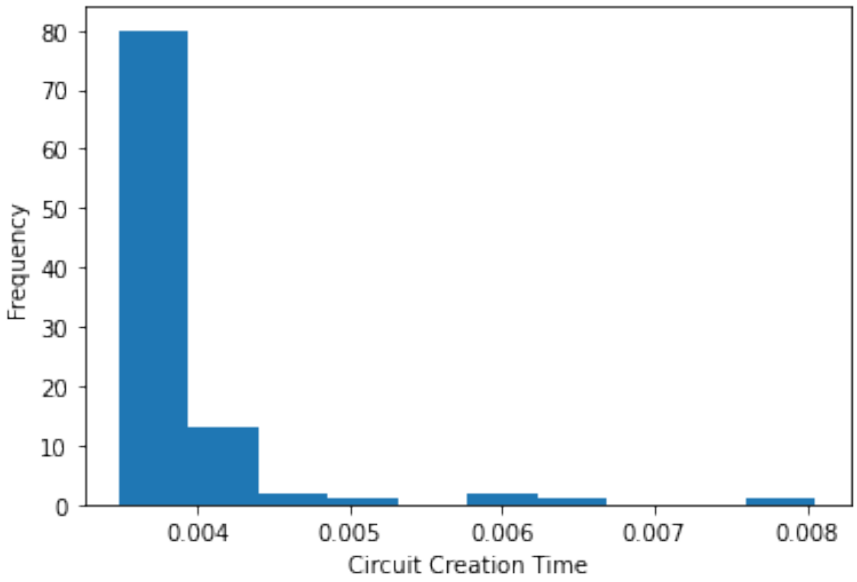


Figure 8: Circuit Generation time for different U_f when $n = 5$ and $a = 1$.

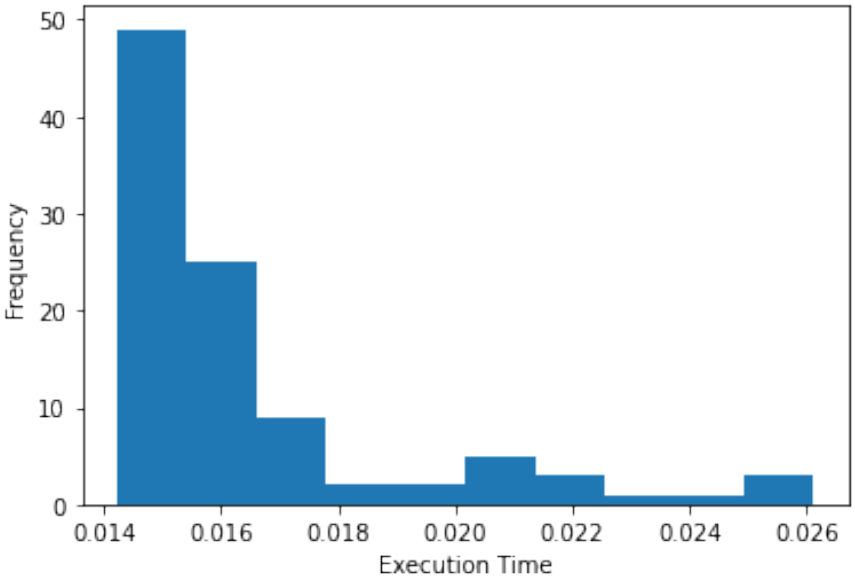


Figure 9: Execution time for different U_f when $n = 5$ and $a = 1$.

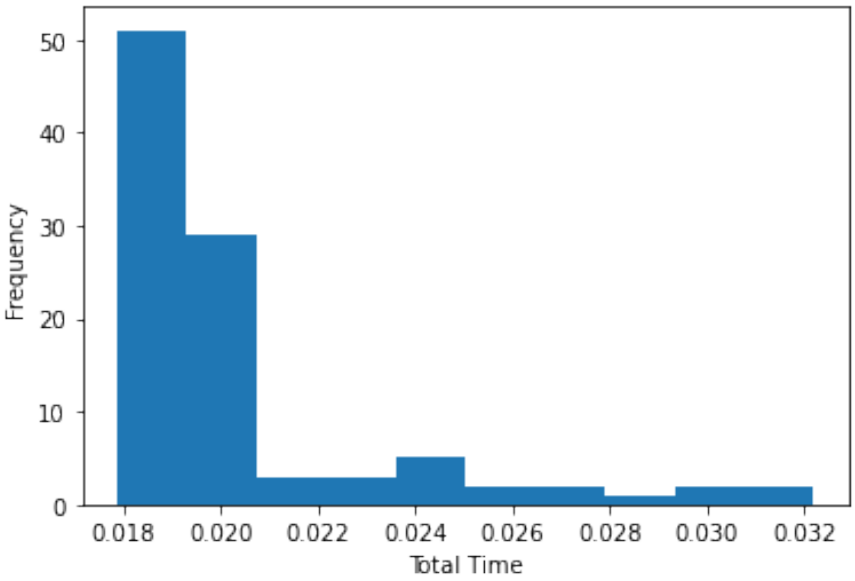


Figure 10: Total time for different U_f when $n = 5$ and $a = 1$.

