Four Algorithms in Cirq

John Gardiner, Joshua Kazdan, Frederick Vu

1 Design and Evaluation

1.1 The Oracles

The Deutsch-Jozsa, Berenstein-Vazirani, Simon, and Grover problem statements each involve an oracle gate which implements the unitary $U_f : |x\rangle |y\rangle \mapsto |x\rangle |y \oplus f(x)\rangle$ for some function f on bit strings x. In our implementation, however, we are not given the oracle directly, just the function f. That puts it upon us to build the oracle ourselves as a separate step, before implementing the algorithm. This was the least straightforward part of our task, as there are many ways to describe the same desired unitary gate. Different choices have different potential advantages.

Cirq makes it straightforward to create custom gates directly from their description as unitary matrices. We ultimately did not use this method, for reasons we will give below, but we describe it here anyway. The `cirq.MatrixGate()` function accepts a matrix in the computational basis and produces the corresponding quantum gate. The matrix for the desired oracle given above can be easily constructed by looping over integers from 0 to $2^n - 1$. For example, we can start with a $2^n \times 2^n$ array, `oracle_matrix`, of zeros then replace the entry in the (x, y)-th column and $(x, y \oplus f(x))$-th row by 1:

```
for x in range(2**n):
    for y in range(2**n):
        oracle_matrix[x+2**n*y][x+2**n*(f(x)^y)] = 1
```

The resulting matrix can then be passed into the MatrixGate function in Cirq. This implementation has several limitations, both in terms of memory and run time. It requires a classical computer to store a matrix of size $2^{n+1} \times 2^{n+1}$. Our computers struggled to store the required matrices even for relatively small numbers of qubits. The time for construction was also exponential in n. In Figure 1 we plot the construction time in the number of qubits, for one case we tried. Although exponential construction time is of course unavoidable in general when building an oracle from a function without special structure, it is certainly possible to produce the oracle with $O(K2^{n+1})$ space where $K \ll 2^n$. Whereas, with the original oracle, we already risk stack overflow with $n = 20$, which translates into a storage requirement of $4,294,967,296$

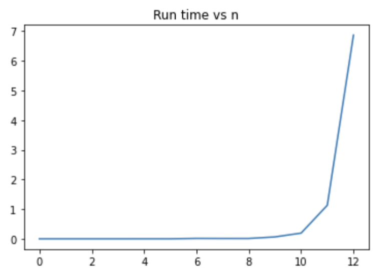


Figure 1: Run time versus number of qubits for the construction of U_f using MatrixGate.

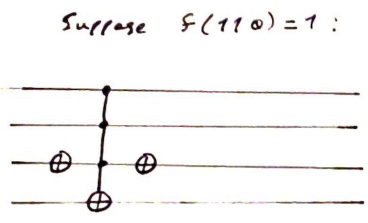


Figure 2: Example of the gate that would be included if $f(110) = 1$. In the case that $f(110) = 1$, this manifestly implements $|110\rangle\,|y\rangle \mapsto |110\rangle\,|y \oplus f(110)\rangle$.

integers. Finally, and perhaps most importantly, describing the oracle this way does not give a list of instructions that can directly be implemented by a quantum computer. We expect this to become salient when we port our cirq code to qiskit to run on IBM's quantum hardware.

So instead of using the above method we opted to construct the oracle out of common gates, specifically CNOT and single-qubit gates. We achieved this via two different methods. Both methods involve describing the action of the unitary with multi-controlled NOTs and then decomposing these CC...CNOT gates into elementary gates. We first describe how to implement U_f in terms of CC...CNOT gates. For simplicity consider a function f from strings of n bits to single bits. For each input $x = x_1 x_2 \cdots x_n$ that outputs 1, we add the gate $(X^{x_1} \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1})\,(\text{CC}\ldots\text{CNOT})\,(X^{x_1} \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1})$ where the target of the CC...CNOT is on the last qubit. For example a function on 3 bits with $f(110) = 1$ would include the pictured in Figure 2: We add such a gate for every x such that $f(x) = 1$. Note that all such gates commute. Together they manifestly implement $|x\rangle\,|y\rangle \mapsto |x\rangle\,|y \oplus f(x)\rangle$.

