CS239 - UCLA

SPRING 2019

HOMEWORK 10: REPORT

Using Cirq for Quantum Algorithms: Deutsch-Jozsa and Bernstein Vazirani

GROUP MEMBERS

Rishab Doshi - 305220397

Srishti Majumdar - 105225254

Vidhu Malik - 305225272

Deutsch-Jozsa and Bernstein Vazirani

Part 1. Design

We have organized our code into the below broad categories:

- Common Util Functions
- Representing Functions
- Quantum Circuit
- Post-processing on Quantum Circuits
- Driver code to verify

Common Util Functions

This section defines common utility functions that are used throughout of the program.

Table 1.1: Common Util Functions

Description	Function
Get decimal number from binary bit String	getDecimalNo(bitString)
Get qubit vector from the binary bit vector String	getQubitVector(bitString)
Get all possible bitstrings(2^n) for n bits	getAllPossibleNBitStrings(n)

Representing Functions

In this section, we have defined a generic object that can be used to represent functions. We have created a top level class called FunctionObject that defines the blueprint for all the functions that we consider.

The FunctionObject class has:

Table 1.2: Function Representation related functions

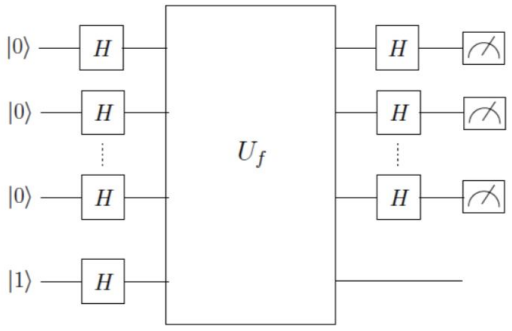
Description	function
The pointer to the actual function that is to be executed on the input	applyFx(input)
The N value for the domain of the function. i.e., f(x) : $\{0,1\}^n$ -> {0,1}	getN()
The logic to create the oracle function matrix U_f matrix	createUf
The code to verify that outputs generated by U_f and f(x) are the same	verifyUf

In addition to this, we also create subclasses to represent

- Deutsch-Jozsa
    - Has the logic to create random Balanced or Constant Functions
- Bernstein-Vazirani
    - Has the logic to create random Bernstein Vazirani Functions based on a and b values

Quantum Circuit

This part of the program uses Pyquil to create the quantum circuit as below. The same circuit is used for both Deutsch-Jozsa and Bernstein-Vazirani functions. We use the DefGate library in PyQuil to create a U_f gate based on the U_f matrix generated.



Post-processing on Quantum Circuit

After applying the quantum circuit to the qubits |000...01> and measuring, we inspect the first n bits.

- In case of Deutsch-Jozsa, if all the n bits are 0, the function represented by U_f is constant and 1 otherwise
- In the case of Bernstein-Vazirani, all the n bits that are measured are the value of a in ax + b. To get the value of b, we perform a classical step where we apply f(x) on a bitstring of all 0s.

Driver Functions to Test Code

We also create driver functions for both Deutsch-Jozsa and Bernstein-Vazirani. These test the values output by the quantum circuit and assert that they are consistent with actual internal function implementation. Specifically, that a constant function is identified to be constant, similarly with balanced functions. Also, in case of Bernstein-Vazirani, the value of a identified by the circuit is consistent with that of actual function.

1.1 Implementation of U_f

To implement U_f (the oracle function gate) we make use of the fact that U_f has to be a permutation matrix. We use the below steps to generate U_f matrix are:

1. If the input size of X is N, then generate all bit combinations of size N+1.
2. Map each combination to its decimal counterpart. We would hence get a mapping from the bit combinations to its first 2^(N+1) decimal counterparts.
3. The decimals represent the row and column indices of the unitary matrix.
4. We know that Uf |x> |b> = |x> |f(x) + b> where '+' is addition mod 2.Hence, each N bit |x> is associated with both possible values of the single bit b to get the above-mentioned N+1 input bits.
5. Now out of these bit combinations, the first N bits are sent to the function f, and the generated output f(x) and b are added modulus 2 to get the N+1 output bit. The first N bits of input and output are the same. Now, using these input to output mappings we can fill the matrix.
6. The input bits are used to get the row number and the output bits are used to get the column number using the mapping created between bit patterns and indices initially.
7. All other bits are made 0 in that row and column.
8. Once the process is completed for all input bit combination, we would not have the unitary matrix that is also a permutation matrix and represents the function f in its reversible form.

