A Quantum Approach to Recognizing Context Free Grammars

Sophia Kolak
sdk2147@columbia.edu
Columbia University, New York

ABSTRACT

Context free grammars (CFGs) are a formal notation for expressing classes of languages, with applications in compiler technology and natural language processing. While generating strings using the rules of a grammar is intuitive, working in the opposite direction is hard. The best classical algorithm for deciding if a grammar recognizes a string has cubic run-time in the length of the string. In this paper, we utilize quantum computing to present an alternate approach to checking string membership in a context free language. We implement this approach for a few simple cases as proof of concept, and provide the necessary conditions for this method to produce a run-time improvement over the classical solution in the general case.

1 INTRODUCTION

A context free grammar (CFG) is a finite set of variables and production rules for constructing recursive languages of strings. CFGs, and the languages they generate, have many important applications in the parsing of programming languages, natural language processing and other data handling systems.

The grammar itself may be composed of two parts: variables, which represent languages, and terminals, which are primitive alphabet symbols. To illustrate how these grammars work, observe the CFG, which defines a type of valid sentence construction in the English language over its standard alphabet $\Sigma = \{a, b, ..., z\}$

$$\begin{aligned} <sentence> &\rightarrow <nounphrase><verbphrase> \\ <nounphrase> &\rightarrow <adjective><nounphrase> \\ &\quad | <adjective><singularnoun> \\ <verbphrase> &\rightarrow <singularverb><adverb> \\ <adjective> &\rightarrow a|the|little \\ <singularnoun> &\rightarrow boy|girl \\ <singularverb> &\rightarrow ran|swam|jumped \\ <adverb> &\rightarrow quickly|quietly \end{aligned}$$

Using the production rules of a CFG, we can generate strings in the context free language (CFL) it defines. For instance, "a boy ran quickly" or "the girl swam quietly" are both recognized by this grammar.

In practice, however, algorithms are often more concerned with going in the opposite direction. For instance, we might want to start with the sentence "the little boy jumped quickly" and ask if it is grammatical or not.

Interestingly, but perhaps unsurprisingly, this is a much more difficult problem. In fact, the best Big-O run-time for any classical approach is the CYK algorithm, which incurs a cost of $O(|G|n^3)$, where n is the length of the string, and G is the size of the grammar's alphabet.

Why exactly is checking whether a certain string is well-formed with respect to some grammar so computationally expensive? The answer lies in the large space of potential sub-strings and partial parse-trees that any derivation of a string could generate. Even the most clever dynamic programming based approaches cannot avoid having to consider all potential sub-strings, since they inherently require building up a complex string from the bottom.

An alternative approach is to go from the top down: to start by generating potential final strings of a given length n, removing the ones which violate the rules of the grammar, and then searching through the resulting strings for the target. Unfortunately, do this classically would cause the search space to undergo an exponential blowup of a^n, where a is the alphabet of the grammar, making it substantially worse than the CYK algorithm.

While classically, this brute-force method is infeasible, in the quantum realm we may be able to achieve a practical time complexity. By using a variation of Simon's algorithm, where the oracle function is replaced with a new function $f(x)$, which maps $\forall S \notin L \rightarrow S \in L$, we propose a method which would significantly improve the run-time necessary for deciding if a given string s is recognized by a grammar G, so long as the initial mapping is also efficient.

Contributions:
(1) We propose a simple quantum algorithm for solving the grammar membership problem.
(2) We show that, for specialized cases, this approach provides a polynomial speed up in the time necessary for checking grammar membership.

(3) For these specialized cases, we implement quantum circuits that produce the desired output in simulation.
(4) We provide a brief discussion of the necessary conditions for extending this algorithm to the entire class of CFGs

2 PROBLEM

We formally state the problem of checking grammar membership as follows:

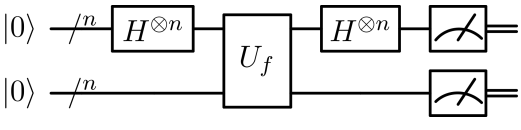
Problem. *Given some string s, |s| = n, and some CFG 'G' check if G generates s*

At a high level, the quantum algorithm for solving this problem can be described in three steps. First, generate all possible strings of length n by putting n qubits in superposition. Then, apply some function f which "removes" the possibility of measuring ungrammatical strings. This effectively generates all strings in the grammar of length n VIA measurement. The final step is to sort the resulting strings using merge-sort in $nlog(n)$ time, and then to binary search through the strings in $log(n)$ time for the string s.