Our goal is to decompose the oracle into CNOTs and single-qubit gates. Unfortunately the expansion of CC...CX gates into these simpler gates is not simple. We use two methods to do this both with different advantages. The first method (which we will call the "Toffolify" method) follows the method described by [Gid] using one ancilla qubit at the first step of decomposition into three smaller $C^k X$'s. For odd n, these gates are all $C^{\lceil \frac{n}{2} \rceil} X$'s while for even n, they are 2 $C^{\lceil \frac{n}{2} \rceil} X$ and 1 $\lceil \frac{n}{2} \rceil + 1X$. The next step of the decomposition takes a $C^k X$ for and decomposes it into $4(k-2)$ CCX's using $k-2$ "ancilla" qubits that may be given in any state and are returned in their given state after the decomposition process. This step takes advantage of the first decomposition step. Finally, we decompose the Toffoli gates, of which there are $12(\frac{n}{2} - 2) + 6$ in the case of odd n and $12(\frac{n}{2} - 2) + 4$ in the case of even n. The choice of decomposition of the Toffolis depends on which quantum computer one intends to use. The default decomposition chosen was a factorization into a product of 5 2-qubit gates consisting of 3 CNOT's and 2 CNOT$^{1/2}$'s. For a balanced input function f for the Deutsch-Jozsa algorithm, this results in $(60(\frac{n}{2} - 2) + 30)2^{n-1}$ or $(60(\frac{n}{2} - 2) + 20)2^{n-1}$ 2-qubit gates for the cases of odd and even n, respectively, along with several Hadamard and Pauli X gates for each CC...CNOT. This method results in an abundance of CNOTs, leading to a very long circuit. One downside of this method is that it is oblivious to some simplifications, for example the fact that two multi-controlled NOT gates like

$$\left(X^0 \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1}\right)(\text{CC}\ldots\text{CNOT})\left(X^0 \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1}\right)$$

and

$$\left(X^1 \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1}\right)(\text{CC}\ldots\text{CNOT})\left(X^1 \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1}\right)$$

can be combined to get a single multi-controlled not gate with one less control:

$$(\mathbb{1} \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1})\,(\mathbb{1} \otimes \mathrm{CC}\ldots\mathrm{CNOT})\,(\mathbb{1} \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1})$$

whenever there are inputs of the form $0x_2x_3\ldots x_n$ and $1x_2x_3\ldots x_n$ both evaluating to 1. There are potentially many such simplifications for functions with much structure.

The second method we used to decompose the oracle into CNOTs and single-qubit gates was to use the formalism of *phase polynomials*. A phase polynomial is a unitary whose action is given by $|x\rangle \mapsto e^{ig(x)}|x\rangle$, where the function g is a linear combination of terms made up of XORs of bits of x. (See section 5.6 of [vdW] for a review.) For example, the most general phase polynomial for two qubits has a function g of the form $g(x) = \alpha x_1 + \beta x_2 + \gamma x_1 \oplus x_2$. A *phase gadget* is a phase polynomial where g has a single term. Phase gadgets are the building blocks of more phase polynomials, in the sense any phase polynomial can be easily decomposed into simpler phase gadgets. A phase gadget $|x\rangle \mapsto e^{i\alpha(x_{i_1}\oplus\cdots\oplus x_{i_k})}|x\rangle$ is characterized by the qubits it involves (here $x_{i_1},\ldots,\ x_{i_k}$) and the phase coefficient (here α). Phase gadgets have two particularly nice properties. First, they always commute with each other, and second, two phase gadgets on the same qubits can be combined to give a new phase gadget (whose phase coefficient is the sum of the original two). The importance of phase gadgets for our purposes is that the gates of the form $(X^{x_1} \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1})\,(\mathrm{CC}\ldots\mathrm{CNOT})\,(X^{x_1} \otimes X^{x_2} \otimes \ldots \otimes X^{x_n} \otimes \mathbb{1})$ that implement our oracle are nearly phase polynomials. Specifically, conjugating one of these gates by Hadamards on the last qubit results in a phase polynomial. This phase polynomial can be decomposed into phase gadgets, and the phase gadgets from each such gate can be combined, with the potential for cancellations when phase coefficients add to zero. Finally phase gadgets themselves have a known decomposition into the more familiar gates R_z and CNOT. This method allows us to decompose the multi-controlled NOTs implementing the logic of our function f into CNOTs and Z rotations in a way that catches cancellations like the one described in the previous paragraph. This gives a potentially more efficient construction of the oracle, at least for some functions. We call this method the "phase gadget" method. We used this phase gadget method for both Deutsch-Jozsa and Simon's algorithms.