1.2 Preventing user from accessing the implementation of U_f

The method that creates U_f is present inside the FunctionObject class. It is made as a private member function by adding a prefix of __ to its name (__createUf). This ensures that it is not accessible to clients of this FunctionObject. Further, this __createUf function is invoked by the FunctionObject constructor and not accessible to a user outside. The user hence has no access to the actual implementation of Uf creation. In python, if a user cannot access a function, then the user cannot change its definition/implementation either.

To ensure that the U_f matrix that is created conforms to the user-specified function, we have created a function called verifyUf that verifies that

$$U_f|x_1x_2x_3...x_n\rangle|b\rangle = |x_1x_2x_3...x_n\rangle|b+f(x)\rangle$$

For all possible x. i.e., all possible 2^n bit strings for bit vectors of size n.

Hence, with the effective use of classes and private methods, the implementation of U_f is encapsulated away from the user.

1.3 Parameterizing solution in n

All the sections of our code are completely parametrized in n.

Function Representation
Even for function creation, all the user has to pass is n.
To create a functionObject for DeutschJozsa, the user has an option of passing in only n value. The constructor will then create a Balanced or Constant function at random for this n value. This function object can then be used across the program for running the quantum circuit, testing the code in the drivers, etc.
To create a functionObject for BernsteinVazirani, user can stop at just passing in the value of n. This is then used to randomly initialize a bit-string of size n, which is used as the value of a.

Quantum Circuit
The quantum circuit is parametrized by the functionObject which is then used to get the n value of the function. Based on the n value, we apply the NOT gate to the (n+1)th bit and Hadamard to all the bits, followed by U_f to all the qubits and Hadamard to the first n bits. The U_f gate is created using the DefGate function in PyQuil which can create a gate for a given matrix. The matrix is created when we create the FunctionObject itself(invoked by the constructor).

Post-Processing Code
The post-processing code for both the Deutsch-Jozsa and Bernstein-Vazirani functions are parametrized by the functionObject and the n value is retrieved using this functionObject.

Verification of Quantum Circuit
Even for testing the code, we have created Drivers that take in value of n and create random functions that have the range as $\{0,1\}^n$. Using these functions and the quantum circuit outputs, we verify the outputs by checking consistency with the methods exposed by the FunctionObject. (getType() in Deutsch-Jozsa, getA() and getB() in Bernstein-Vazirani)

1.4 Codesharing

Most of the core implementation code for Deutsch-Jozsa and Bernstein-Vazirani is common. This includes:

- Code for the quantum circuit
- Code to construct the Oracle Function Matrix
- Common Util Functions

The major differences are in Function Representation, Post Processing and Testing, which makes sense as both these circuits are solving two different problems and code to process output, test and represent input will be context specific.

In table 1.4, we show the results of code sharing and reuse for our implementations of the algorithms. Comments were excluded as different group members document code differently for personal and user understanding.

Core Implementation [excluding comments]		Whole code [excluding comments]	
Lines of Code Common	Percentage of Code Common	Lines of Code Common	Percentage of Code Common
114	67%	114	48%

Table 1.4: Result of Reusing Code.

While considering core implementation, we excluded function verification and testing functions in both codes as compared to the whole code. Overall, we succeeded in reusing our implementation in Deutsch-Jozsa for Bernstein-Vazirani. On observing the core implementation for both, one can see that the only difference between the two is in function object creation.

Part 2. Evaluation

In this section, we check how well our code performs and its scalability in terms of n.

2.1 Testing our program and testing results

We created drivers for both Deutsch-Jozsa and Bernstein Vazirani that took n as a parameter and created random functions based on Algorithm type and tested the quantum circuit on them.