2. We expect that for a fixed n and small a's, increasing the value of a should increase the execution time. We expect this because the larger the value of a, the more the deviation of G from the identity matrix. Hence, for $a = 1, 2, 3$, we computed the average circuit generation and execution time over 100 iterations for each case. As expected, the execution time increases with increasing a (see 11, 12, 13). a = 1 alone has a higher value. This makes sense as a = 1 requires 2 iterations instead of 1.

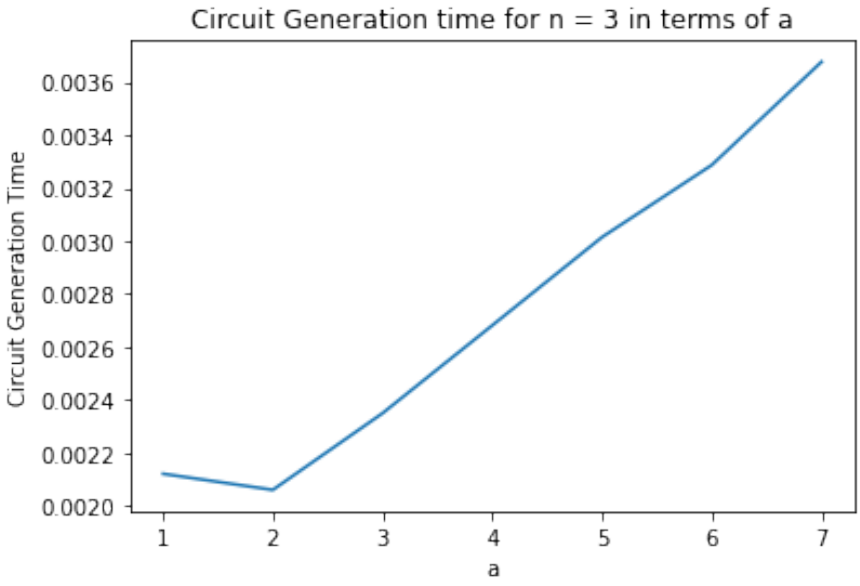


Figure 11: Average Circuit Generation time for U_f when $n = 3$ and $a = 1 \rightarrow 7$.

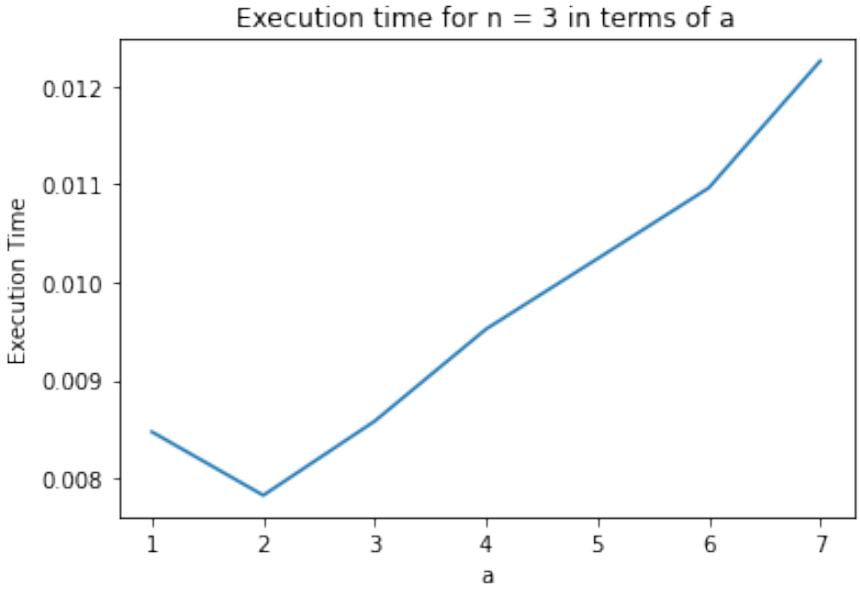


Figure 12: Average Execution time for U_f when $n = 3$ and $a = 1 \rightarrow 7$.

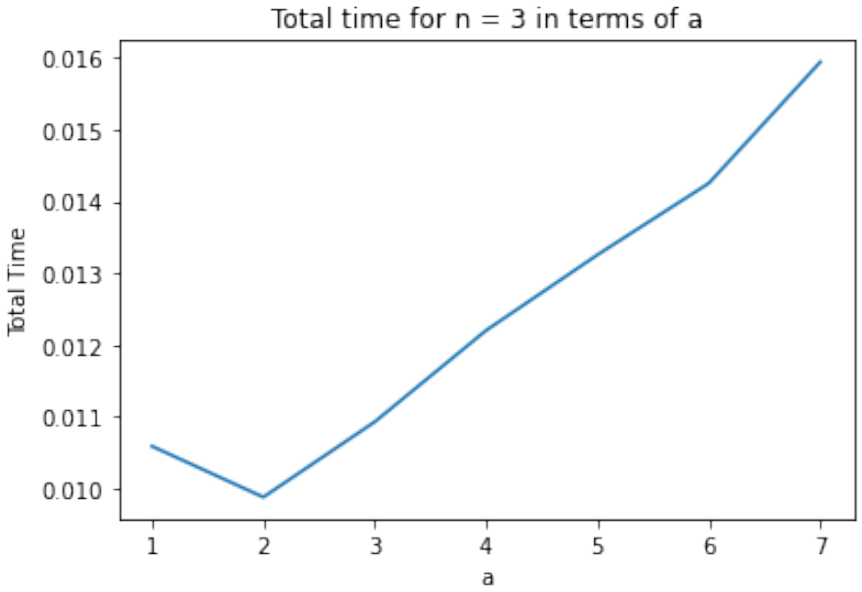


Figure 13: Average Total time for U_f when $n = 3$ and $a = 1 \rightarrow 7$.