The "Toffolify" and "phase gadget" methods discussed above work for arbitrary f. However, for some of the algorithms the given functions f have enough structure that we can easily create their oracles without using the above methods. For Bernstein-Vazirani (BV), for instance, we were able to construct the oracle out of a linear in n number of gates. Recall that for BV, the function is of the form $f(x) = a \cdot x \oplus b$. For each $i \in \{0, ..., n-1\}$, we check whether $a[i] = 1$, and if so, we add a CNOT gate where the control is on the ith qubit and the NOT acts on the auxiliary qubit.

Finally, in Grover's algorithm, we utilized the Toffolify method for constructing the oracle, as in the Deutsch-Jozsa algorithm. That is, to build the oracle for Grover's problem, we constructed a CC...CNOT using the Toffolify method (or more precisely, the `CrX_maker` method) for each instance that $f(x) = 1$.

1.2 Implementation Design

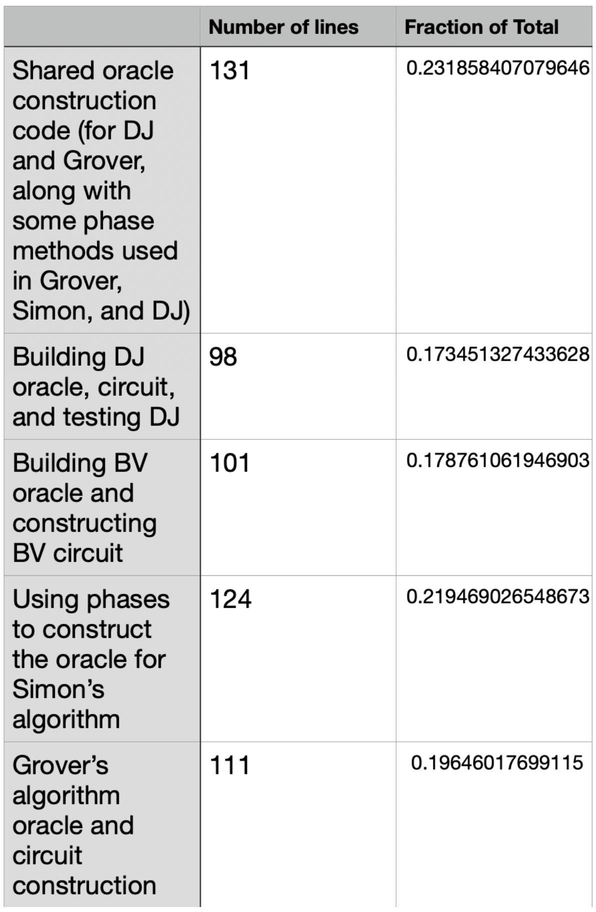
1.2.1 Grover's

There are some features special to our implementation of Grover's algorithm. In the original problem statement of Grover's algorithm, as presented on previous course assignments, there was only ever one such input x. However, a generalized version was discussed in the course, and our implementation is of this generalization which allows for more than one search target. We opted for the Toffolify method as opposed to the phase gadget method as the latter has a high chance of yielding exponentially many gates, on the order of $2n2^n$, whereas the number of elementary gates used in the Toffolify method would scale linearly with the number of search targets `a`. A complication arises in this more general case, as when a is close in value to 2^{n-1}, the probability that one finds a search target in one run of the usual Grover's algorithm, which is given by $\sin^2\left((2r+1)\arcsin\left(\sqrt{a/2^n}\right)\right)$ where r is the number of applications of the rotation operator, might require one to consider an uncomfortably high number of applications. In general, it is not straightforward how to go about maximizing the probability while minimizing the number of repetitions. The workaround in order to find and return all search targets we settled on use a while-loop that would find some of the search targets and create modified circuits during for each run.

This is done by observing that in the oracle U_f, if one has already found a subset $U \subset \mathbb{F}_2^n$ of inputs for which $f(x) = 1$, then one may consider the indicator function g_U for U and append the oracle U_g after U_f to obtain U_{f-g}. One can then run Grover's algorithm using this gate to search for the remaining search targets. To see that this composition of oracles yields U_{f-g}, one may observe that U_f may be built out of multiple CC...CNOTs which all commute with one another, and the CC...CNOTs in the constructed U_g will cancel out exactly the contribution of g to f, or alternatively, of U to $\{x|f(x) = 1\}$.

