Zach Harris, 105221897
Trevor Rollins, 105227465

Deutsch-Jozsa and Bernstein-Vazirani in Cirq

I. Introduction

The Deutsch-Jozsa (DJ) and Bernstein-Vazirani (BV) algorithms are among the first problems that show a quantum computer can reduce the number of calls to a black-box function when compared to the most efficient classical solution.

II. Design

The overall design of our system is described below. Two classes are constructed, namely the `Deutsch_Jozsa` class and `Bernstein_Vazirani` class. As should be obvious from the naming convention, they set up and perform an evaluation on the DJ and BV problems respectively.

Because the circuits are so similar, the `Deutsch_Jozsa` and `Bernstein_Vazirani` classes were able to share a great deal of code (almost all of it). Five of their functions are identical, and a sixth (left_hadamards) is only slightly altered to account for the extra helper qubits in DJ. The global circuit governing both problems is shown below.

$$outcome = H^{\otimes n} I \circ U_f \circ H^{\otimes n+1} \circ |0^n> \otimes |1>$$

The U_f is entirely abstracted from the user by having them input a representation of f instead. For example, in DJ they will input a truth table that represents all possible values of f and in BV they will input the underlying a and b that constitute f.

Our system prevents the user from accessing the implementation of U_f. We never store U_f as a variable that the user can access. Rather, it is added to a circuit program that is only available to the user after the entire circuit is generated.

The two classes share a similar API for executing the algorithm. The only major differences stemmed from the building of U_f and accounting for extra helper quits in DJ. These differences are addressed in the next two sections.

Deutsch-Jozsa
We implemented the U_f black box function through a series of n-bit controlled NOT gates. The goal was that for each input x, where x is in {0, 1}^n, if f(x) = 1 then the helper bit, b, would be flipped from 1 to 0 by the controlled NOT gate. This is straightforward for n=1 as all that was needed was a single CNOT(q, b), in which f(q) = 1. If q happens to have a value of 0, then a X(q) would have to be placed on either side of the CNOT gate (this is true for any bit with a value of 0 in x). For n=2, two CCNOT(x1, x2, b) gates will be needed, where x1 and x2 are the bits in x where f(x) = 1. This process continues for n > 2, with the controlled not gate having n control bits.

Figure 1 shows an example of a circuit having n=3, where all three inputs (top three qubits) must be false in order to flip b (bottom qubit). As we mentioned, 1 helper bit (fourth qubit) is used to store the intermediate value. The last CCNOT is presented to flip h back to its initial '1' state.

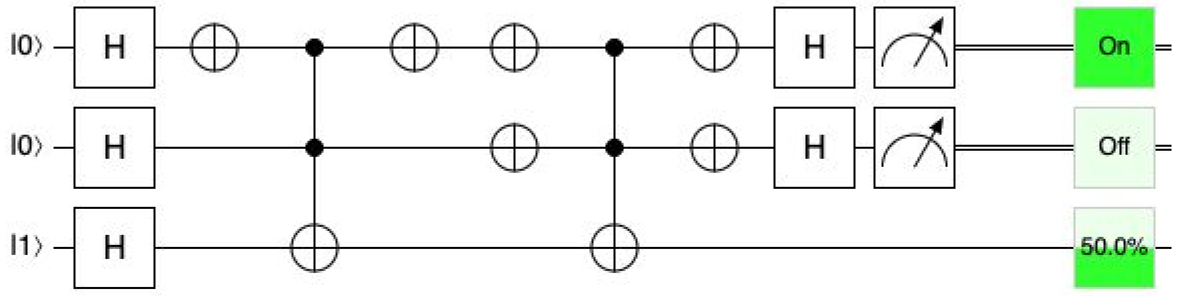

Figure 1 Example of DV circuit when n=2

This solution is successful in implementing U_f, but does require a large number of gates. However by having only one controlled NOT gate per x, where f(x)=1, it is not too difficult for the reader of the code to follow along. Unlike with PyQuil, we were ably to implement this without recursion, which makes the `build_Uf()` easier to follow.

The building of U_f is completely hidden from the user inside the `Deutsch_Jozsa` class. All the user needs to do is pass the oracle function, f, into the object to build the entire circuit. The oracle function here is expected as a truth table of inputs/output rather than a real function. This means that although our implementation of U_f hidden from the user behind an easy to use API.