1.2.6 Scalability

We benchmarked the average circuit generation and execution time for every $n \in \{1, 2, 3, 4, 5, 6, 7\}$ and $a = 1$. We observed that the circuit generation and execution time for the quantum circuit simulation grows exponentially with n. As can be seen in figures 14 and 15 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 15 as shown in 17, 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.

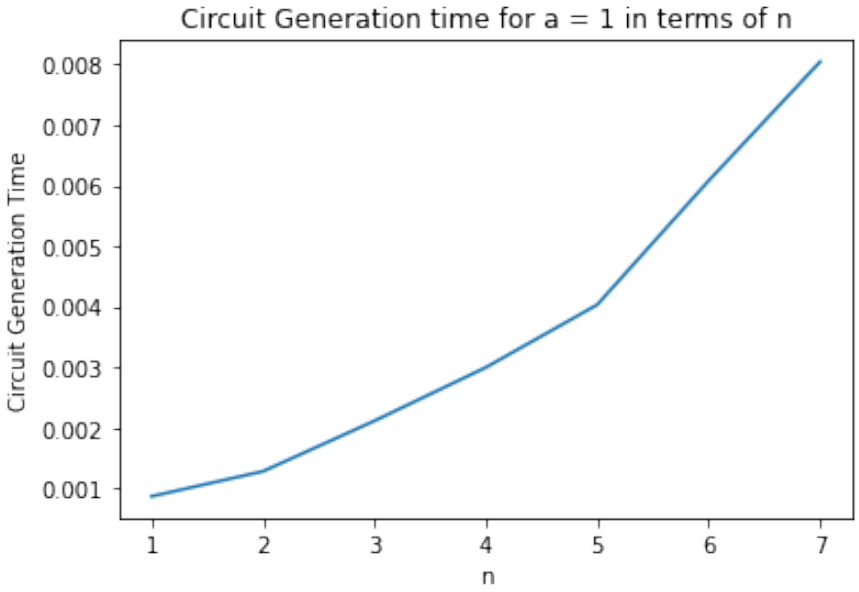


Figure 14: Circuit Generation time scaling for Grover's algorithm.

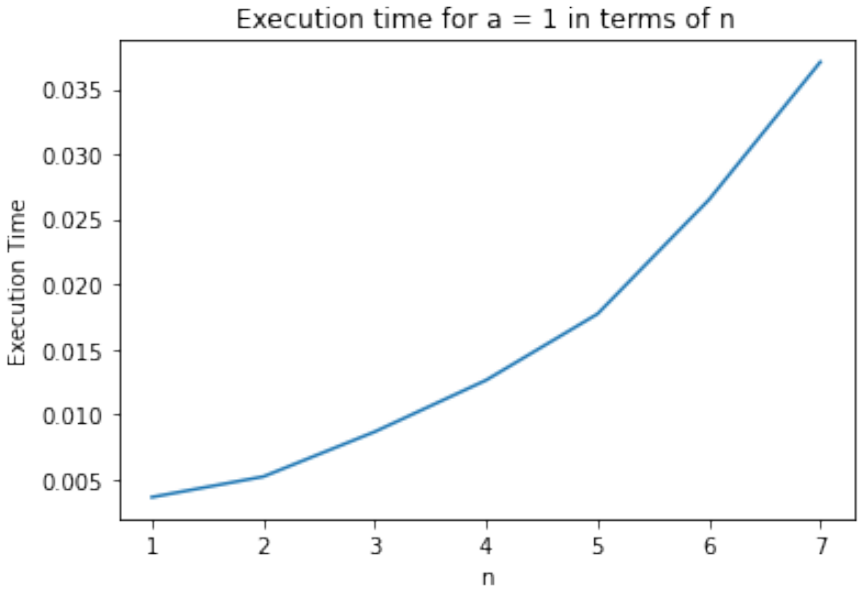


Figure 15: Average Execution time scaling for Grover's algorithm.

