QAOA and Shor in Cirq

John Gardiner, Joshua Kazdan, Frederick Vu

1 Shor's Algorithm

1.1 Design

Our implementation of Shor's follows the circuit designed by Beauregard in [B] in 2003 which utilizes only $2n+3$ qubits to factor N, where $n = \lceil \log_2 N \rceil$, with a circuit depth of $O(n^3)$ and a gate count of $O(n^4)$. We chose this design as it uses a relatively low number of qubits, though we mention that it is not currently the best algorithm in this regard - this distinction is shared by many circuits which only use $2n+2$ qubits such as those found in [TK] from 2006 and [HRS] from 2017. We are primarily concerned with reducing the number of qubits needed in Shor's algorithm as this heavily affects the simulation runtime. There are two main components of Shor's algorithm: the gate which implements modular multiplication in the computational basis, and the inverse quantum Fourier transform which allows for one to estimate the phases or eigenvalues of said gate. There are opportunities to reduce the qubit count in both halves, though the optimization of the latter is well-understood, and there seems to be no room for improvement. We first discuss the modular multiplication gates.

We elected to construct the circuit that represents modular multiplication explicitly as opposed to utilizing certain functionalities available to us in Cirq such as the `ArithmeticOperation` helper class or the `MatrixGate()` method (one could use `MatrixGate()` to construct the permutation matrix on the first N basis states associated with multiplication of $(\mathbb{Z}/N\mathbb{Z})$ by $a \in (\mathbb{Z}/N\mathbb{Z})^\times$) as these cannot currently be exported to QASM via methods available in Cirq. Additionally, it seemed to go against the spirit of the assignment. We briefly explain now how modular multiplication gates, which for a given $a \in (\mathbb{Z}/N\mathbb{Z})^\times$ sends $|x\rangle \mapsto |ax\rangle$, are constructed in our circuit.

The modular multiplication gates are built up out of modular addition gates, and these are constructed using regular addition gates (also called adders). The gates utilize two registers R_1 of n qubits and R_2 of $n+1$ qubits. We mention that these arithmetic gates all have controls for the purposes of Shor's algorithm, though below the controls may not be mentioned in each description as they may not be pertinent to the arithmetic actions of the gates themselves.

In Shor's algorithm, one identifies a unit $a \in (\mathbb{Z}/N\mathbb{Z})^\times$, where $1 \leq a \leq N-1$, classically. In the adder gate `phaseAdder`, the second register R_2 is utilized which is assumed to be already in the Fourier basis, i.e., that a quantum Fourier transform (QFT) has already been applied at an earlier step in the circuit and that it holds, say, the integer b modulo N in the phase of the state. The addition gate is implemented by applying rotations about the Z-axis of the Bloch sphere of each qubit using the predetermined a. The reason for the need for $n+1$ qubits in R_2 is to account for possible overflow of the value $a+b$ (that is, it might be that $a+b \geq 2^n$). Notationally, the adder gate sends $|\phi(b)\rangle \mapsto |\phi(a+b)\rangle$ where ϕ is used to signify the Fourier basis. It should be mentioned here that while n was defined to be $\lceil \log_2(N) \rceil$, the adder gate works generally outside the context of Shor's algorithm for any a, b, n with $0 \leq a, b < 2^n$. The gate count $n \in O(n)$ and the gate depth is $1 \in O(1)$ without controls, but as used in Shor's algorithm, the depth is $n \in O(n)$.

The modular addition gate `modPhaseAdder` is also implemented in the Fourier basis. Here, the gate utilizes R_2 as well as an ancilla qubit that is used to aid in handling overflow. This is the named **`ancilla`** qubit in the code. The gate takes an input state $|\phi(b)\rangle|0\rangle$ where $0 \leq b < N < 2^n$ is an integer stored in in the Fourier basis in R_2 and the final $|0\rangle$ is the ancilla, and returns the state $|\phi(a+b \bmod N)\rangle|0\rangle$. Again, a, N, n are allowed to be any integers such that $0 \leq a < N < 2^n$ in general. As QFT is used to allow for the assessment of overflow, the gate count is $O(n^2)$ and the depth, with or without controls, is $O(n)$.