1.3 Engineering complexity

The different algorithms took very different amounts of work to implement. In the following table, we provide a breakdown of the number of lines required for each part of our code:



	Number of lines	Fraction of Total
Shared oracle construction code (for DJ and Grover, along with some phase methods used in Grover, Simon, and DJ)	131	0.231858407079646
Building DJ oracle, circuit, and testing DJ	98	0.173451327433628
Building BV oracle and constructing BV circuit	101	0.178761061946903
Using phases to construct the oracle for Simon's algorithm	124	0.219469026548673
Grover's algorithm oracle and circuit construction	111	0.19646017699115

We used the same general purpose oracle construction function for both Deutsch-Jozsa and Grover's algorithm. Our implementations of Bernstein-Vazirani and Simon's algorithms each had their own code for constructing their oracles. The oracle construction code for Simon's algorithm is a generalization of the phase gadget method code used to construct the DJ oracle.

Although we did not reuse the circuit construction code between different algorithms, we did reuse a lot of the complicated code for constructing oracles, which is encapsulated in the

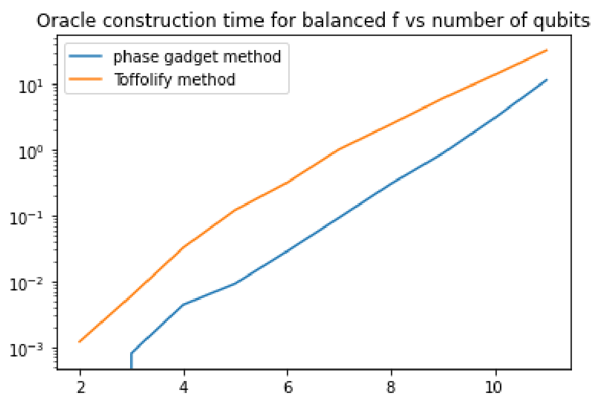


Figure 3: The average time it took to build oracles for balanced functions f versus the number of qubits. The yellow line represents the time of construction using the Toffolify method, and the blue line represents the time of construction using the phase gadget method. In separate tests, we found that both construction methods were faster than the method using Cirq's built-in matrix gate. Note the time for two qubits using the phase gadget method was not distinguishable from zero.

"Shared oracle construction code." We also reused the code for constructing the DJ oracle when building Simon's algorithm. The DJ oracle construction code is contained in the DJ code rather than the shared code for convenience.

The most intensive part of the project consisted of building the oracle constructors, since we decided not to use the built in MatrixGate functions. We also faced engineering challenges in order to avoid re-initializing the necessary Z_f and Z_0 gates with each pass in Grover's algorithm. Several speed-ups demanded extra lines of code too: for instance, we implemented machinery (discussed in the section on Grover's algorithm) that reduces the probability that the algorithm finds the same values twice. For this additional optimization we had to build another custom gate, in addition to the oracle. We used the general purpose oracle construction code we had written to do this.

1.4 Testing

1.4.1 Deutsch-Jozsa

In all test cases, we randomly initialized functions that met the requirements of the problems. In the DJ algorithm, we randomly selected half of the values in $[0, 2^{n+1} - 1]$ and assigned $f(x) = 1$ on these values to create a random balanced function. There are two parts of the algorithm that take up a significant amount of time. First, we build the oracle circuit using one of the two methods explained in Section 1.1, either the phase gadget method or the Toffolify method. The time this takes is plotted as figure 3. We performed trials for 2 through 11 qubits, finding that the time taken to prepare the oracle was roughly exponential in n for either method.

The second part of the program that takes up a significant amount of time is cirq's simulation of the resulting circuit. We measured the time it took to (classically) simulate the quantum circuit for $n = 2$ to $n = 12$ and for both the phase gadget and Toffolify methods. The results are found in figure 4. The simulation time for the quantum circuit appears exponential in the number of qubits. This is of course unsurprising, and reflects the speedup we'd expect from actually running a quantum circuit on a quantum computer.