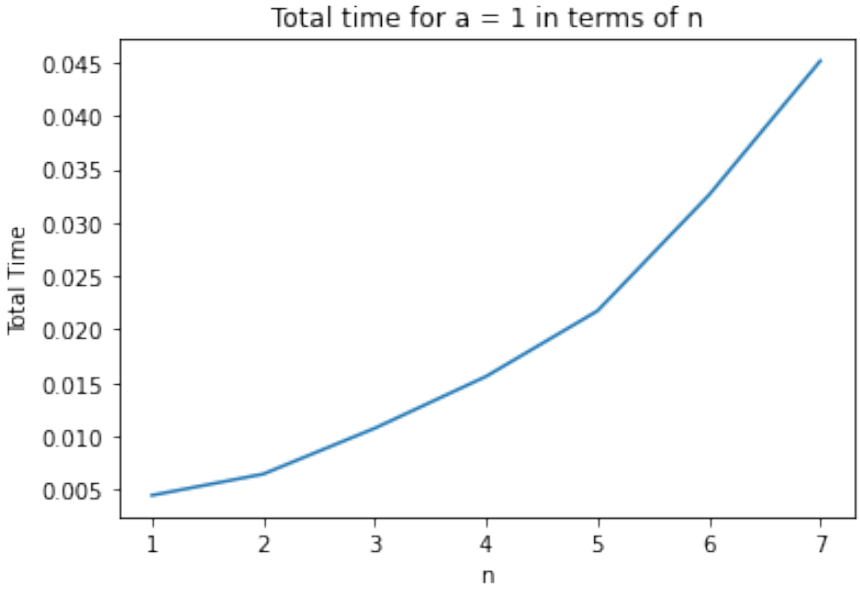


Figure 16: Total time scaling for Grover's algorithm.

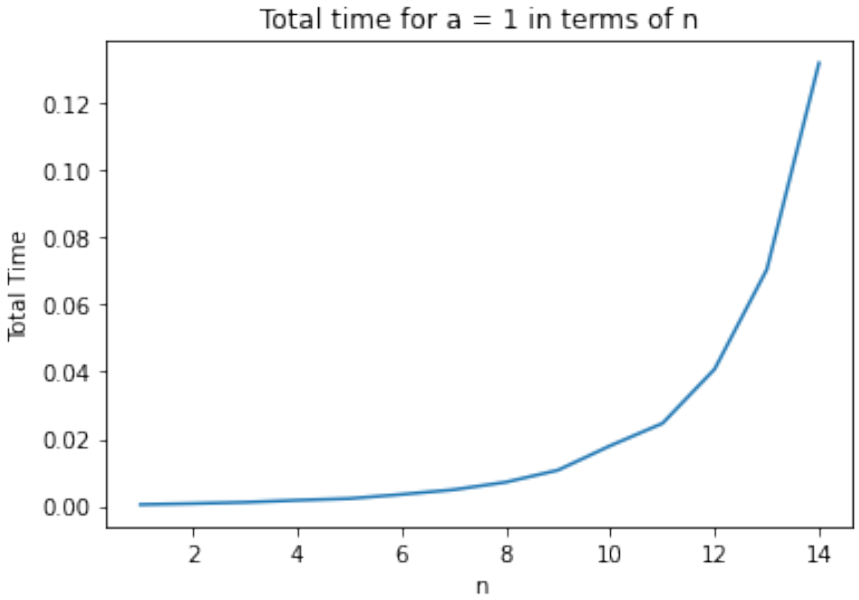


Figure 17: Total time scaling for Grover's algorithm.

1.2.7 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 100 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.

1.2.8 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.

1.2.9 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. Moreover, while applying the gates, we have made sure to use cirq features like on_each and $controlled\_by$, so that there is one gate per line without for loops. This makes it easier to understand the circuit.

1.3 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. 18. If the measured state is $\left|0\right\rangle^{\otimes n}$, then the algorithm concludes that f is constant, and balanced otherwise.

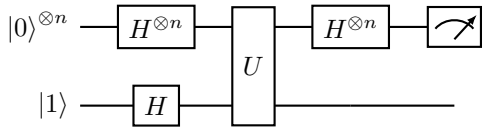


Figure 18: Deutsch-Jozsa algorithm

Now we discuss various aspects of our code design:

1.3.1 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 `Oracle` in Cirq in order to get the Gate object.

1.3.2 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(f)`: Creates the Uf matrix using the logic mentioned above.
2. `bitstring(bits)`: Converts bits of number to a string.
3. `runAndPrint(n, qubits, ufMatrix, 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.

1.3.3 Parametrizing the solution in n:

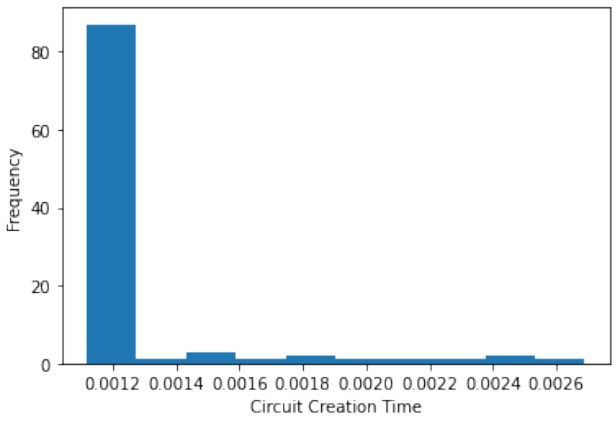
The circuit is different for every value of n. We dynamically create the circuit corresponding to any given value of n, run the circuit, and interpret the result of the measurement to determine whether the function is balanced or constant.