The modular multiplier gate **`modMultiplier`** is unfortunately named, as it is *not* quite the modular multiplication gate discussed in the introduction on which one would perform phase estimation as in Shor's algorithm. It is implemented outside of the Fourier basis, but of course uses the Fourier basis as it is built out of `modPhaseAdder`'s. It utilizes both registers as well as the necessary ancilla qubit used in `modPhaseAdder` and sends the state $|x\rangle|b\rangle|0\rangle \mapsto |x\rangle|b+ax \bmod N\rangle|0\rangle$ for $0 \leq a, b < N$. It is built out of a ladder of controlled `modPhaseAdder`'s with classical inputs $a2^i \pmod N$ controlled by the qubit corresponding to bit x_i of x. This of course is motivated by the observation that

$$ax \bmod N = (\ldots((a2^0x_0 \mod N) + a2^1x_1) \mod N) + \ldots + a2^nx_n) \mod N.$$

This gate again only requires that $0 \leq a, b, x < N < 2^n$. As **`modMultiplier`** is simply n `modPhaseAdder`'s, the gate count is $O(n^3)$, and the circuit depth is $O(n^2)$.

Finally, the "multiplication-by-a-gate", denoted by U_a, acts on R_1, which is considered the target register, but also utilizes R_2 as an ancillary register along with the standard `ancilla` qubit. This is the gate whose phase is to be estimated as in Shor's algorithm. It operates with all ancillae initialized to the 0 state, and sends $|x\rangle|0^{n+1}\rangle|0\rangle \mapsto |ax\rangle|0^{n+1}\rangle|0\rangle$ for $0 \leq x < N$. This gate requires that $a \in (\mathbb{Z}/N\mathbb{Z})^\times$ as is clear from its construction as a composite of `modMultiplier(a)`, `SWAP(R1,R2)`, and `modMultiplier(ainv)`$^{-1}$, which act by

$$|x\rangle|0^{n+1}\rangle|0\rangle \mapsto |x\rangle|ax \bmod N\rangle|0\rangle \mapsto |ax \bmod N\rangle|x\rangle|0\rangle \mapsto |ax \bmod N\rangle|0^{n+1}\rangle|0\rangle.$$

The final mapping is due to the inverse `modMultiplier` gate, which subtracts from R_2 rather than adds. This gate utilizes $2n+2$ qubits with a gate count of $O(n^3)$ and depth of $O(n^2)$.

We should address the readers' possible concern at what seems to be a startlingly high qubit count requirement for the modular multiplication gate. Recall that the our intentions were to achieve a low qubit count in an implementation of Shor's algorithm to facilitate its simulation on a classical computer. The qubit saving measure happens in the inverse QFT phase of Shor's algorithm, as we now explain.

It was first pointed out in [GN] in 1996 by R. Griffiths and his student C.S. Niu that QFT may be implemented using only single-qubit gates with classical controls (and so similarly for the inverse QFT), whence the name semiclassical QFT. This was explained using the consistent histories interpretation of quantum mechanics - originated in part by Griffiths - which "allow[s] one to use the results of a measurement to infer the state of a quantum system prior to the measurement." More formally, this can be explained by observing that the projection-valued measure is invariant under the application of the controlled gate, which is to say the probability distribution of the possible measured values of the QFT and the semiclassical QFT are equivalent. From this observation, one can deduce that only a *single control qubit* suffices to implement phase estimation, and so similarly for Shor's algorithm, which is a large improvement over the standard $2n$ control qubits used in a more typical formulation. This explains the $2n+3$

qubit implementation of Shor's algorithm as described in [B] as well as the gate count of $O(n^4)$ and circuit depth of $O(n^3)$ stated in the introduction of this report.

In Cirq, it is straightforward to build the U_a gate. However, for the semiclassical QFT, some more work has to be done as it turns out that the most recent stable release of Cirq, version 0.13.1, does not support classically controlled gates. We discovered that in December 2021, support for classically controlled gates was added to the Cirq repository on GitHub and is available on pre-release versions of Cirq 0.14.0. We were able to successfully implement semiclassical QFT both on Cirq 0.13.1 and on Cirq 0.14.0. The former implementation was accomplished by constructing and simulating a new circuit $2n$ times using a for-loop. Measurements were recorded outside of the loop for use in determining the appropriate gates for the next circuit, and final state vectors of each simulation were passed to the next circuit.

