Zach Harris, 105221897
Trevor Rollins, 105227465

Simon and Grover in Cirq

I. Evaluation

Both algorithms were rigorously tested to verify the predicted with simulated value. Each experiment is evaluated using the `run_simon` and `run_grover` function calls. The circuit is first built using the corresponding `Simon` or `Grover` class, and then simulated using the Cirq simulator. We measure more than once to show that although the circuits may not give the same result each time, the correct result is indeed the most frequent.

Simon

For each cycle of the quantum portion of Simon's algorithm produces a y such that n-1 cycles are run to produce n-1 y vectors. From here, this series of dot products are used to find a set of valid values for s. Simon's algorithm is probabilistic in that there is at least a ¼ chance that all y's returned will be linearly independent. This means that there is a high chance over overlap between y's, and a single value for s will likely not be found from n-1 cycles. To test the algorithm, we first implemented a naive solver that checked every possible value of s with the set of y's returned from the quantum part. We ran many iterations (where each iteration is n-1 cycles) and compare the sets of valid s values returned. After enough of these iterations, the intersection of these sets will be of size one with a high probability. This is the non-zero value for s. In testing we found that a good value for number of iterations is $2n$, though this can be adjusted as a parameter of the `run_Simon()` function. The return value will be the set of valid s values found (ideally one value).

To test our algorithm, we used 4 oracle functions with input sizes (n) between 1-4. Each of these functions have a valid non-zero s value. In testing, when iterations < 5, a set of values of size > 1 was sometimes returned. This is to be expected, because if there is significant repetition between the y's used to generate the s values, then there will be many legal s values possible. Though a set of size > 1 is not ideal, these sets always included the real s value. This indicates that the algorithm was valid, and that the iterations just needed to be increased.

At iterations = 5, for every test run, a set of size 1 was returned and included the real s. We these results, we were confident that the algorithm performs correctly.

We implemented U_f for this algorithm by creating controlled X-gates (CX). A CX was created for each 2^n possible inputs for each output bit, b, when b = 1. The CX is controlled on the n input bits. For functions that have fewer output bits equal to 1, less CX's will need to be created, and execution time will be better. Also if CX are needed on input bits that are more heavily weighted towards equaling 1, these bits will not have to be NOT'd before and after the CX (because all control bits need to be 1 inside CX), which can also improve execution time. An example is shown in Figure 1.

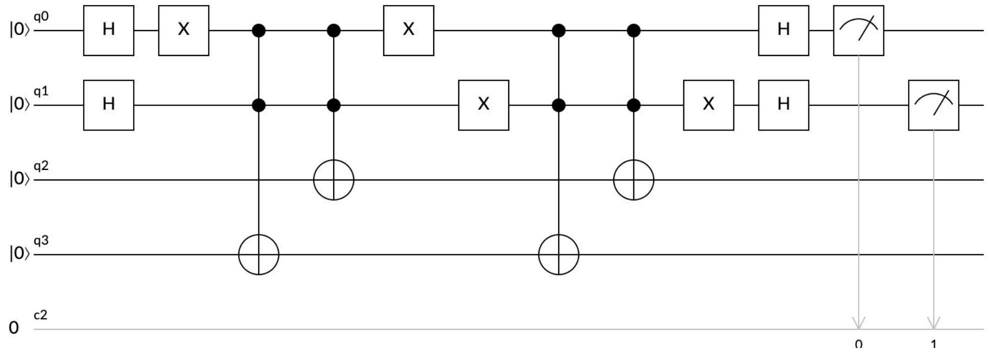

Figure 1 Example U_f for the function below

x	00	01	10	11
f(x)	00	11	11	00

On average however, half of the output bits will need CX's, and half the input bits will need X-gates before and after the CX. This leads to the number of CX gates needed to be: $2^n \cdot \frac{n}{2} = n2^{n-1}$ with each CX being controlled on n bits. And the number of X gates needed will be n(CX gates)= $n^2 2^{n-1}$. This means that as n increases, runtime increases exponentially. This effect can be seen in Figure 2. Execution time does not scale well in this implementation of U_f.