Additionally, we were able to parameterize our solution in n by through the oracle function. The class will infer the n from the size of the inputs in the truth table, and build the circuit accordingly. We may expand our system to any n given that the Cirq simulator can support the number of bits necessary for our computation.

Bernstein-Vazirani

We implemented the U_f black box function through a series of CNOT gates to represent an f that has the form $f(x) = \overline{a} * x + \overline{b}$. After making a truth table containing |f(x)+b> and mapping out how $\overline{a}$ and $\overline{b}$ affected the outcome of U_f, we came to the realization that for any bit $\overline{a_i}$ that is '1', a corresponding CNOT(x_i, b) will be present in U_f where x_i is the ith qubit in the circuit. The truth table method also showed that we must add a NOT to b if $\overline{b}$ is odd.

It is a characteristic of the BV problem that $\overline{b}$ plays no role in U_f and after finding the value of $\overline{a}$ we can simply plug it in to determine $\overline{b}$. This can be shown by recalling that the result after measuring a BV problem will be the state of $\overline{a}$, and will never provide any information about $\overline{b}$. We therefore do not perform this operation on the quantum computer and rather leave it up to a classical computer to find $\overline{b}$.

Generalizing to n bits is a matter of adding a CNOT to b for any corresponding bits that are '1' in $\overline{a}$. This means that the system is very easily scalable and requires a minimum amount of gates that only depend on the underlying value of $\overline{a}$.

Since we know f is of the form $\overline{a} * x + \overline{b}$, we pass the values of $\overline{a}$ and $\overline{b}$ as integers directly into the `Bernstein_Vazirani` class instead of a truth table. This makes the classical derivation much easier, and frees us from having to first find $\overline{a}$ and $\overline{b}$ classically before being able to construct U_f. We therefore require $\overline{a}$, $\overline{b}$, and n as the three inputs to our **`insert_Uf()`** function.

Figures 2 and 3 below demonstrate two cases for the BV problem. Figure 2 shows the outcome of evaluating the circuit when $\overline{a} = 2$ and $\overline{b} = 1$. Figure 3 shows the outcome of evaluating the circuit when $\overline{a} = 2$ and $\overline{b} = 2$. Note that in either case, the value of the output when measured is '010' or 2, with a certainty of 1. U_f in either case is made up by the gates in between the Hadamards.