1.2 Testing

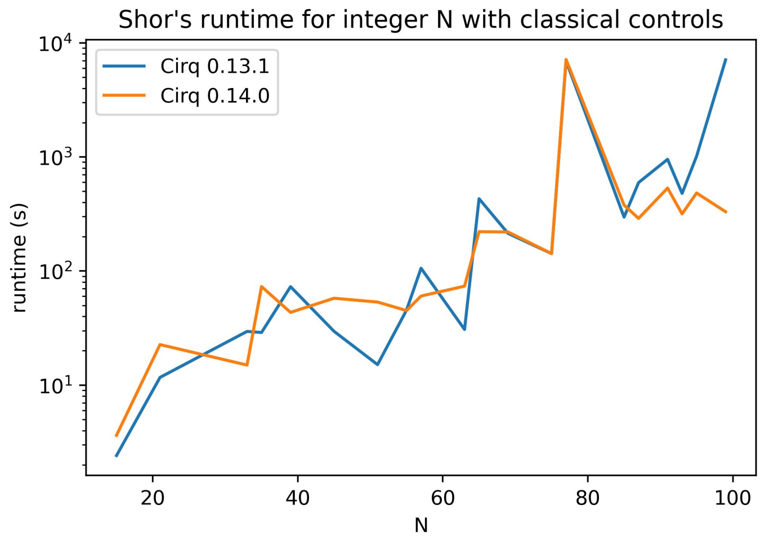


Figure 1: This graph shows the runtime for one trial each of the two implementations of Shor's algorithm for integers between 1 and 100, excluding those that were processed without the use of the quantum subroutine.

We tested both `Shor()` and `Shor2()`, the latter of which requires Cirq 0.14.0, on the integers 1 through 100. The testing took a rather significant amount of time, and so we only tested each once. There is a high degree of chance that goes into these runtime tests, as it may be that the random element $a \in \mathbb{Z}/N\mathbb{Z}$ chosen shares a nontrivial common divisor with N, and so the methods would not entire the quantum sub-routine. Additionally, the quantum sub-routine may need to be run several times if it turns out that the order r of $a \in (\mathbb{Z}/N\mathbb{Z})^\times$ is not even, or that $a^{r/2} \equiv -1 \bmod N$. Evidence of this randomness can be seen in the large spikes of Figure 1 for the higher values of N. Effects of randomness could be suppressed had we attempted more trials. However, the total runtime for the two methods were 308 minutes for `Shor()` and 175 minutes for `Shor2()`. We believe this large discrepancy between the two methods to be due almost entirely to chance, as the single attempt to factor 99 took `Shor2()` only 5.5 minutes while it took `Shor()` nearly 2 hours, and for all other inputs, the differences in runtime were relatively small. Additionally, the circuits of the methods are identical in design, and so we expected both methods to perform similarly. If anything, we would expect `Shor2()` to perform better due to the likelihood of Cirq's implementation of classical controls being more efficient than the ad hoc method employed in `Shor()`. The runtime grows exponentially with the value of n, as expected for a simulation on a classical computer. For comparison with $N = 99$ and $n = 7$, both methods were able to factor $N = 15$ quickly, taking 2.4 seconds for `Shor()` and 3.6 seconds for `Shor2()`.

2 QAOA

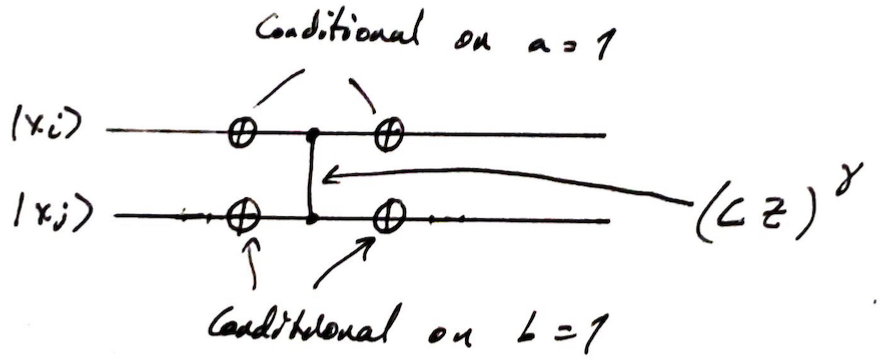
2.1 Circuit construction

Recall that the QAOA algorithm accepts a series of logical statements and returns a bit-string that satisfies a large number of these logical statements with high probability. We implemented the special case of QAOA in which all of the logical statements are in one of the following forms:

$$(x_i \vee x_j), (\neg x_i \vee x_j), (\neg x_i \vee \neg x_j).$$

The logical statements ("phrases") that we wish to satisfy are parameterized by tuples ("clauses") (a, b, c, d) where $a, c \in \{0, 1\}$ and $b, d \in \{0, ..., n-1\}$. The integer n denotes the number of variables allowed in a given phrase. The clause corresponding to the tuple (a, b, c, d) is $(x_b = a) \vee (x_d = c)$. We also allow for clauses of the form (a, b), which we take to mean $(x_b == a) \vee (x_b == a) = (x_b == a)$. The algorithm also accepts step sizes for β and γ that are used in the optimization. The overall phrase looks like $c_1 \wedge c_2 \wedge \cdots \wedge c_m$ where each c_i is a clause.

We now describe the construction of the mix and sep gates based on the series of logical statements. We construct the sep gate clause by clause. For each clause (a, b, c, d) in the phrase, we build the following gate where x_i corresponds to b and x_j corresponds to c:



It is not difficult to see that this circuit corresponds to the diagonal 2×2 matrix M in the basis for $|x_i x_j\rangle$ with diagonal entries $e^{i\gamma}$ when (x_i, x_j) fails to satisfy the clause and 1 otherwise. Multiplying M by $e^{-i\gamma}$, $e^{-i\gamma}M$ is an operator that shifts $|x_i x_j\rangle$ by $e^{-i\gamma}$ if and only if (x_i, x_j) satisfies the corresponding clause. The effects of a global phase change are not measurable, allowing us to assume that $e^{-i\gamma}M$ and M are equivalent operators in the circuit.

In order to build the mix gate, we simply add an $X^{-\beta}$ gate to each of the qubits in the circuit.

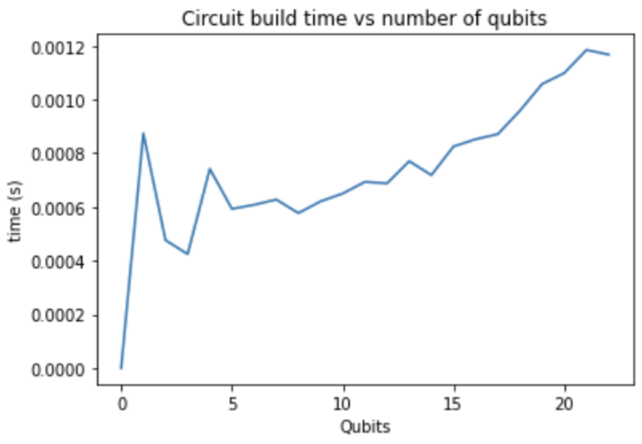
2.2 Optimization

Initially, we implemented the naive optimization algorithm for QAOA, which consists of initializing (β, γ) at regular intervals over $[0, 2\pi] \times [0, 2\pi]$. We used various step sizes and affirmed that smaller step sizes (implying a finer lattice on the domain) led to better results in terms of the number of clauses satisfied. However, as discussed in the next section, we discovered that this was not any better than randomly guessing an equivalent number of random bit strings.

We realized that in order to harness QAOA effectively, we needed to make use of the optimization landscape that it provides. Without QAOA, we face a discrete combinatorial optimization problem that does not lend itself to gradient descent. We subsequently implemented a gradient descent optimization algorithm. We first choose a random point $(\gamma, \beta) \in [0, 2\pi) \times [0, 2\pi)$. We then test 8 points evenly spaced around a circle of radius radius `step_size` centered at the randomly initialized point. At each of these eight points, we run QAOA 5 times and average to compute an approximate expectation. If one of these eight points has a higher expectation than the randomly initialized point, we recenter the circle at this new point and repeat the process. The benefits of this optimization process are difficult to see when the number of qubits and clauses are relatively low. However, one can see that after several steps, this algorithm begins to give better (on average) than random initialization.