1.3.4 Code testing:

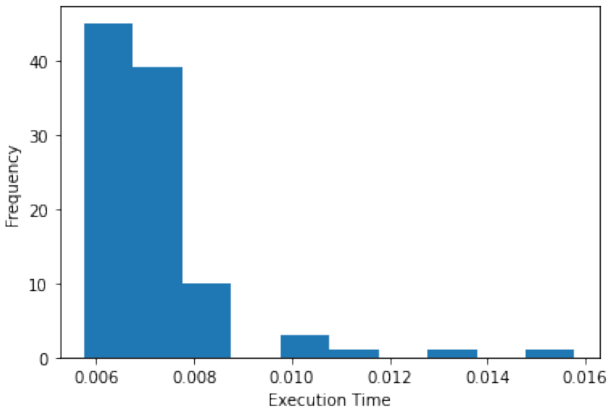
We first performed correctness check by running over various values. We tested for various values of n and various different types of Uf and it has always provided accurate results 100% of the time, as expected.For any given function it returns all 0s if it is constant and anything other than that when it is uniform.

1.3.5 Dependence of execution time on U_f:

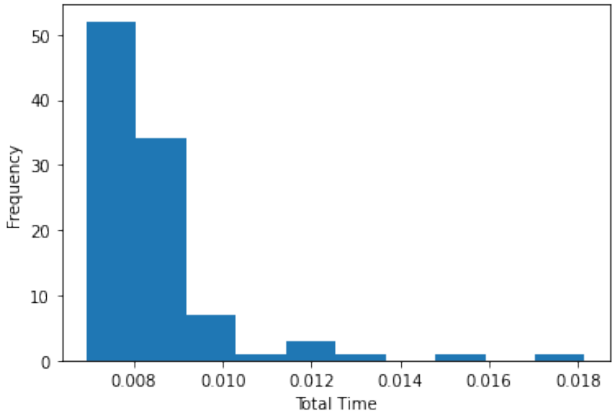
1. We ran the implementation for $n = 5$, 100 times, each time generating a random uniform function. The fastest U_f was close to twice as fast as the slowest. We divided the time into two: time required to create the circuit and the simulation run time. (see 19a, 19b, 19c).



(a) Circuit Generation time for different U_f when $n = 5$



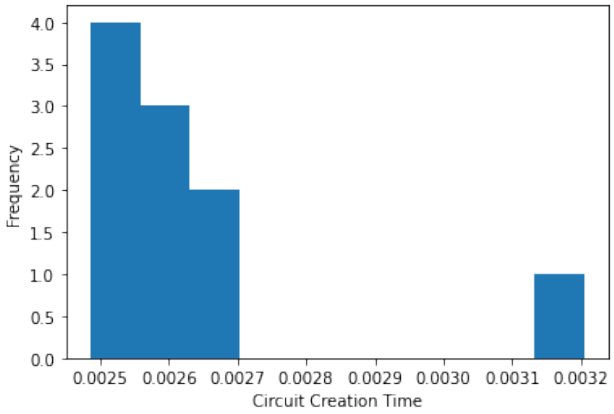
(b) Average Execution time for different U_f when $n = 5$



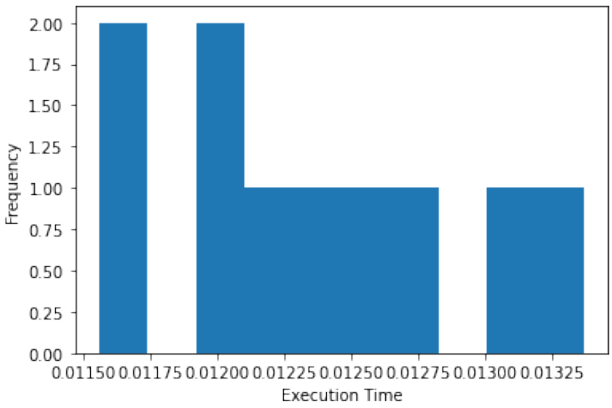
(c) Total time taken for different U_f when $n = 5$

Figure 19: DJ algorithm: Effect of Uf on time (Uniform case)

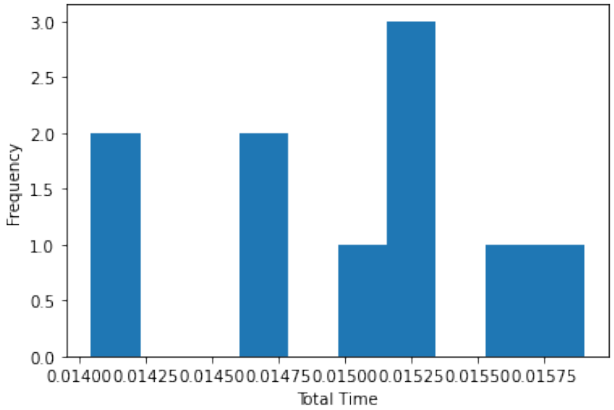
2. Specific to the constant function, we repeated the entire process of finding U_f for the two cases possible: all 0's or all 1's. Our observations are plotted and shown in the figures(see 20a, 20b, 20c). We see that in this case the times are much better than



(a) Circuit Generation Time range for n=5 for different Uf.



(b) Exection Time range for n=5 for different Uf.