To Summarize, we found that for both the pre-step of oracle preparation, and the actual simulation of the eventual quantum circuit, the phase gadget method gave faster results than the Toffolify method, though with the same scaling.

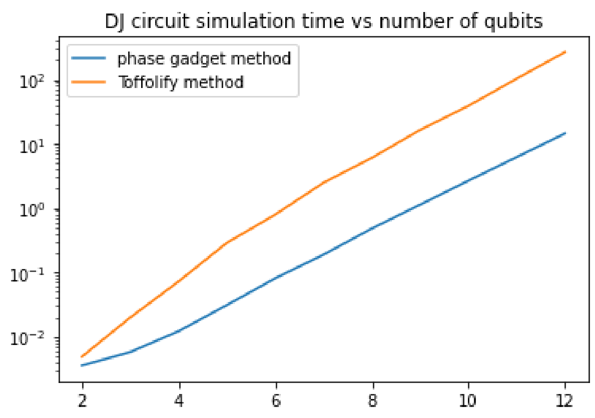


Figure 4: The simulation time differs for the circuit with an oracle constructed using phase gadgets versus a the Toffolify method. The two methods lead to circuits of different lengths as well as circuits whose simulation takes different amounts of time for cirq.

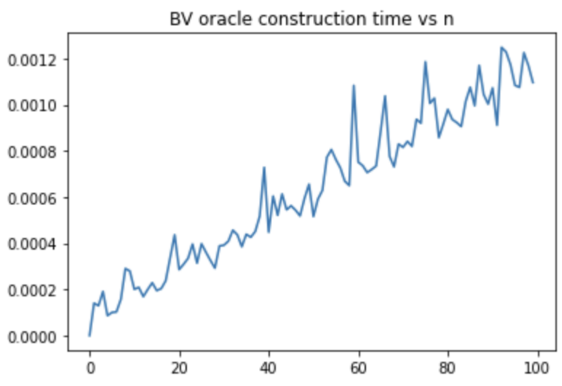


Figure 5: Oracle construction time versus the number of qubits. As expected, oracle construction for BV scales linearly with n.

1.4.2 Bernstein-Vazirani

We tested both the full runtime of our BV program as well as the runtime for just the construction of the oracle. Test functions of the form $f(x) = a \cdot x \oplus b$ were randomly generated by selecting $a \in \{0,1\}^n$ and $b \in \{0,1\}$. Because the function is simple, we used a simpler construction for the BV oracle than for the DJ oracle. We were able to construct it in linear time in n, as seen in figure 5. This is expected as our oracle for BV is made up of merely n or $n+1$ gates. Note, however, that the run time of our BV program is exponential. See figure 6 This is the expected slowdown from simulating a quantum circuit on a classical computer.

1.4.3 Simon's

To test our program for Simon's algorithm we generated random functions by the following method: First, we chose a random value for s from $\{0,1\}^n$. Then we chose a random map g from the set $\{0,1\}^n$ to itself. The bitwise XOR action of s on the set $\{0,1\}^n$ divides $\{0,1\}^n$ into pairs. Within each pair we consider the smaller of the two bitstrings, understanding the bitstrings as binary representations of integers. The value of g on the smaller bitstring is then what we call the value of f on either of the pair.

First we measured the time to construct the oracle circuit. We used the phase gadget method to do the oracle construction, as it had proved faster than Toffolify in tests of previous algorithms. The results are plotted in figure 7 Note the expected exponential relationship.

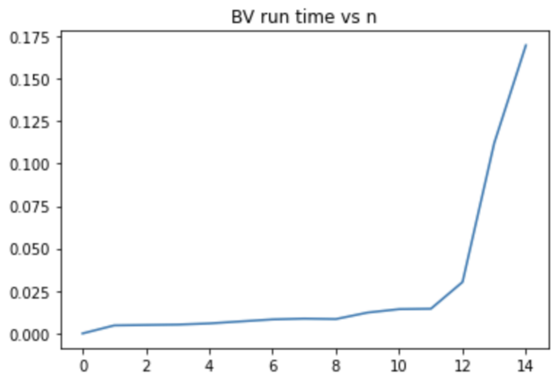


Figure 6: Runtime of BV versus number of qubits. As expected, the runtime of a classical simulation of BV is exponential in n.