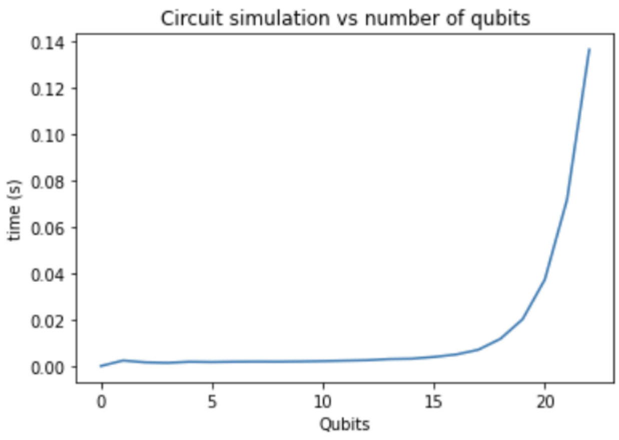
2.3 Testing

We first tested the construction time of the circuit for QAOA on n qubits using only two clauses. We constructed the circuit five times with randomly generated phrases and averaged the run time. This produced the following graph:

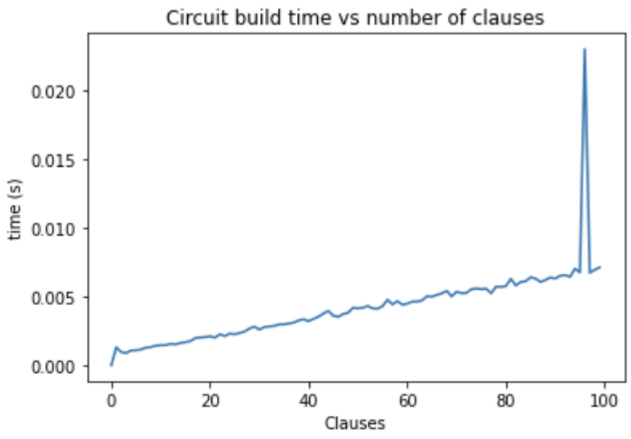


The run time appears to grow roughly linearly with the number of qubits, with a few unusual spikes. The number of qubits corresponds, of course, to the number of variables in the phrases that we wish to satisfy. The trend in the run time is consistent with the fact that to build the circuit, one only needs extra wires when extra qubits are added, which results in roughly linear time increases.

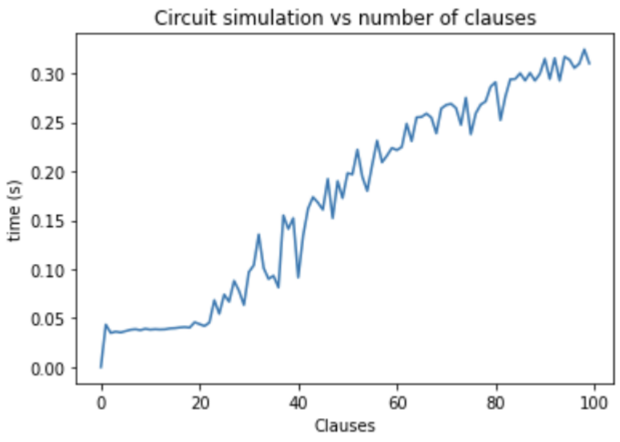
Once again, we initialized 2 phrases on n variables randomly and measured the simulation time. We repeated this experiment five times and averaged the results. Unlike the circuit construction time, the simulation time grows exponentially with the number of variables allowed in our logical statements. This is consistent with the run time trends that we observed when simulating "four algorithms" in our last homework. A classical computer requires exponentially more time to handle linear increases in the number of qubits.



We now test how the circuit construction time and simulation time grow when we increase the number of clauses in a given phrase. We fix the number of variables at 20 and increase the number of clauses from 0 to 100.



As expected, the circuit construction time grows linearly in the number of clauses. There is an unexpected spike near 95 that seems to be anomalous. We expect that the unusual spike was caused by slowdowns due to background processes running on the machine during simulations.