While this algorithm maps naturally onto CFG's with a restricted alphabet (i.e., an alphabet of only two symbols), it is more difficult to consider how it works when there is no longer a one-to-one mapping from alphabet symbols to qubit states. Additionally, it is complicated to consider how this problem changes when the grammar only generates the empty string at a given length. Thus, for the majority of this paper, we will consider only grammars over the binary strings which are non-empty at every n. We can now redefine our problem as such:

Simpler Problem. *Given some string s, |s| = n, and some CFG 'G' with alphabet* $\Sigma = \{0, 1\}$ *that is non-empty at every n, check if G generates s*

To approach the restricted problem, we build on the design of Simon's algorithm, which solves the problem of finding a hidden bit-string $s \in \{0, 1\}$ that satisfies $f(x) = f(y)$ iff $y = x \oplus y \ \ \forall x \in \{0, 1\}$. The circuit for Simon's problem is shown below.



By first applying the Hadamard gate, the circuit puts n bits in superposition, creating a state-space of size 2^n. Then, VIA the oracle function U_f, the qubits which are not in superposition are altered to produce only outputs consistent with the oracle function, so that when re-apply the Hadamard gate and then measure the second register, only bit-strings which satisfy the condition $f(x) = f(y)$ iff $y = x \oplus y \ \ \forall x \in \{0, 1\}$ are observed. From here, post-processing is performed to recover the true value of the secret string s.

Using this general principle, we can extend Simon's algorithm to work as a context free grammar recognizer.

Consider how we could do this for the palindromes of length $n = 2$ defined by the CFG

$$P \rightarrow 0|1|1P1|0P0|\epsilon$$

We start, as in Simon's algorithm with $2 \cdot n$ qubits. We then put the first n qubits into superposition with a Hadamard transformation.

$$\psi_0 = |00\rangle_1|00\rangle_2$$

$$\psi_1 = \frac{1}{2}(|00\rangle_1 + |01\rangle_1 + |10\rangle_1 + |11\rangle_1)|00\rangle_2$$

Then, we apply some function f which alters the value of the last n qubits based on the bit-values of the strings in superposition. As a result of f, we create the following states:

$$\psi_2 = \frac{1}{2}(|00\rangle_1|00\rangle_2 + |01\rangle_1|00\rangle_2 + |10\rangle_1|11\rangle_2 + |11\rangle_1|11\rangle_2)$$

Now when when we measure the second register, we have a $\frac{1}{2}$ chance of seeing 11, and a $\frac{1}{2}$ chance of seeing 00.

In other words, with equal non-zero probability we will measure precisely the strings generated by our grammar of length n. Thus, once we've run some number of shots and performed our measurement, the only work remaining is to check if s matches any of the measured strings. Assuming the function f exists and can be computed efficiently, the quantum circuit provides a polynomial speed-up in the amount of time it takes to check grammar membership.

3 GENERALIZED PALINDROME CIRCUIT

While the algorithm clearly works in the case where $n = 2$, does it work in general for larger n? And what exactly is the nature of the function f which brings the state of our system from ψ_2 to ψ_3?

To answer this question, we will delve deeper into the structure of f. What the function really does is map $s \notin L \rightarrow s \in L$. To see this in more detail, examine the truth table for $n = 2$

1	2
00	00
01	00
10	11
11	11

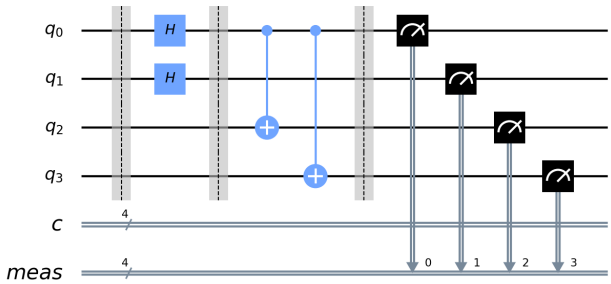
The table defines a two-to-one function, where all palindromes map to themselves, and all non-palindromes map to whichever palindrome shares its most significant bit. This

can easily be implemented with two *CNOT* gates from the most significant bit to each of the bits in the lower register.