Deutsch Jozsa Testing

Testing for Deutsch-Jozsa was done as follows:

1. We created drivers to test the Deutsch-Jozsa circuit. These drivers instantiated function objects and ran the quantum circuit and verifies the consistency of the results.
2. Using the above driver we ran multiple simulations for the Deutsch-Jozsa Circuit.
3. The simulations were run separately for constant and balanced functions separately.

Observations:

- We noticed that for n>=6 we always received a timeout.
- We tested for constant and balanced functions with n=1,2,3,4,5 qubits and noticed that all the values output by the quantum circuit were always equal to the actual function type.
- We note that generally for each n, the execution time is similar regardless of the U_f. As n increases, the time for execution required increases.

Bernstein Vazirani Testing

Testing for Bernstein- Vazirani was done as follows:

1. Check for n=1,2,3,4...14
2. For each n, ‘a’ and ‘b’ were generated and displayed to be used as referenced.
3. The program was run. Quantum computing circuit is used for ‘a’ and classical solving for ‘b’ (using function solveB).
4. The results from the quantum circuit and the original function are checked and compared. Specifically, the ‘a’ and ‘b’ in Step 2.

N	a	b	Time In s
1	1	0	0.006150960922
2	10	1	0.001856088638
3	101	0	0.003164291382
4	0011	1	0.004128932953
5	01110	0	0.00420832634
6	000111	1	0.006457090378
7	1001001	0	0.01150202751
8	01101110	1	0.0127120018
9	111001001	0	0.02702784538
10	0001010111	1	0.07390999794
11	11110000010	1	0.2545011044
12	000011010100	0	0.8174929619
13	1100011011111	0	14.61310482
14	10110010100010	1	195.2954853

Table 2.1.b.1: Bernstein-Vazirani Testing Results.

We see our program is reliable and gives correct results on all instances. For each n, first n bits checked for ‘a’.
Note: Last bit can be disregarded in results for ‘a’ as it is the result of the measurement of transformation on helper bit.

The effect of changing U_f on execution time was also checked for n=1,2,3,4...14. For each n, 3 types of U_f was checked. Since we use a random function generator, we have noted the results only when U_f is different. The results for different simulations for n=1...4 ar are in Table 2.1.b.2.

Note: for a given n, it was found that when resulting U_f is same, the timing is usually similar.

n	Function Generated: a,b	Time Required
1	[0 1], [1]	0.022934436798095703
	[0 0], [0]	0.019999980926513672
	[1 0],[0]	0.016003131866455078
2	[0 0 0], [0]	0.0319974422454834

	[0 1 0], [1]	0.024006366729736328
	[1 1 0], [0]	0.024001121520996094
3	[1 1 1 1], [1]	0.03200009727478027
	[0 1 0 1], [1]	0.03999781608581543
	[1 0 1 1], [0]	0.03195500373840332
4	[0 0 0 1 0], [0]	0.028003931045532227
	[1 1 1 0 0], [1]	0.03599953651428223
	[0 0 1 1 0], [1]	0.03999781608581543

