QCircuitNet: A Large-Scale Hierarchical Dataset for Quantum Algorithm Design

Anonymous Author(s)
Affiliation
Address
email

Abstract

Quantum computing is an emerging field recognized for the significant speedup it offers over classical computing through quantum algorithms. However, designing and implementing quantum algorithms pose challenges due to the complex nature of quantum mechanics and the necessity for precise control over quantum states. To address these challenges, we leverage AI to simplify and enhance the process. Despite the significant advancements in AI, there has been a lack of datasets specifically tailored for this purpose. In this work, we introduce QCircuitNet, a benchmark and test dataset designed to evaluate AI's capability in designing and implementing quantum algorithms in the form of quantum circuit codes. Unlike traditional AI code writing, this task is fundamentally different and significantly more complicated due to the highly flexible design space and the extreme demands for intricate manipulation of qubits. Our key contributions include: 1. The first comprehensive, structured universal quantum algorithm dataset. 2. A framework which formulates the task of quantum algorithm design for Large Language Models (LLMs), providing guidelines for expansion and potential evolution into a training dataset. 3. Automatic validation and verification functions, allowing for scalable and efficient evaluation methodologies. 4. A fair and stable benchmark that avoids data contamination, a particularly critical issue in quantum computing datasets. Our work aims to bridge the gap in available resources for AI-driven quantum algorithm design, offering a robust and scalable method for evaluating and improving AI models in this field. As we expand the dataset to include more algorithms and explore novel fine-tuning methods, we hope it will significantly contribute to both quantum algorithm design and implementation.

1 Introduction

Quantum computing is an emerging field in recent decades, which can attribute to the fact that algorithms on quantum computers may solve problems significantly faster than their classical counterparts. From the perspective of theoretical computer science, the design quantum algorithms have been investigated in various research directions - see the survey [Dalzell et al., 2023] and the quantum algorithm zoo [Zoo, 2024]. However, the design of quantum algorithms on quantum computers has been completed manually by researchers. This process is notably challenging due to highly flexible design space and extreme demands for a comprehensive understanding of mathematical tools and quantum properties.

Submitted to the 38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks. Do not distribute.

For these reasons, quantum computing is often considered to have high professional barriers. As the discipline evolves, we aim to explore more possibilities for algorithm design and implementation in the quantum setting. This is aligned with recent advances among "AI for Science", including AlphaFold [Jumper et al., 2021], AlphaGeometry [Trinh et al., 2024], etc. Recently, large language models (LLMs) has also become crucial among AI for science approaches [Yang et al., 2024, Zhang et al., 2024, Yu et al., 2024]. Therefore, we attempt to gear LLMs for quantum algorithm design. As far we know, there has not been any dataset for AI in quantum algorithm design. Existing work combining quantum computing and AI are mostly targeting at exploiting quantum computing for AI; there are some papers that apply AI for quantum computing, but they consider niche problems [Nakayama et al., 2023, Schatzki et al., 2021] or limited functions [Tang et al., 2023, Fürrutter et al., 2024], not quantum algorithm datasets of general interest. See more discussions in Section 2.

Key contributions. In this work, we propose QCircuitNet, the first comprehensive, structured dataset for quantum algorithm design. Technically, QCircuitNet has the following key contributions:

- It formulates the task of quantum algorithm design for Large Language Models (LLMs), providing guidelines for expansion that may evolve to be a training dataset.
- It has automatic validation and verification functions, allowing for scalable and efficient evaluation.
- It provides a fair and stable benchmark that avoids data contamination, a particularly critical issue in quantum computing datasets.