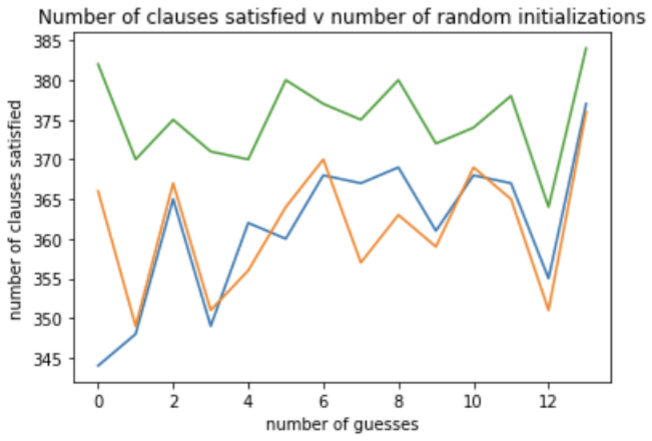


The simulation time also appears to increase roughly linearly in the number of clauses: there appears to be a significant amount of noise in the run time. This trend in the run time is expected, since the number of clauses corresponds to the length of the circuit.

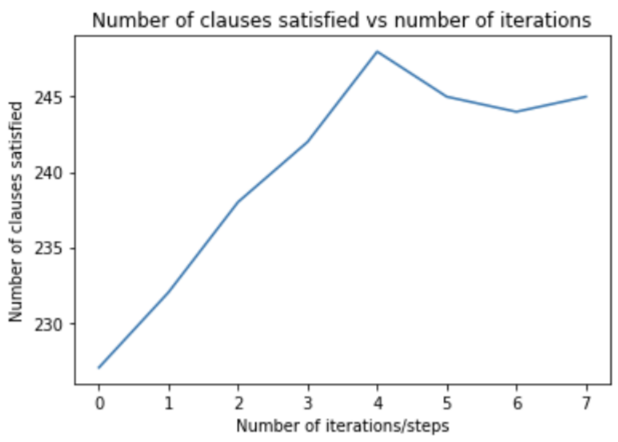
On a genuine quantum computer, the time savings occur because the simulation time would no longer increase exponentially in the number of qubits. The quantum simulation and construction times would likely demonstrate the same trend with the respect to the number of clauses.

We next test the efficacy of the optimization process. We perform the following experiment. First, we initialize a random phrase consisting of 30 clauses on 10 variables. We then compute the maximum number of clauses that can be satisfied by testing all possible bitstrings classically. Finally, we measure the number of clauses satisfied by the best solution generated by randomly selecting k values for β and γ. We measure the number of clauses that QAOA manages to satisfy and the number that it is possible to satisfy. This experiment gives us a sense of how many iterations of QAOA are necessary using randomly initialized variables in order to get an optimal result.

The following graphs the number of clauses satisfied against the number of iterations of QAOA performed. That is, we randomly choose k values of γ and β for a given phrase and check how many clauses are satisfied. As a control, we also chose random bit strings and checked the number of clauses satisfied. The green line represents the maximum number of clauses that can be satisfied (there were 500 clauses total). The blue represents the number of clauses satisfied by the result of iterating QAOA randomly k times, and the orange line represents the number that could be satisfied by the best of k randomly initialized bit strings. We see that without some directed optimization strategy, QAOA is not really any better than randomly choosing vectors.



We therefore implement the gradient descent algorithm mentioned in Section 2.2. This pseudo gradient descent algorithm has benefits that are not well-observed with a relatively small number of phrases and variables. We run a final test with 500 clauses on 10 variables. However, one can still see that after several steps, the gradient descent algorithm gives on average better results than simply choosing random initialization points. The maximum number of clauses that could be satisfied was 248 in this example.



We see that there is an upward trend in number of clauses satisfied by the gradient descent algorithm with step size 0.2.

References

[B] Stéphane Beauregard. Circuit for Shor's algorithm using 2n+3 qubits. Quantum Information & Computation, 3(2):175–185, 2003.

[HRS] T. Häner, M. Roetteler, and K. M. Svore. Factoring using 2n + 2 qubits with Toffoli based modular multiplication. Quantum Information & Computation, 17(7–8):673–684, 2017.

[GN] Robert B Griffiths and Chi-Sheng Niu. Semiclassical Fourier transform for quantum computation. Physical Review Letters, 76(17):3228, 1996.

[TK] Yasuhiro Takahashi and Noboru Kunihiro. A quantum circuit for Shor's factoring algorithm using 2n+2 qubits. Quantum Information & Computation, 6(2):184–192, 2006.