The execution times here are proportionally similar to those in PyQuil. Time appears to increase exponentially with n. However in absolute terms the difference could not be more stark. A n=4 function took ~0.065s to execute in Cirq, while this same function took ~700s in PyQuil. This is an unbelievable speedup of more than 10,000x!

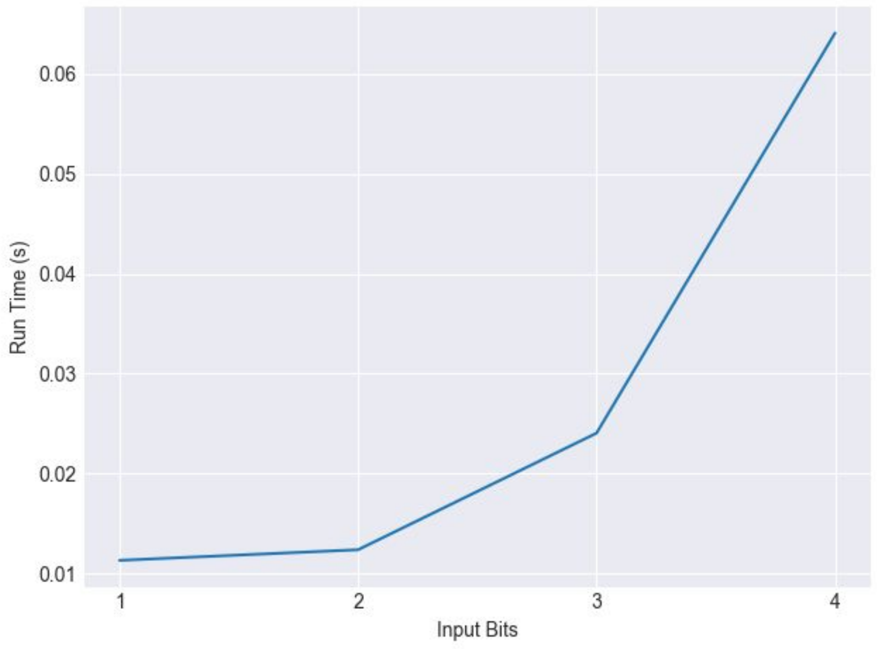

Figure 2 Run time of Simon's algorithm

Grover

Because Grover is not guaranteed to give the correct answer with probability 1, it is necessary to test the outcome multiple times. In order to produce a more accurate measurement for the system, we simply run the simulation multiple times using the `numshots` parameter from `run_grover` and count the most common-occurring situation as the correct answer. We found that running the simulation for 2^n times was enough to ensure that the correct answer was the dominant result from the simulation. For example, when running n=4 and k=2, we get the following output from the terminal:

```
Counter({2: 4, 5: 1, 0: 1, 4: 1, 7: 1})
most probable result: k=2 (4/8)
```

It is evident that k=2 is the correct answer, because the frequency of occurences is much higher than the others. After running multiple trials, we find that the probability of getting k=2 is 50% on average. This is close to the probability predicted by the Quirk circuit simulator, which is 47.3% (proof). We performed the same tests with n up to 6 to verify our results were accurate.

Our circuit scales very well with n. An example of a G block is shown in the figure below. The red box represents z_f, which is where f is introduced into the circuit. The Z gate is applied to any bit (the first qubit in this case), and a pair of X gates are added to either side for each bit in k that is 0. In this case, k=0 so there are a pair of X gates for each qubit. Notice that z_0, which negates the output if x=0, is equivalent to z_f since k=0.