2 Related Work

To the best of our knowledge, QCircuitNet is the first dataset tailored specifically for quantum algorithm design. Previous efforts combining quantum computing with artificial intelligence primarily fall under the category of Quantum Machine Learning (QML), which aims at leveraging the unique properties of quantum systems to enhance machine learning algorithms and achieve potential improvements over their classical counterparts [Schuld et al., 2015, Biamonte et al., 2017, Ciliberto et al., 2018]. Corresponding datasets often focus on encoding classical data into quantum states, which we may call "Quantum for AI". For instance, MNISQ [Placidi et al., 2023] is a dataset of quantum circuits representing the original MNIST dataset [LeCun et al., 1998] generated by the AQCE algorithm [Shirakawa et al., 2021]. Considering the intrinsic nature of quantum properties, another category of datasets focuses on collecting quantum data to demonstrate quantum advantages since classical machine learning methods could fail to characterize particular patterns of quantum data. For example, Nakayama et al. [2023] created a VQE-generated quantum circuit dataset for classification of variational ansatzes and shows the quantum supremacy on this task. NTangled [Schatzki et al., 2021] further emphasized on the different types and amounts of entanglement and composed quantum states with various multipartite entanglement for classification. While these datasets successfully demonstrate the supremacy of quantum computing, they address rather niche problems which might not have practical applications.

There have also been efforts in the direction of "AI for Quantum", which explores the possibility of leveraging the huge potential of AI to facilitate the advancement of quantum computing. QDataSet [Perrier et al., 2022] collects data from simulations of one- and two-qubit systems and targets training classical machine learning algorithms for quantum control, quantum tomography, and noise mitigation. LLM4QPE [Tang et al., 2023] is a large language model style paradigm for predicting quantum system properties with pre-training and fine-tuning workflows. While the paradigm is interesting, the empirical experiments are limited to two downstream tasks: quantum phase classification and correlation prediction. Fürrutter et al. [2024] studied the application of diffusion models [Sohl-Dickstein et al., 2015, Rombach et al., 2022] to quantum circuit synthesis [Saeedi and Markov, 2013, J. et al., 2022]. Although their methodology is appealing, scalability issues must be addressed to achieve practical and meaningful unitary compilation.

The aforementioned works represent meaningful explorations at the intersection of artificial intelligence and quantum computing. However, none of these datasets or models considers the task which

interests the quantum computing community (from the theoretical side) the most: quantum algorithm design. Our work aims to take the first step in bridging this gap. It is worth noting that several quantum algorithm benchmarks already exist, such as QASMBench [Li et al., 2023] and VeriQBench [Chen et al., 2022]. However, these benchmarks are designed to evaluate the performance of NISQ (Noisy Intermediate-Scale Quantum) [Preskill, 2018] machines, rather than for training and evaluating AI models. For instance, QASMBench includes a diverse variety of quantum circuits from different domains based on the OpenQASM assembly representation [Cross et al., 2022], covering quantum circuits with qubit sizes ranging from 2 to 127. However, each algorithm is represented by only 2-3 QASM files at most. While this is sufficient for benchmarking the fidelity of quantum hardware and the efficiency of QC compilers, it fails as a dataset for AI in that it does not capture the design patterns of each algorithm and ignores the construction of different oracles, which are crucial to quantum computing. Similar limitations apply to VeriQBench.

3 Preliminaries for Quantum Computing

In this section, we will introduce necessary backgrounds for quantum computing related to this paper. Additional preliminaries can also be found in Appendix B. A more detailed introduction to quantum computing can be found in the standard textbook by Nielsen and Chuang [2000].

Quantum states. In classical computing, the basic unit is a bit. In quantum computing, the basic unit is a *qubit*. Mathematically, n ($n \in \mathbb{N}$) qubits forms an N-dimensional Hilbert space for $N = 2^n$. An n-qubit *quantum state* $|\phi\rangle$ can be written as

$$|\phi\rangle = \sum_{i=0}^{N-1} \alpha_i |i\rangle, \text{ where } \sum_{i=0}^{N-1} |\alpha_i|^2 = 1. \tag{1}$$

Here $|\cdot\rangle$ represents a column vector, also known as a ket state. The tensor product of two quantum states $|\phi_1\rangle = \sum_{i=0}^{N-1} \alpha_i |i\rangle$ and $|\phi_2\rangle = \sum_{j=0}^{M-1} \beta_j |j\rangle$ with $M = 2^m$, $m \in \mathbb{N}$ is defined as

$$|\phi_1\rangle \otimes |\phi_2\rangle = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \alpha_i \beta_j |i, j\rangle, \tag{2}$$

where $|i, j\rangle$ is an $(n+m)$-qubit state with first n qubits being the state $|i\rangle$ and the last m qubits being the state $|j\rangle$. When there is no ambiguity, $|\phi_1\rangle \otimes |\phi_2\rangle$ can be abbreviated as $|\phi_1\rangle|\phi_2\rangle$.