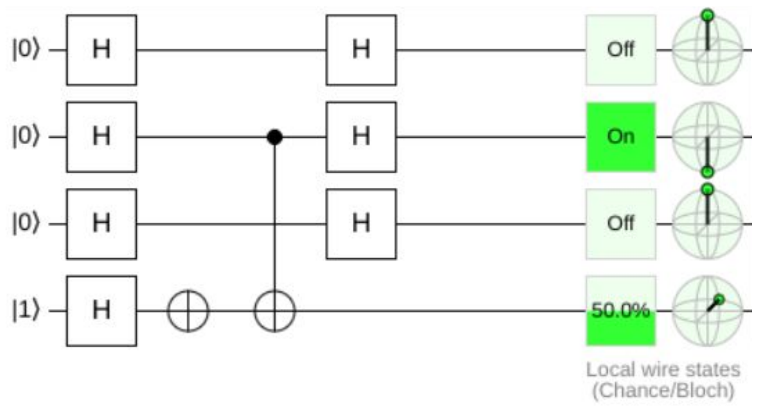

Figure 2 BV for $\overline{a} = 2$, $\overline{b} = 1$

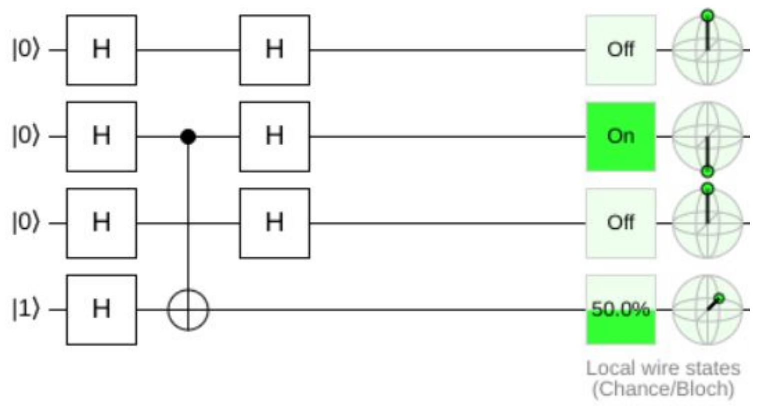

Figure 3 BV for $\overline{a} = 2$, $\overline{b} = 2$

III. Evaluation

Both algorithms were rigorously tested to verify the simulated value matched the predicted value. Each experiment is evaluated using the `run_DJ` and `run_BV` function calls. The circuit is first built using the corresponding `Deutsch_Jozsa` or `Bernstein_Vazirani` class, and then simulated using Cirq's simulator. We measure more than once to show that the circuit results are what we expect with probability 1.

Deutsch-Jozsa

For testing, 1 balanced and 2 constant oracle functions (f) were used for each number of input bits (n) between 1 and 4. Two constant f's were used at each n to test the case where f(x)=1 and f(x)=0 for all x. In every test case, constant f's returned $|0^n\rangle$ and balanced f's returned non-$|0^n\rangle$, which is exactly what was expected.

As seen in the figure below, the magnitude of n does not significantly affect runtime when evaluating a constant f. This is because U_f for a constant f can be achieved with a single X gate (or no gate at all if f(x)=0). However the size of n has a dramatic impact on runtime when evaluating a balanced f. This is to be expected as our implementation of U_f requires n/2 n-bit controlled-NOT gates. This is further compounded by necessary X gates. If q is the number of 0-bits in X, where X is the set of x's such that f(x)=1, then 2q X gates will also be required.

Constant functions do not need many gates to implement U_f. For constant function that always returns 0, U_f is simply identity. And for a constant function that always returns 1, U_f is a NOT gate on the helper bit.

The plot of these runtimes is proportionally similar to those in PyQuil. Balanced functions are exponential, and constant functions are approximately constant execution time. However in absolute terms the difference could not be more stark. A n=5 balanced function took about 0.006s to execute in cirq, while this same function took about 24s in PyQuil. This is almost a 4000x speedup!

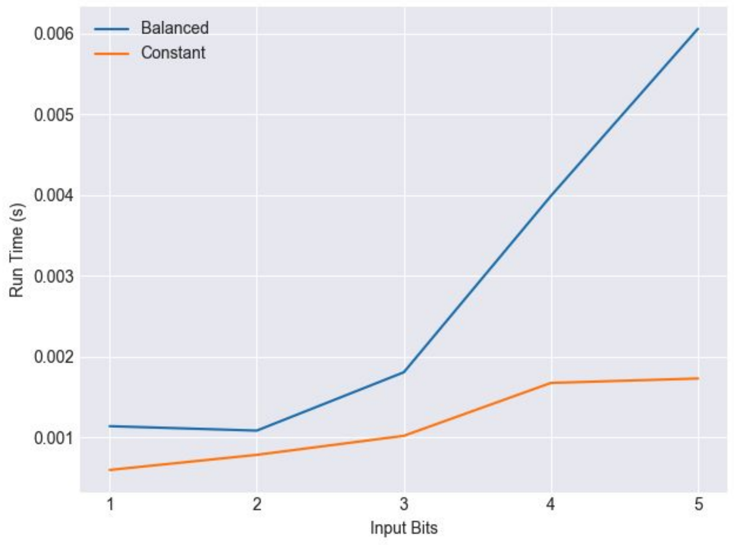

Figure 3 Execution times for DJ

Bernstein-Vazirani

The value of $\overline{a}$ was verified for each value of $\overline{a}$ up to 2^5-1=31. This was done by inputting every case of $\overline{a}$ and $\overline{b}$ into the system and ensuring the output was again $\overline{a}$. Since this problem scales with n, we can safely assume that it will work for any integer. Next, the quantum circuit was simulated for increasing values of n up to 8. The figure below shows the time taken by the simulator to perform each simulation. The blue line represents the most complex case for each n ($\overline{a} = 2^n - 1$ and $\overline{b} = even$), and the orange line represents the bare minimum ($\overline{a} = 0$ and $\overline{b} = odd$, no gates are actually added to U_f).