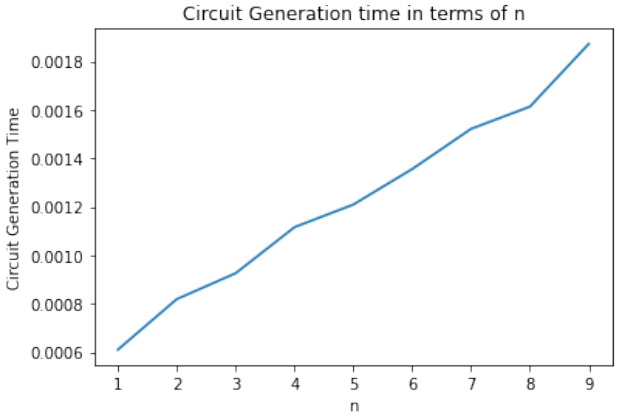


(c) Total Time range for n=5 for different Uf.

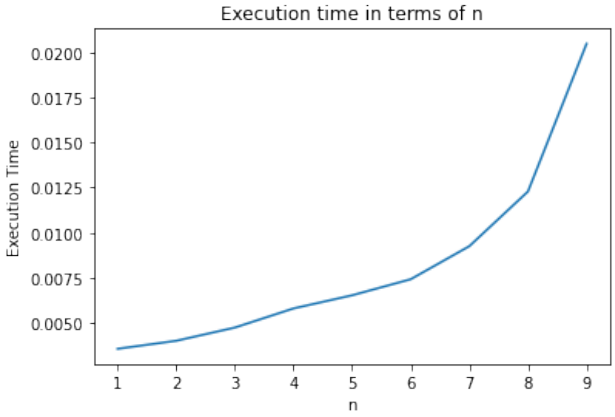
Figure 20: DJ algorithm: Effect of Uf on time

1.3.6 Scalability:

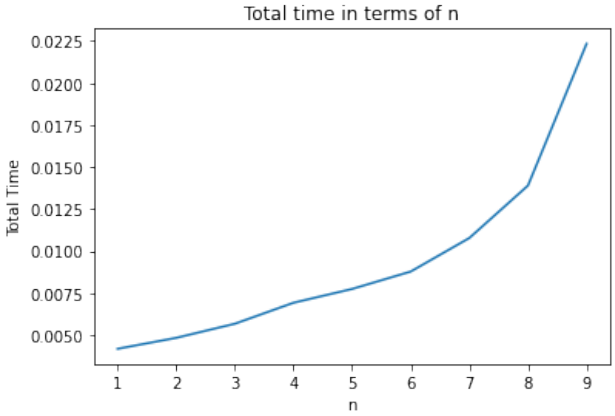
We observed that the execution time for the circuit simulation grows exponentially with the number of qubits involved, i.e. $n+1$. For each $n \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, we executed the Deutsch-Jozsa algorithm for five randomly chosen U_f's, and obtained the average execution time for each value of n. As can be seen in figure 20, the execution time grows exponentially, with $n = 6$ requiring about 0.0225 total time. 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 scaling for DJ algorithm.



(b) Average Execution time scaling for DJ algorithm.



(c) Total time scaling for DJ algorithm.

Figure 21: DJ algorithm: Effect of n on time

1.3.7 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.

1.3.8 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.

1.3.9 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. 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.

1.4 Bernstein-Vazirani algorithm

First, we summarize the problem statement:

Given a function $f : \{0,1\}^n \to \{0,1\}$ 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. 18). 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

1.4.1 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.

1.4.2 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:`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'. `decimalToBinary(x,n)`: This converts a decimal value to binary. `getFx(x, a, b, n)`: This generates f(x) = x.a xor b value and returns it. `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.

1.4.3 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 right 'a' was found.

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. For any

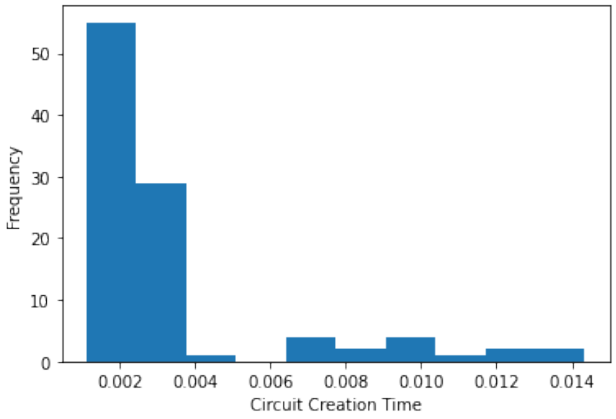
given function it returned the exact value of 'b' through classical means and then returns 'a' through quantum means.

1.4.5 Similarities in our codes for the BV and DJ implementations

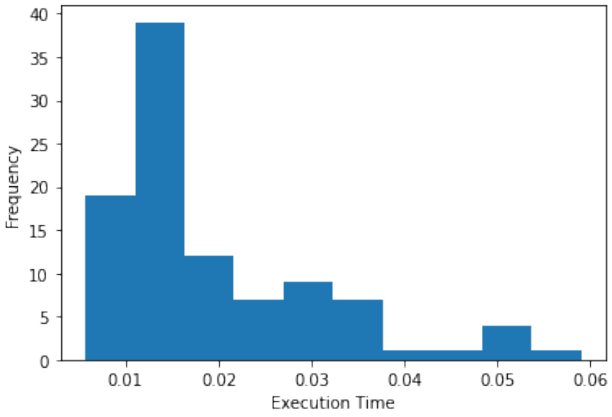
We estimate about 60% code similarity between our BV and DJ algorithms. In particular, the functions verb—Oracle—, `bitstring` and `runAndPrint` are identically defined in both the implementations. Moreover, the only difference between the `BV` and `DJ` functions is in the interpretation of the outputs, 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.