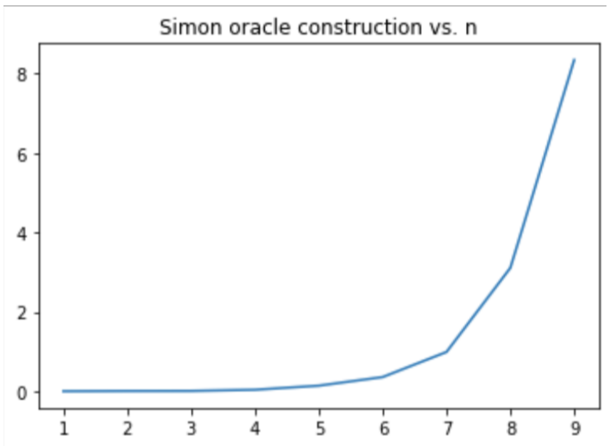


Figure 7: The average time taken to construct the oracle for a random function satisfying the problem statement for Simon's algorithm, as a function of n.

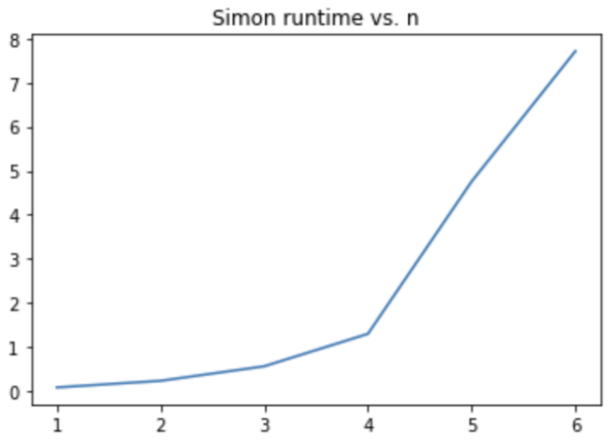


Figure 8: Simon run time versus n. One can again see that the run time scales exponentially with n. There is a point at 5 at which we measured an unexpectedly high run time relative to the $n = 6$ case.

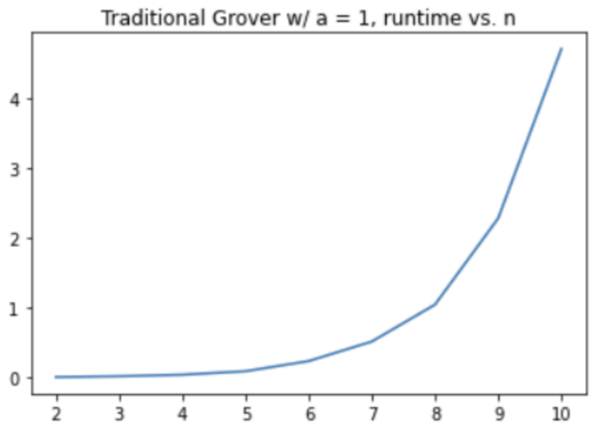


Figure 9: This graph shows the run time for Grover's algorithm where the algorithm must identify a unique element x on which $f(x) = 1$.

We also measure the runtimes for the complete algorithm for $n = 1$ to $n = 6$. The results in seconds are plotted in figure 8.

1.4.4 Grover's

We tested Grover's algorithm by randomly choosing one number x such that $f(x) = 1$. This yielded an exponential run time, also as expected. In addition to this we tested a more general version of Grover's algorithm in which there exist $2^{n-1} - 1$ elements x such that $f(x) = 1$. Our program was built to handle functions with any number of outputs being 1, and having $2^{n-1} - 1$ outputs being 1 is in a sense the "hardest case" our algorithm can handle. One can see from figure 9 and figure 10) that our implementation for Grover's algorithm with one input giving 1 runs significantly more quickly than when $2^{n-1} - 1$ inputs give 1.

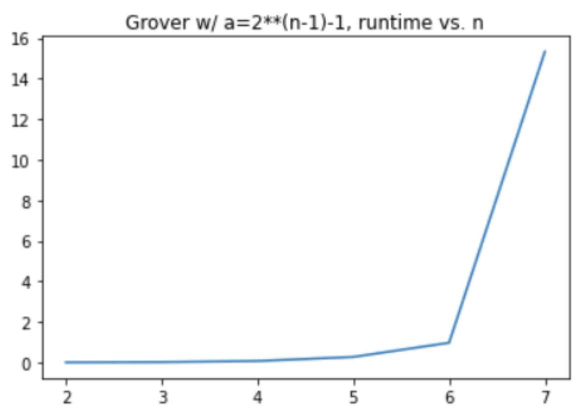


Figure 10: This graph shows a variant search problem where the function evaluates to 1 on $2^{n-1} - 1$ values.