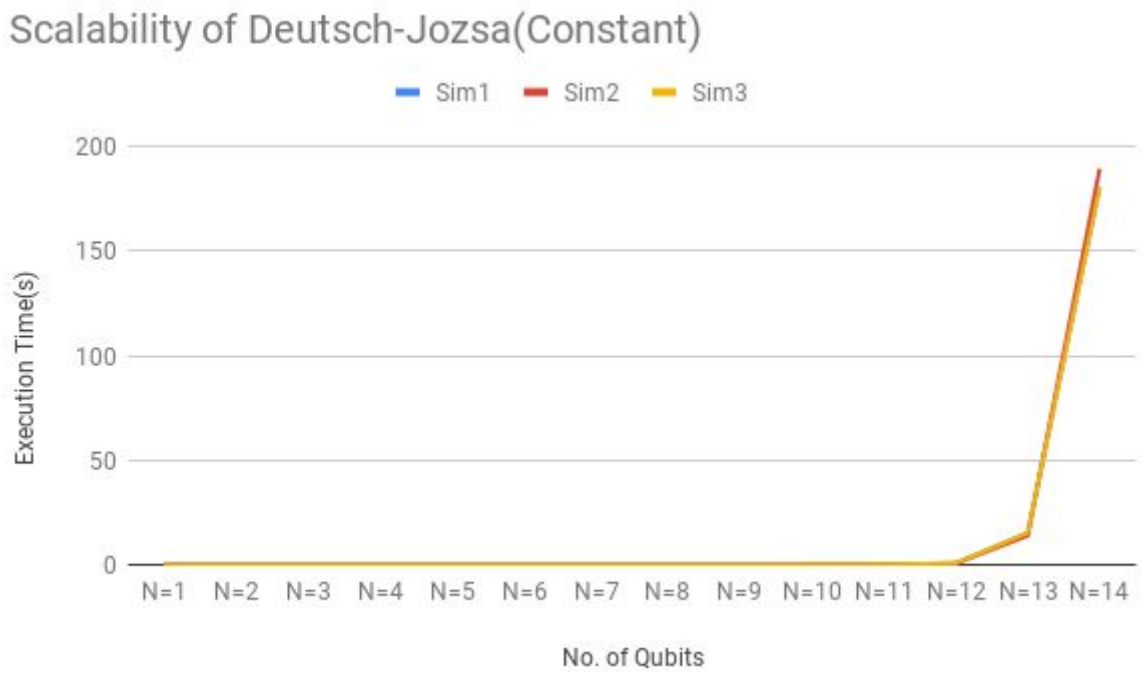
Table 2.1.b.2: Bernstein-Vazirani - Effect of U_f.

We note that generally for each n, the execution time is similar regardless of the U_f. As n increases, the time for execution required increases.

2.2 Scalability with respect to n

In this section, we show the different values of execution times seen for different n values. We use these charts to draw conclusions about the scalability of the quantum algorithms.

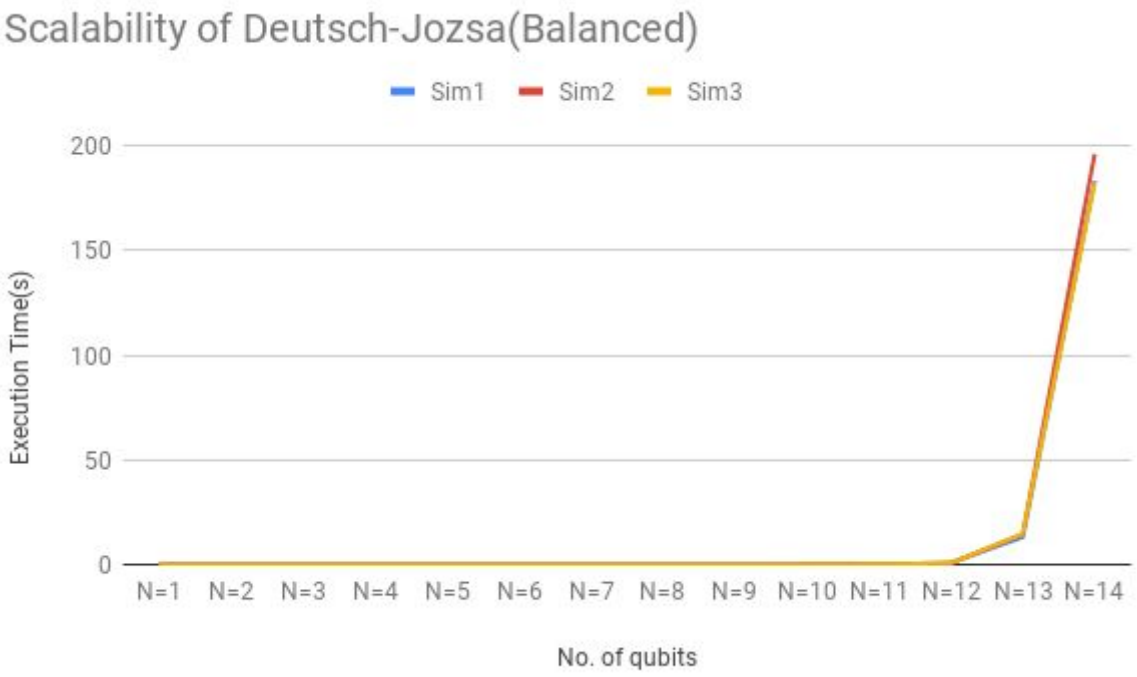
Figure 2.2.b.1: Execution Time vs N for DJ-Constant Functions



As can be seen in Figure 2.2.b.1, the execution times for the same n values are almost similar across different runs of the quantum circuit. Also, we see that with an increase in n value, the time taken grows almost exponentially as can be seen in Figure 2.2.b.1.