1.4.6 Dependence of execution time on U_f

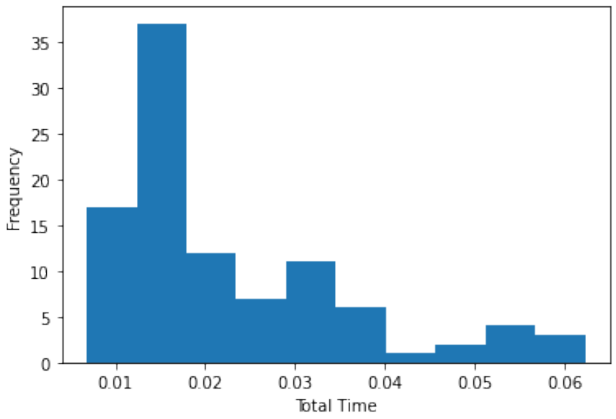
Similar to the DJ algorithm, we observe a variance in the execution time with U_f. For $n = 5$, we simulated 100 randomly chosen U_f's, and plotted the histogram in figure 22. Clearly, there is a higher variance in the execution times. This can be justified by the fact that the allowed U_f's in the BV problem have a larger variation from the identity matrix, compared with the allowed U_f's in the DJ problem.



(a) Circuit Generation Time range for n=5 for different Uf.

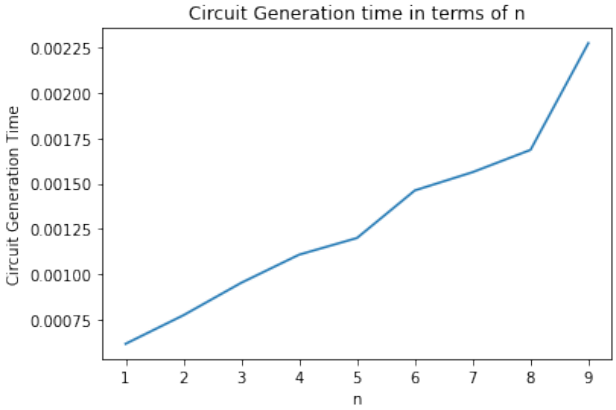


(b) Exection Time range for n=5 for different Uf.

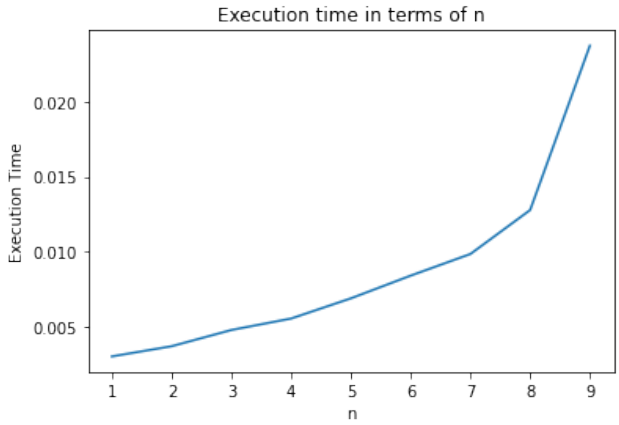


(c) Total Time range for n=5 for different Uf.

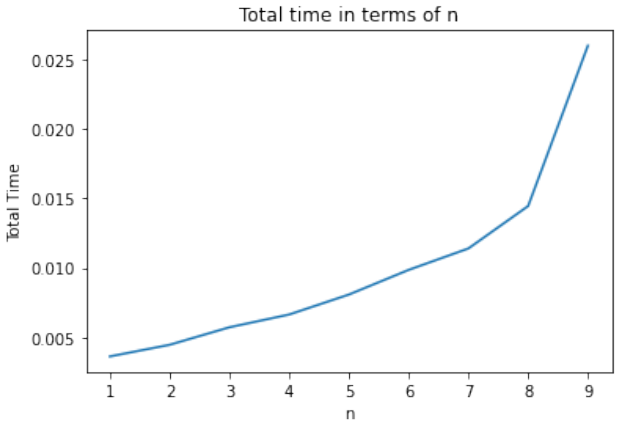
Figure 22: Bernstein-Vazirani algorithm: Effect of Uf on time



(a) Circuit Generation Time for different n.



(b) Execution Time range for different n.



(c) Total Time range for different n.

Figure 23: Bernstein-Vazirani algorithm: Effect of n on time

1.4.7 Scalability

Similar to the DJ algorithm, we observed an exponential increase in the execution time with n. For each $n \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, we obtained the average execution time for 100 randomly chosen U_f's. As can be seen in figure 23, 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.

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.

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.

2 Outline: The Language & Documentation

Most of the assignment prompts have been answered in the sections above. We also summarize our approach and observations below.