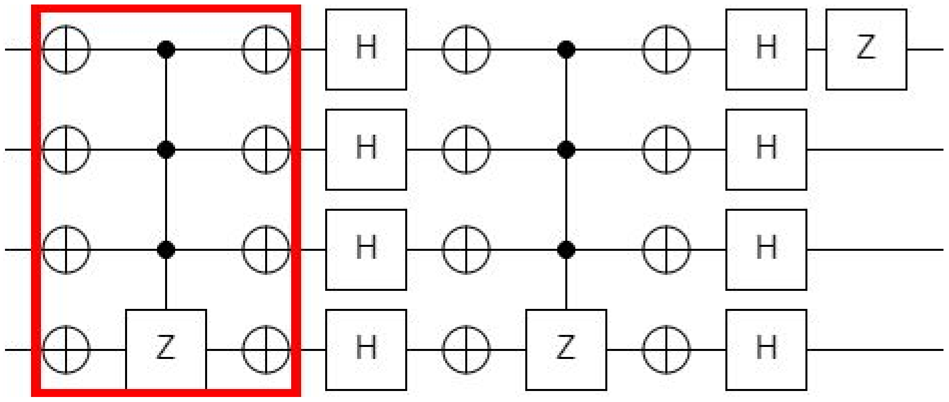

Figure 3 G circuit block

In order to scale to any n, we use (n-1)-controlled Z gates and insert the X gate pairs where needed. Although this presents an easy problem to code, the number of controls scales linearly with n. Figure 4 below shows the relationship between execution time and n. Compile time is included in this measurement. It is evident that although the number of controlled bits increases linearly, the execution time increases exponentially. This suggests that by adding controls to a gate will increase the execution time exponentially. There is little change from when the least number of gates are required (orange line, k is all 1's) to when the most number of gates are required (blue line, k is all 0's). This suggests that the time increase is entirely due the impact of adding more controls to the Z gates.

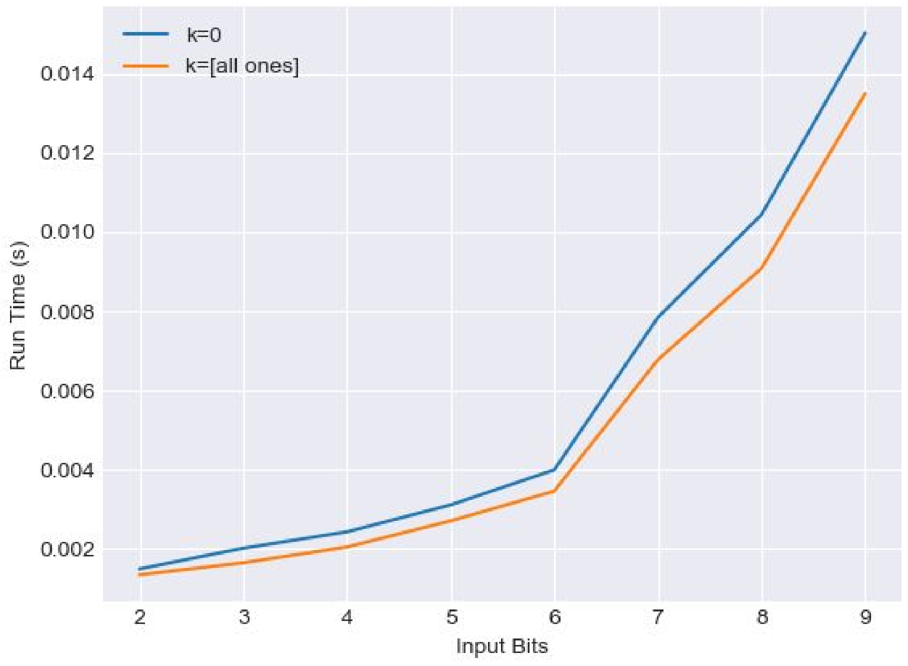

Figure 4 Relationship between Grover execution time and n

II. 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.

Simon

`simon.py` is the script used to run Simon's 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, where n is then number of input bits.
Each inner list will be 2n in length.
The first n elements will be the input bitstring (x) to f, and the final n element will be the resulting output f(x).

Example:

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

Here n = 2 and s = [1,1].
f must have a valid s value, or the algorithm will fail.

Once you have your f ready, all that is necessary is to call `run_Simon(f, naive=False)`.
This function will run Simon's algorithm on f. `naive=True` will use an naive solver to find s. False will use matrix null_space method (which is more efficient); naive defaults to False. This function will print out the non-0 value for s.

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

Grover
`grover.py` is the script used to run the Grover algorithm on a given function. To use this script, you will need to define the integers n and k. The variable n is used to represent the number of input bits. The variable k is used to represent the value of x where f(x)=1. It is assumed that in all other cases of f, f(x)=0.