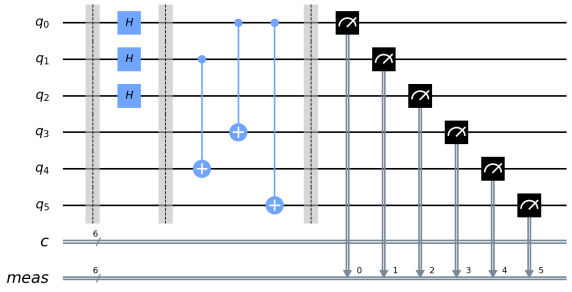
For $n = 3$, we can define a similar mapping, this time based on the two most significant bits.

1	2
011	010
010	010
001	000
000	000
111	111
110	111
101	101
100	101

Here we once again have a two-to-one function, where the left column of the truth table represents the state of the first register after the Hadamard transformation, and the right column of the truth table represents how the qubits in the second register should be changed accordingly. This truth table can be implemented using only 3 CNOT gates. We can also visualize the implementation using Qiskit.



For $n = 3$, the circuit is almost identical, with the addition of one more CNOT gate. Comparing the $n = 3$ circuit to its $n = 2$ counterpart suggests the number of gates required for palindrome checking scales linearly with the length of the string.

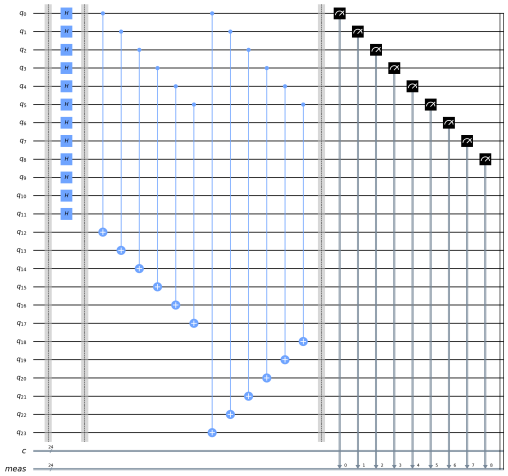


The function shown below, which automatically generates the palindrome checking circuit for any n, confirms the suspected linear growth. For this CFG, the circuit can always always be implemented with n CNOT and Hadamard gates.

```
def make_palindrome_circuit(n):
    num_qbits = 2*n
    circuit = QuantumCircuit(num_qbits,num_qbits)
    circuit.h([i for i in range(n)])
    control, response = [], []
    sig_bits = math.ceil(n/2)
    end = num_qbits - 1
    for i in range(sig_bits):
        control.append(i)
        response.append(i+n)
        if i == sig_bits - 1 and n%2 != 0:
            continue
        control.append(i)
        response.append(end-i)
    circuit.cx(control, response)
    circuit.measure_all()
    return circuit
```

The function runs in linear time, showing that the oracle function can be significantly more efficient than the classical algorithm, however, it does not mean that this will be the case for all CFGs.

The fact that a simple pattern does exist in this case, however, is still quite interesting. Another interesting aspect of this circuit, is that it can be intuitively constructed using only CNOT gates. These CNOT gates also follow a clear pattern as we increase the size of n. This pattern is made most apparent when n is large, for instance the maximally sized circuit on Qiskit with $n = 12$ illustrates the pattern of CNOTs, which always go from the first half of the first register (the significant qubits) to the bottom-most qubit in the second register, incrementing through the SQBs until each qubit has a CNOT, and then doubling up on CNOTS in the same fashion until every qubit in the second register has a CNOT.



Thus, we have shown that a polynomial improvement in run-time over the classical grammar membership algorithm exists for the special case of the binary palindromes.

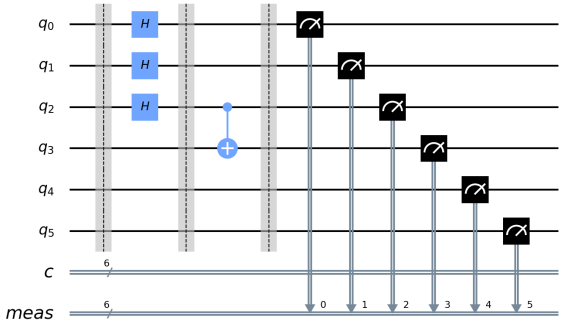
4 OTHER CONTEXT FREE GRAMMARS

Although this algorithm has been shown effective for the special case of the palindromes, there are many other CFGs with different structures, that may not map so easily onto quantum circuits. Does the algorithm still work in any of these other cases?