Quantum oracles. To study a Boolean function $f\colon \{0,1\}^n \to \{0,1\}^m$, we need to gain its access. Classically, a standard setting is to being able to *query* the function, in the sense that if we input an $x \in \{0,1\}^n$, we will get the output $f(x) \in \{0,1\}^m$. In quantum computing, the counterpart is a quantum query, which is instantiated by a *quantum oracle*. Specifically, the function f is encoded as an oracle U_f such that for any $x \in \{0,1\}^n$, $z \in \{0,1\}^m$,

$$U_f|x\rangle|z\rangle = |x\rangle|z \oplus f(x)\rangle, \tag{3}$$

where $\oplus$ is the plus modulo 2. Note that a quantum query to the oracle is stronger than a classical query in the sense that the quantum query can be applied to a state in *superposition*: For an input state $\sum_i c_i|x_i\rangle|z_i\rangle$ with $\sum_i |c_i|^2 = 1$, the output state is $\sum_i c_i|x_i\rangle|z_i \oplus f(x_i)\rangle$; measuring this state gives x_i and $z_i \oplus f(x_i)$ with probability $|c_i|^2$. A classical query for x can be regarded as the special setting with $c_1 = 1$, $x_1 = x$, $z_1 = 0^m$, and $c_i = 0$ for all other i.

Quantum gates. Similar to classical computing that can stem from logic synthesis with AND, OR, and NOT, quantum computing is also composed of basic quantum gates. For instance, the Hadamard H is the matrix $\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, satisfying $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. In general, an n-qubit quantum gate is a unitary matrix $\mathbb{C}^{2^n \times 2^n}$.

4 QCircuitNet Dataset

4.1 Task Suite

For the general purpose of quantum algorithm design, we consider two categories of tasks: oracle construction and algorithm design. These two tasks are crucial for devising and implementing a complete quantum algorithm, with oracle construction serving as the premise for algorithm design.

4.1.1 Task I: Oracle Construction

The construction of such an oracle U_f using quantum gates is deeply rooted in the research topic of reversible quantum logic synthesis, which remains a challenge for complex Boolean functions. In this dataset, we mainly focus on the construction of textbook-level oracles: Bernstein-Vazirani Problem [Bernstein and Vazirani, 1993], Deutsch-Jozsa Problem [Deutsch and Jozsa, 1992], Simon's Problem [Simon, 1997], and Grover's algorithm for unstructured search [Grover, 1996] (including constructions of both the oracle and the diffusion operator). We also consider more advanced oracle construction tasks which we refer to as "Problem Encoding". For example, one can apply Grover's oracle to solving constraint problems such as SAT and triangle finding [Ambainis, 2004]. The intrinsic nature of formulating problem encoding tasks for LLMs slightly differs from quantum logic synthesis, and we refer the readers to Appendix A for more detailed discussion.

4.1.2 Task II: Quantum Algorithm Design

A general description of a quantum algorithm in natural language could be verbose and vague. Considering that quantum circuits stand at the core of designing and implementing a quantum algorithm, and that they resemble a special type of "language", we decide to use quantum circuits as the main medium for LLMs to generate for algorithm design. There are certain crucial points to consider when designing this framework to formulate the task precisely:

- From the perspective of quantum algorithm design, the oracle is usually provided as a blackbox gate since the goal of many algorithms is to determine the property of the function $f(x)$ encoded by the oracle U_f. If the model has access to the gate implementation of the oracle, it can directly deduce the property from the circuit, failing the purpose of designing a quantum algorithm to decode the information. However, for all experiment platforms, a quantum circuit needs to be explicitly constructed to compile and run successfully, which means the oracle should be provided with exact gate implementation. Most tutorials and benchmarks (especially those based on OpenQASM) simply merge the circuit implementation of the oracle and the algorithm as a whole for demonstration purposes. In our task of gearing LLMs for quantum algorithm design, how to separate the algorithm circuits from oracle implementation to avoid information leakage is a critical point to consider.
- A quantum algorithm constitutes not only the quantum circuit, but also the interpretation of execution (typically measurement) results of the quantum circuit. For example, in Simon's algorithm, the measurement results y_i are not direct answer s to the problem, but rather satisfies the property of $s \cdot y_i = 0$. Linear equations need to be solved to obtain the final answer. In this case, for a complete algorithm design, the model should also specify the way to process the execution results to derive the answer to the original problem.
- Quantum circuits for the same algorithm vary with different qubit number n. Although this is trivial for theoretical design, it needs to be considered when implementing concrete quantum circuits.

Beyond quantum algorithm design, we also consider quantum teleportation and quantum key distribution, since these protocols are widely used in quantum information. We cover their details in Appendix B.

4.2 Dataset Structure

The overall structure of QCircuitNet is illustrated as follows (more details are given in Appendix A):

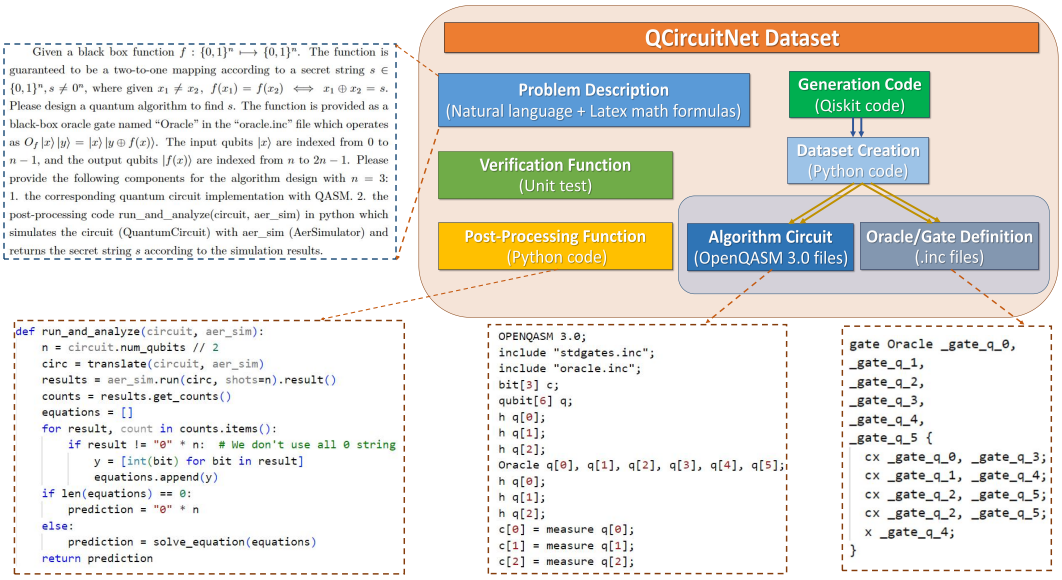

Figure 1: Structure of QCircuitNet. The components of QCircuitNet are presented in the frame on the top-right. As a showcase, this figure presents the components for Simon's problem [Simon, 1997], including its problem description in natural language, post-processing function in python code, circuit in a .qasm file, and oracle definition in a .inc file.

Design Principles. As discussed in Section 4.1, a critical consideration in formulating the framework is the dilemma between providing the oracle as a black box for quantum algorithm design and the need for its explicit construction to execute the circuit and interpret the results, making the algorithm design complete. Additionally, model training and reference present challenges, particularly for LLMs in generating complex and precise composite gates and evaluating the results efficiently. To address these obstacles, we highlight the following construction principles, which are specially designed to adapt to these two tasks:

- For algorithm design tasks, as discussed in Section 4.1.2, we provide the oracle as a black-box gate named "Oracle" with the explicit definition in a separate "oracle.inc" library, which is supported by the OpenQASM 3.0 grammar. In this way, we make sure that the model can use the oracle without accessing its underlying function, which solves the problem of isolating oracle definition from the algorithm circuit.
- For oracle construction tasks, we ask the model to directly output the quantum circuit in QASM format. For algorithm design task, we require both a quantum circuit and a post-processing function to derive the final answer from circuit execution results. Moreover, we ask the model to explicitly set the shots needed to run the circuit itself in order to characterize the query complexity, which is critical in the theoretical analysis of algorithms.
- For available quantum gates, we provide the definition of some important composite gates not included in the standard QASM gate library in a "customgates.inc". Hierarchical definition for multi-controlled X gate contains 45060 lines for qubit number $n = 14$ in OpenQASM format, which is impossible for AI models to accurately generate at the time. Providing these as a .inc file guarantees the correctness of OpenQASM's grammar while avoiding the generation of complicated gates, which is a distraction from the original design task.
- To verify models' output automatically without human evaluation, we compose verification functions to validate the syntax of QASM / Qiskit and the functionality of the implemented circuits / codes. Since comprehensive Logic Equivalence Checking (LEC) might be inefficient for the throughput of LLM inference, we perform the verification by directly checking the correctness of output with extensive test cases.

Based on theses principles, we proposed the framework of QCircuitNet. Below is a more detailed explanation for the 7 components of the dataset:

1. **Problem Description:** carefully hand-crafted prompts stating the oracle to be constructed or the target problem to be solved in natural language and latex math formulas. If the problem involves the usage of a quantum oracle or composite gates beyond the standard gate library, the interfaces of the oracle / gate will also be included (input qubits, output qubits, function mechanism).
2. **Generation Code:** one general Qiskit [Javadi-Abhari et al., 2024] code to create quantum circuits for oracles or algorithms of different settings, such as distinct secret strings or various qubit numbers. We choose Qiskit as the main experiment platform because it is a general quantum programming software widely used for the complete workflow from creating quantum circuits to transpiling, simulation, and execution on real hardware.
3. **Algorithm Circuit:** a .qasm file storing the quantum circuit for each specific setting. We choose OpenQASM 3.0 [Cross et al., 2022] as the format to store the quantum circuits, because Qiskit, as a python library, can only create quantum circuits at runtime instead of explicitly saving the circuits at gate level.[1]
4. **Post-Processing Function:** this is for Algorithm Design task only, see Section 4.1.2. The function takes a complete quantum circuit as input, uses the Qiskit AerSimulator to execute the circuit, and returns the final answer to the original problem according to the simulation results. For state preparation problems such as creating a GHZ state of n qubits, this function returns the qubit indices of the generated state.
5. **Oracle / Gate Definition:** a .inc file to provide definitions of composite gates or oracles. For oracle construction tasks, this only includes the definition of composite gates required to build the oracle. For algorithm design tasks, we also provide the gate definition of the oracle in this file, which successfully delivers the oracle in a black-box way.
6. **Verification Function:** a function to evaluate whether the implemented oracle / algorithm successfully achieves the desired purpose with grammar validation and test cases verification. The function returns -1 if there exist grammar errors, and returns a score between $[0, 1]$ indicating the success rate on test cases.[2]
7. **Dataset Creation Script:** the script to create the dataset from scratch in the format suitable for fine-tuning / evaluating LLMs. It contains the following functions: 1. generate primitive QASM circuits. 2. extract gate definitions and add include instructions to create algorithm circuit, the direct output of model. 3. validate and verify the correctness of the data points in the dataset. 4. concatenate algorithm circuit with problem description as a json file for the benchmark pipeline.

This structure of QCircuitNet provides a general framework to formulate quantum algorithm design for large language models, with an easy extension to more advanced quantum algorithms.

5 Experiments

5.1 Methodology for Benchmarking

We benchmark the quantum algorithm design capabilities of leading closed-source and open-source large language models using QCircuitNet. The workflow of our benchmark is illustrated in Figure 2. The total computation cost is approximately equivalent to two days on an A100 GPU.

[1] Although currently the Qiskit APIs for importing and dumping OpenQASM 3.0 files are still in experimental stage, we choose to adopt version 3.0 over 2.0 in that it supports parameterized circuits, which allows for extending the framework to variational quantum algorithms [Cerezo et al., 2021] by saving parameterized varational ansatzes.