Simply pass these values into `run_grover(n, k)` and it will create the circuit, compile the program, and run it on the Cirq simulator. The function takes the following extra parameters: `numshots` and `print_p`. `numshots` governs how many times the simulation will run. `print_p`, when `True`, will print the Cirq circuit in ascii for debugging.

The result and execution time is returned by this function. You may add your calls to `run_grover()` at the end of the `main()` function.

III. Cirq Reflections

Three aspects of programming in Cirq that turned out to be easy to learn

1. Adding gates to a Cirq circuit only requires appending an array of gates to a circuit object.
2. Cirq is a Python library which meant that it was easy to integrate with other existing Python libraries. This was good for quick implementation and post-processing.
3. Debugging was very straightforward, because Cirq lets us print the layout of a circuit at any time.

Three aspects of programming in Cirq that turned out to be difficult to learn

1. Cirq introduced the concept of a Moment, which contains a series of Operations on a set of qubits. This way of thinking was difficult to learn.
2. Adding gates to a Cirq circuit is less intuitive because the default addition method does not create a new Moment, but rather pushes the gates as close together as possible. This was different in PyQuil and made adding a column of gates to a circuit object difficult in the beginning.
3. Cirq leaves it to the programmer to provide the structure of the qubits in the system. It was difficult to learn of the different ways because they were spread throughout the documentation.

Three aspects of programming in Cirq that the language supports strongly

1. Simulation only requires a `simulation` object, and no extra running server (which was the case in PyQuil)
2. The python `print` feature gives reasonable output for circuit instances.
3. Gate creation supports python list compression, which is very handy when adding many of the same gate to a circuit.

Three aspects of programming in Cirq that the language supports poorly

1. Because cirq is not compiled in the same way as PyQuil, there is no way to look at the "assembly" code of your circuit.
2. The result returned by the simulator is slightly harder to work with than PyQuil. Rather than a list of arrays, it returns a `TrialResult` object. In order to turn this into a list of arrays (which is needed for some algorithms), we had to do a fair amount of manipulation of the data stored in this object.
3. Creating custom unitary matrices is not as well supported as in PyQuil.

Which feature would you like Cirq to support to make the quantum programming easier?

Cirq presents a great system to build a quantum circuit. However, the programmer is at a loss for optimizing and simplifying their circuit. It would be great for Cirq to introduce a way to convert a lengthy circuit into a simpler equivalent one.

List three positives of the documentation of Cirq

1. It includes a tutorial page.
2. It is direct and to-the-point.
3. It includes an examples page.

List three negatives of the documentation of Cirq

1. The example presented in the Cirq tutorial (variational method) is a bit overly complicated for introducing the language.
2. More examples should be inserted to demonstrate the functionality of each definition in Cirq. Most of the examples are just Cirq scripts (albeit well-commented) and not actual documentation pages.
3. Some of the important features of the language are not emphasized well-enough in the docs. For example the ControlledGate documentation was buried in API reference

In some cases, Cirq has its own names for key concepts in quantum programming. Give a dictionary that maps key concepts in quantum programming to the names used in Cirq

Cirq jargon	**QP jargon**
GridQubit	2D lattice topology
LineQubit	1D lattice topology
Repeats	Runs on Quantum Computer/Simulator
Moment	Vertical slice of operations on circuit
Operation	Gate applied to several qubits
InsertStrategy	How gates are ordered in circuit

How much type checking does the Cirq implementation do at run time when a program passes data from the classical side to the quantum side and back?

Cirq does type checking both when building a circuit and when running. Only gates or other circuits can be appended to a circuit. It is possible to append many gates at once through a list or generator of gates, but all objects must be legal Cirq gates. In addition to this, Cirq also checks that the qubits in these gates are legal. Any violation will result in a error

The `simulator.run()` will check that the parameter is a Cirq Circuit object, anything else will result in an error.

Cirq returns a `TrialResult` object with the results of a run. Cirq undoubtably does some checking behind the scenes as this object is being built to be passed back from the quantum to the classical side.