2 Cirq

2.1 Ease of use and learning

We found the following three things in Cirq to be easy to learn:

1. The construction of the circuits themselves was straightforward. Large numbers of operations could be added to circuits with one line of code.

2. It was convenient that Google built in ways to initialize large numbers of qubits in parallel. The idea that qubits were objects that didn't need to be initialized and were organized either geometrically or together in "device" objects was intuitive and suggested a close connection to actual hardware.

3. Cirq has impressive functionality for modifying existing gates. For example, and gate can be used to define a controlled version of that gate with one method. As another example, gates can be taken to arbitrary powers to get new gates, all with simple notation.

We also encountered these three difficulty while trying to learn Cirq:

1. We initially tested whether Cirq could decompose custom matrix gates and custom controlled gates such as C^rX and found that it could not in both cases, using the `cirq.decompose()` method. Additionally, Cirq (currently) does not support the porting of a custom C^rX gate to QASM using `cirq.qasm`. It would be useful if Google created a function that automatically decomposed a C^rX gate into elementary gates for addition to the circuit. The oracle gate seems like such a fundamental gate that we found it surprising that we needed to put in a substantial amount of work to build a function to construct these gates from basic CNOT gates. In other words, we believe that Google should create a decomposition function, or at least include easy constructors for relatively simple gates such as control gates.

2. Currently, the method of building custom gates requires one to input a matrix in the form of a `NumPy.ndarray` into a gate constructor function. These matrices can quickly become too large to fit in memory, which we discovered while building the DJ algorithm. Oftentimes, despite requiring very large matrices to describe, these gates have simple descriptions and are very sparse. We believe that, in line with our first comment, Google should provide greater support for commonly used gates without requiring the user to build them from matrices, or to utilize alternative packages such as SciPy that can better handle sparse matrix operations in the simulation of quantum circuits.

3. There are, in several cases, different ways to create the same gate. It was not always clear what the best way was, or which ways were compatible with additional modifications of a gate, like powers and controls.

2.2 Documentation

We list three positive aspects of Cirq documentation that we found while working on the project. The most useful aspect was the availability and content of the tutorial pages. This allowed us to quickly get off the ground to implement the most basic versions of the four algorithms that we had in mind, before we attempted to improve the oracle construction method and the generalized Grover's algorithm. Another positive aspect of the documentation was the flexibility in the names of types of Cirq objects, i.e,, many objects had many common aliases that could be used interchangeably. For example, the ability to call `cirq.CX()` using `cirq.CNOT` is one basic example, but perhaps a more useful example is the ability to exponentiate the `cirq.CX` gate in order to apply a `cirq.CNotPowGate`. A final positive aspect could be the thoroughness of the implementations of more advanced topics in quantum programming. A quick skim of the defined operations in `cirq.ops` suggests that there may be many useful tools that one could use for future projects.

Perhaps the most frustrating aspect of the Cirq documentation is the lack of examples on many of the pages for common object types. For example, it is not clear from looking at the documentation page for the `cirq.sim.Simulator` class how one would use the `simulator` or `run()` methods. Figuring out how these worked precisely required us to use several print statements and calls of the built-in `type()` method of Python. A second negative aspect is the general ad hoc naming and design conventions of many of the objects and method. For example, the methods available to `cirq.sim.Simulator` contain a plethora of highly case-specific names, and the objects that are returns by each are manifold. One final negative aspect of the documentation is that it is difficult to find solutions to problems that one might naturally ask, as we discovered while working on the project. We were forced to conclude that no solutions existed or were available through Cirq. This last point might be better phrased as saying that Cirq lacks an FAQ page to explain its current capabilities.

References

[Gid] C. Gidney. Constructing Large Controlled Nots, available at `https://algassert.com/circuits/2015/06/05/Constructing-Large-Controlled-Nots.html`

[vdW] J. van de Wetering, *ZX-calculus for the working quantum computer scientist*, 2020.