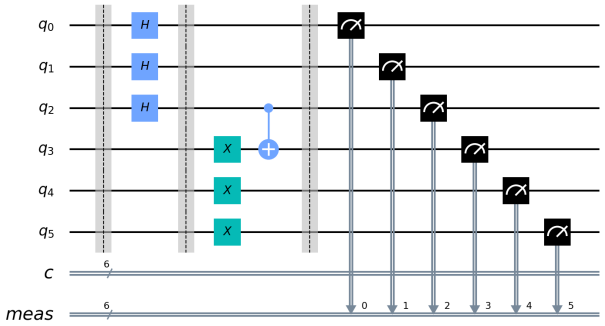
To answer that question, we implemented quantum circuits for the following five CFLs on strings of length $n = 3$.

$$L_1 = \{(0^n + 1^m)^* n > m\} : \text{more zeros than ones (in any order)}$$
$$L_2 = \{(0^n + 1^m)^* m > n\} : \text{more ones than zeros (in any order)}$$
$$L_3 = \{0^n 1^m n > m\} : \text{more zeros than ones, all zeros before ones}$$
$$L_4 = \{1^n 0^m n > m\} : \text{more ones than zeros, all ones before zeros}$$
$$L_5 = L_3 \cup L_4 : \text{either strings in } L_3 \text{ or } L_4$$

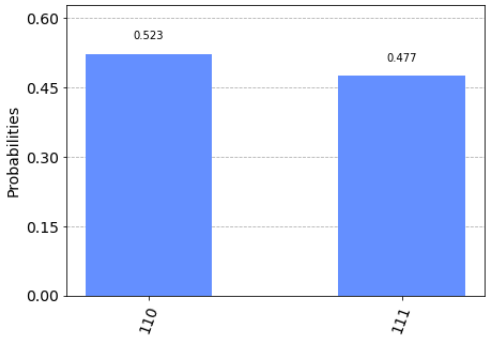
We start by describing the circuits for L_3 and L_4. These two circuits have an even more simple structure than the palindromes. To implement the $n = 3$ case, we only need one CNOT gate for L_3, and one CNOT as well as 3 X gates for L_3.

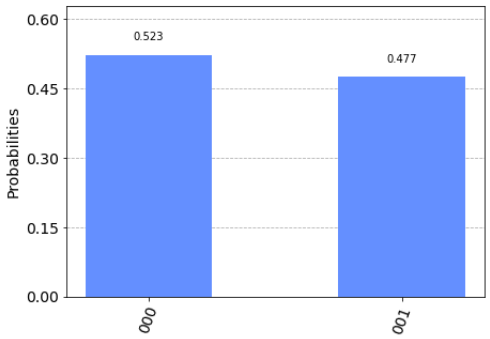


To convert from L_3 to L_4, we simply need to add X gates to reverse the bits in the output.



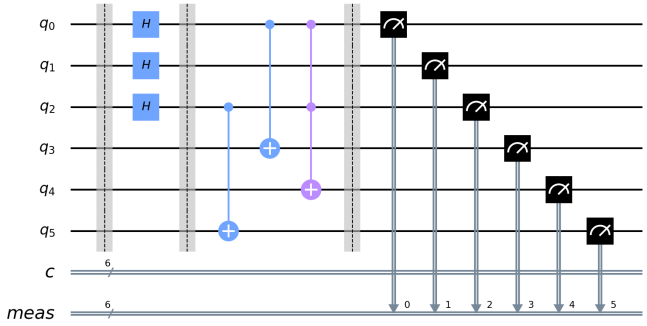
Like before, when we measure these circuits we produce all the strings in the CFL of that length with equal probability. The following graphs show the results of running these circuits for 1024 shots on the IBM quantum simulator.



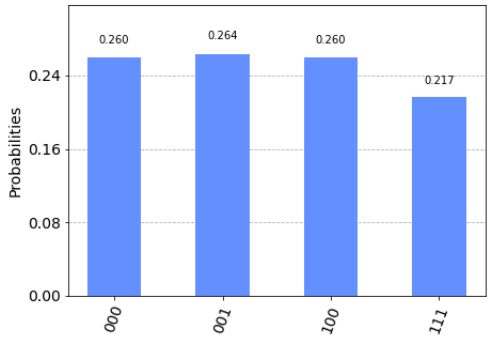


We highlight languages L_3 and L_4 because the existence of the simple rule, which allows us to convert one CFG recognizing circuit to another, suggests that there may be a very natural algorithm for generating f from an arbitrary CFG.