4. Design and evaluation

   (a) Present the design of how you implemented the black-box function U_f. Assess how visually neat and easy to read it is.
   → This part has been discussed in each of the algorithms sections. The oracles are different for each of the algorithms. The Simons take functio and n as the input and output the matrix whereas DJ and BV take only n as input and generate the matrix case by case. Grovers take the list of values where function has output 1 and accordinly creates the matrix. Comments have been written in code for better understandibilty.

   (b) Present the design of how you parameterized the solution in n.
   → To parametrize with n, we used the for loops everywhere. The U_f matrix creation, and quantum circuit generation, all functions have n passed as parameter.

   (c) Discuss the number of lines and percentage of code that your four programs share. Assess how well you succeeded in reusing code from one program to the next.
   → The DJ and BV algorithms had most of the code shared since the circuits are exact similar in the two algorithms. Approximately 60% of the code overlapped between DJ and BV (differences present only in creating the U_f matrix). There are around 100 lines of code in BV and DJ each out of which around 60 are common. The Simon's and Grover's share little code, since the circuits are different and the overall algorithm also differs. The Simon's algorithm had a classical section where we used the SAT solver to get the final solution. The BV algorithm has a classical part in terms of predicting value of b. Overall, across all codes we have tried to maintain consistency in code by introducing variables like `n` and `verbose` and functions like `getFx` and `runMaincircuit`. All circuits have similar modular structure, where we first create a random function in a separate python function, followed by an oracle and then followed by a function for creating the quantum part of circuit. The Testing cases were also resued as much possible, by analysing histograms and execution times. However all circuits had component unique to them like, Grover had totally different oracle function, as opposed to other algorithms, Simon had big chunk of classical part. DJ had the peculiar balanced and constant function types which were explicitly handled during creation of U_f matrix.

   (d) Discuss your effort to test the four programs and present results from the testing. Report on the execution times for different choices of Uf and discuss what you find.
   → The section has been discussed in previous algorithms sections independently.

(e) What is your experience with scalability as n grows? Present a diagram that maps n to execution time.
→ In general we observed that as n grows, time increases exponentially in all circuits. Maximum we could go till n = 7, after which it takes lot of time to run. All the plots were ran for many iterations and averaged out to remove the effects of outliers. All graphs have been added in individual sections of algorithms.

2. README:

(a) 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.

(b) Follow these links: Install numpy and Install matplotlib.

(c) Run the notebook one by one.

(d) 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).

(e) BV: Call runMainCircuitBV with parameters n and verbose (optional). eg: runMainCircuit(3, True).

(f) Simons : Call runMainCircuit with parameters n and verbose (optional). eg: runMainCircuit(3, True).

(g) Grover's algorithm: Call runMainCircuit with parameters n, a (optional) and verbose (optional). eg: runMainCircuit(3, 1, True).

3. Cirq:

(a) List three aspects of quantum programming in Cirq that turned out to be easy to learn and list three aspects that were difficult to learn.
→ Aspects of Cirq which were easy to learn:

i. Constructing gates: Construction of custom gates was straightforward, since it only involved providing the numpy matrix defining the unitary matrix.

ii. Constructing circuits: Construction of circuits is very easy, all one has to do is add it to the program in sequential order. There are multiple ways to add as well like append or create from scratch.

iii. Running and Simulation: It is very convenient in Cirq to simulate quantum computers. The results can be obeserved in a histogram, as a dictionary or even just as bits. Intermediate values of circuit can also be obtained and viewed in various forms including the convenient Dirac's notation.

iv. The interface between the quantum computer and the classical program is pretty seamless. Therefore onboarding was very easy.

→ Aspects of Cirq which were difficult to learn:

i. Lack on enough online resources: It was very difficult to handle errors as the error messages were not very clear. Moreover unlike most python libraries, it was not easy to find online resources addressing the errors.

ii. Implementing not so common gates: Gates like the CZ gates were not easy to understand. The passing of the parameters and the universal controlled by helped us overcome the issue.

iii. Visualization of the results: Particular functions like $plot\_state\_histogram$ were hard to learn. Details like $fold\_func$ parameter required a lot of reading of the documentation.

(b) List three positives and three negatives of the documentation of Cirq.
→ Positives:

i. The documentation was fairly detailed. It provided examples for anything we needed. It showed examples of defining gates in different ways which helped give ideas on how to program.

ii. The documentation was clearly modularized. So we knew which section we had to go to. Moreover it divides the sections into beginners and advanced users. So we are able to learn in order.

iii. The documentation provides an introduction to quantum computing. All the basics are really well explained and serve as a good review before starting any programs.

iv. The documentation has well defined tutorials and colab notebooks to try the code on. So it is not like other theoretical documentation. Here we can practice as well.

v. Finally they have provided Experiments where we can learn about research experiments.

Negatives:

i. Debugging is the biggest concern. The various errors you might encounter during implementation are not mantioned.

ii. There are a number of different gates possible. Though the documentation provides a number of examples on how to use gates like X, H or CNOT, there are hardly any for gates like CZ.

iii. Most of the examples in the documentation cover for one or two qubits. Generalization to an n-qubit model is hardly provided. A lot of reading is needed to lets say learn how to extent CNOT or CZ to 50 qubits.

iv. References section has a lot of information with no examples and hence it is very difficult to understand what the various parameters do until we read through the entire page.