Similarly, for Deutsch-Jozsa Balanced Functions(Figures 2.2.b.2), we see that the execution times for the same n values are almost similar across different runs of the quantum circuit. Also, we see that with an increase in n value, the time taken grows almost exponentially.

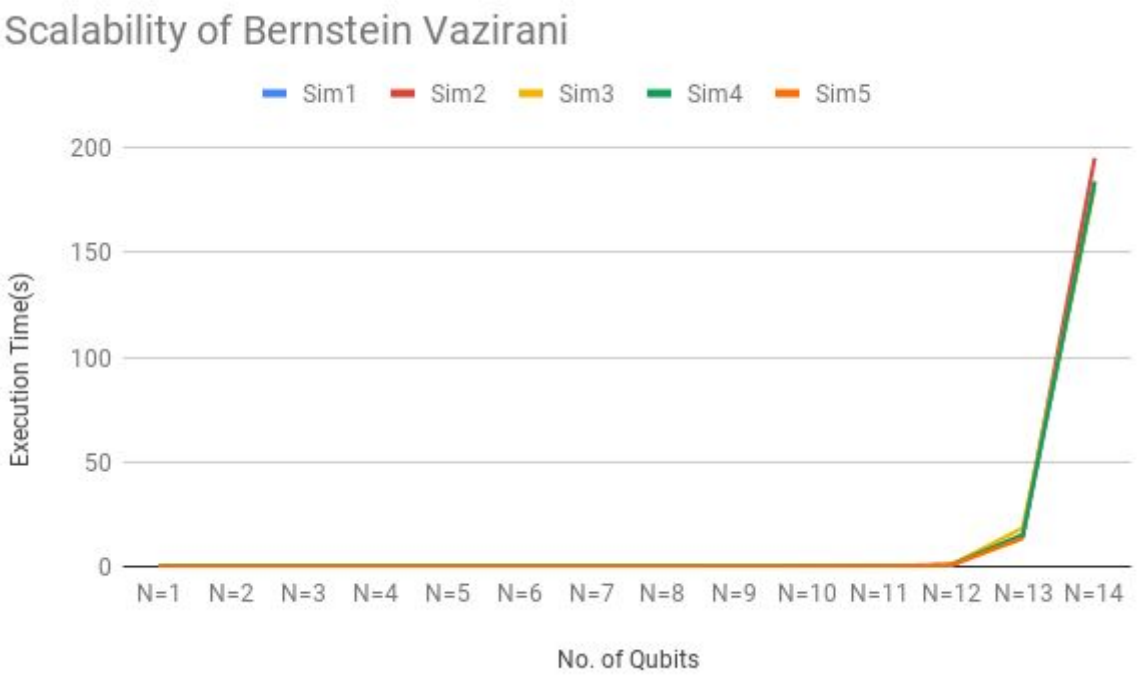
Figure 2.2.b.2: Execution Time vs N for DJ-Balanced Functions



For Bernstein-Vazirani, from Table 2.1.b.2, we see that execution time increases with n. While running the program, we also note that sometimes we received a TimeOut Error for n>14.

For various n, we can see the increase in time required as shown in figure 2.2.b.3. These results are based on simulations different from Table 2.1.b.1. The plot shows timings for 5 runs for n=1,2,3,4...14. As mentioned, n=15 the python notebook kernel just crashed, hence we showed simulations for up to N=14 only.

Figure 2.2.b.3: Execution Time vs N for BV Functions



At n=4, our U_f is 2^5x2^5. We notice that as n increases execution time also increases somewhat proportionally. Based on the default timeout, our code can be implemented successfully for maximum n=5. To make our implementation work for larger n, we need to change the default timeout factor. Hence, we see that our code is not scalable to large U_f matrices.