Much like the case in DJ, the execution time increases as the number of constituent gates in U_f increases. Although, for BV we only need to add at most $n+1$ gates for a given n number of input bits. This is why the run time appears to grow linearly with n. Both the orange and blue lines take the same amount of time to run on average. Although, the orange line is always lower than the blue line which makes sense because it represents the simplest circuit possible for a given n. We note that the blue line is very high when $n = 1$, which may suggest that the Cirq compiler isn't optimized for computing on a single qubit in BV.

Cirq takes at most 8 ms to run BV, whereas PyQuil took at most 1.8 s. For fairness we include the time it took to create the Cirq circuit into the run time measurement. Even so, the Cirq is much faster than PyQuil. A different qubit architecture is used for Cirq, which could be the reason for this immense difference in runtime.

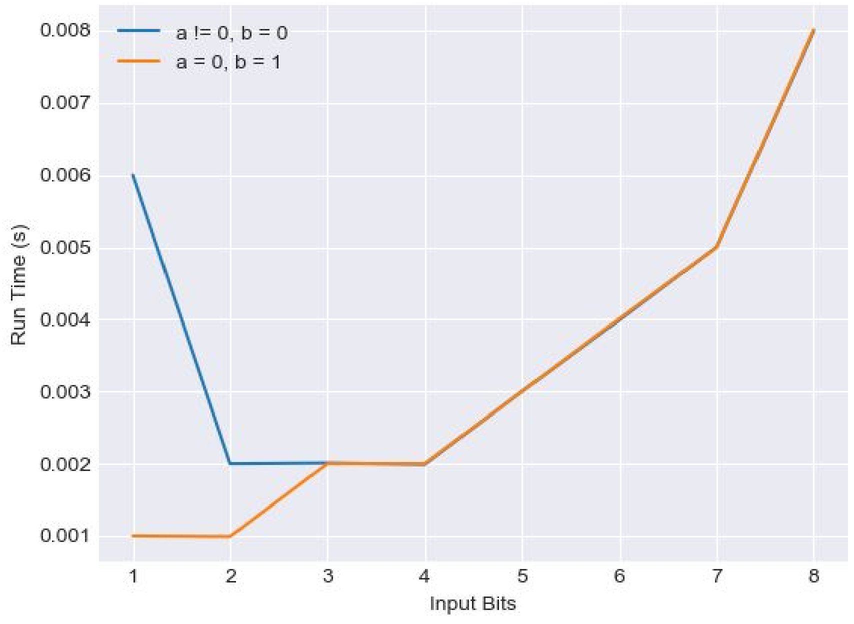

Figure 4 Execution times for BV

IV. Instructions

The README.md file describes the packages necessary and steps involved to run the tests. Please read the descriptions below that summarize this file for each experiment.

Deutsch-Jozsa

`deutsch_jozsa.py` is the script used run the Deutsch-Jozsa algorithm on a given oracle function (f). To use this script, you need to type out f as a list of lists. The outer list will consist of 2^n inner lists. Each inner list will be n+1 in length. The first n elements will be the input bitstring to f, and the final element will be the resulting output. Due to the use of helper bits in this implementation, $n \leq 5$ for any f.

EXAMPLE:

```
f = [[0,0,0], [0,1,0], [1,0,1], [1,1,1]]
```

Here n = 2 and f is a balanced function as half the outputs are 1.

Once you have your f ready all that is necessary is to call `run_DJ(f)`. This function will run Deutsch-Jozsa on f 5 times, and will print the 5 results in an array.

Currently there are 3 calls to `run_DJ()` present at the end of the script inside `main()`. You can add your call after these or simply delete them.

Bernstein-Vazirani

`bernstein_vazirani.py` is the script used run the Bernstein-Vazirani algorithm on a given function. To use this script, you will need to define the integers n, a, and b. The variable n is used to represent the number of input bits. The variables a and b are used to represent the function, $f(x) = \overline{a} * x + \overline{b}$. Simply pass these values into `run_BV(n, a, b)` and it will create the circuit, insert U_f, compile the program, and simulate it using the Cirq simulator. The result and execution time is returned by this function. You may add your calls to `run_BV()` at the end of the `main()` function.

V. Conclusion

Our results show that these algorithms are working as they should and can be tested using a real quantum computer without any risk of failure. By taking control of the gate compilation and distilling our implementation down into basic CNOT and NOT gates, we ensure that the Cirq compiler for both a simulated and real quantum computer will be exactly what we expect.