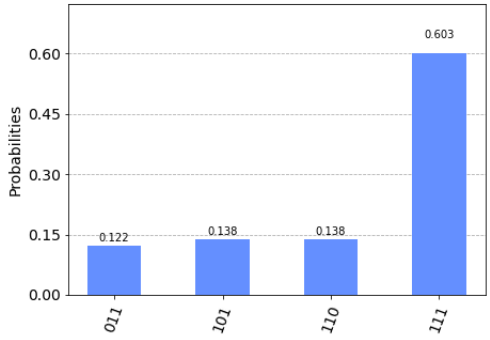
Furthermore, CFLs are closed under union. So we can construct another CFL, L_5 by taking the union of L_3 and L_4. Again, there is an interesting and clear relationship between the original circuits and the one that defines their union, pictured below:

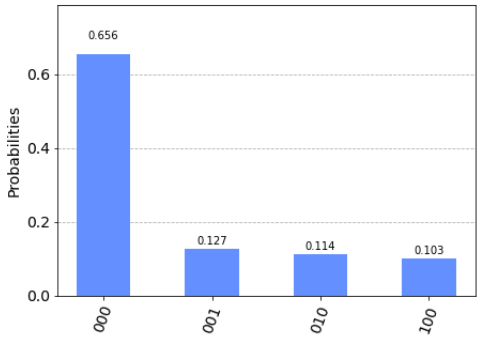


Running 1024 shots of this circuit on the IBM quantum simulator yields each of the four strings recognized by the grammar at length 3 with equal probability.



For languages L_1 and L_2, which measure whether strings have an more zeros than ones, or more ones than zeros, we construct similar circuits, which also have a general form structure that scales linearly with the length of the string. The results of these circuits are shown below:





5 GENERAL CASE COMPLEXITY

In the previous sections we showed that, an efficient general function f which allows us to recognize a grammar does exist for some CFGs, namely, the palindromes, and two types of zero/one imbalances in strings, providing a polynomial run-time improvement over the best classical approach. We now show that such an f must exist for all the simple cases (i.e., non-empty, binary CFGs).

The proof relies on the following theorem proved by Bennett et. al:

THEOREM 5.1. *The CCNOT gate is universal, assuming ancilla inputs (all set to|1) and garbage outputs are allowed; any standard AND/OR/NOT circuit for a function* $f : \{0,1\}^n \rightarrow \{0,1\}^m$ *may be efficiently transformed into a reversible one*

Since any non-empty CFG at a given length n can be expressed as binary functions of n bits, for the limited case, the result of the theorem applies and we know that f both exists and has an efficient mapping.

Going directly from the CFG to the binary function, however, presents yet another challenge, and the question of whether or not this is efficiently computable is less clear.

Supposing, however, that a mapping does always exists from the CFG G to the function f, then this algorithm's final run-time would be determined based on the efficiency of that mapping. If the time complexity of the mapping is $O(nlog(n))$, then this is the final run-time, and the speed-up is quite substantial. If the mapping falls somewhere between $O(nlog(n))$ and $O(n^3)$, then the quantum form still provides some improvement. If, however, the conversion from G to f performs on the order of $\Omega(n^3)$, than the original CYK algorithm performs just as well, so the new algorithm would be of little practical value.

6 CONCLUSION

In this work, we studied whether or not the problem of grammar membership could be more efficiently computed quantumly. We first presented a novel quantum approach to solving this problem. Then, as proof of concept, we implemented it for some simple CFGs. Our results show that, in some limited cases, the quantum algorithm provides a polynomial speed up in computation time.

The most pressing and looming question, however, is whether or not a general-purpose function exists which could convert any CFG into quantum circuit. While we know that for the limited case, i.e., with binary non-empty CFGs, there does always exist an f which produces the correct output, we do not know if this function exists in all cases, or it if it is always efficient. We hope to answer this question in future works.

7 REFERENCES

[Ben73] Charles Bennett. Logical reversibility of computation.IBM Journal of Researchand Development, 17(6):525–532, 1973