[2] The verification function explicitly integrates the oracle / gate definition library with output algorithm circuit since Qiskit importer for OpenQASM 3.0 does not support non-standard gate libraries currently.

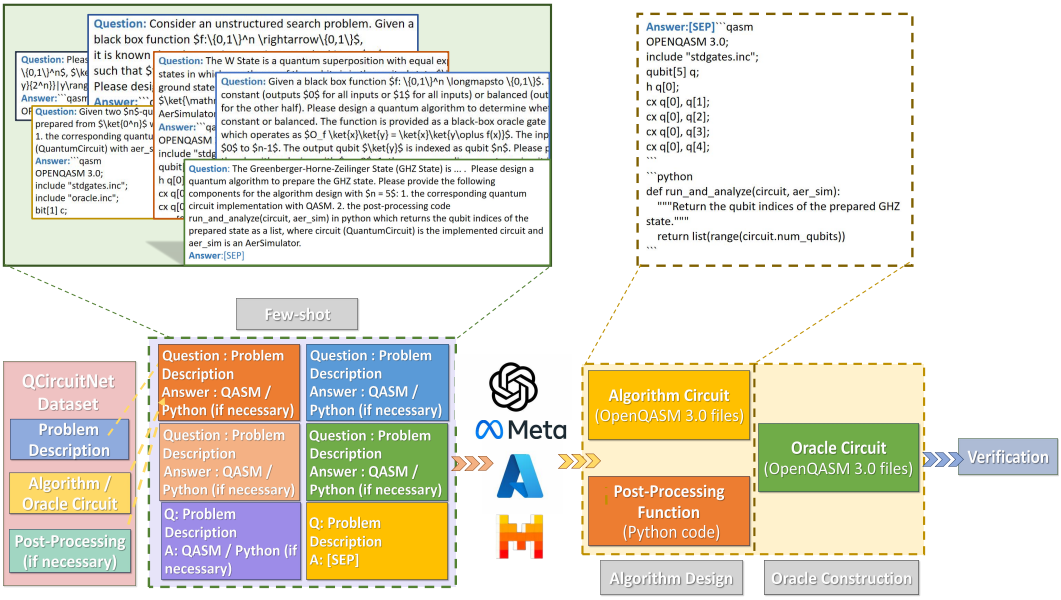

Figure 2: Flowchart of benchmarking QCircuitNet.

Models. Recently, the GPT series models have become the benchmark for generative models due to their exceptional performance. Specifically, we include two models from OpenAI, GPT-3.5-turbo [Brown et al., 2020] and GPT-4 [OpenAI et al., 2024], in our benchmark. Additionally, the LLAMA series models [Touvron et al., 2023a,b] are widely recognized as leading open-source models, and we have selected LLAMA-3-8B for our study. For a comprehensive evaluation, we also benchmark Phi-3-medium-128k [Abdin et al., 2024] and Mistral-7B-v0.3 [Jiang et al., 2023].

Prompts. We employ a few-shot learning framework, a prompting technique that has shown considerable success in generative AI [Xie et al., 2021]. In this approach, we utilize either 1 or 5 examples, followed by a problem description. To ensure we do not train and test on the same quantum algorithm, we implement k-fold validation. This method involves using one problem as the test set while the remaining problems serve as the training set, rotating through each problem one at a time.

Evaluation Metrics. We use three evaluation metrics:

1. BLEU Score: this metric measures how closely the generated code matches the reference code, with a higher BLEU score indicating greater similarity.
2. Byte Perplexity: this metric evaluates the model's ability to predict the next byte in a sequence. Lower byte perplexity indicates better performance by reflecting the model's predictive accuracy.
3. Verification function: this function checks the syntax validation and the result correctness of the code produced by the language model, and returns a score depending on the performance. See Section 4.2 for more detailed discussion.

5.2 Results

The results for BLEU and verification function score are shown in Figure 3, Table 1, and Table 2. We include the results of Byte Perplexity and more experiments in Appendix C.

As illustrated in the table, verification scores for the output of the model reveal that almost none can produce a correct algorithm, because a single mistake could make the whole algorithm fail. However, we can still partially assess the models' ability to solve quantum problems by measuring the BLEU

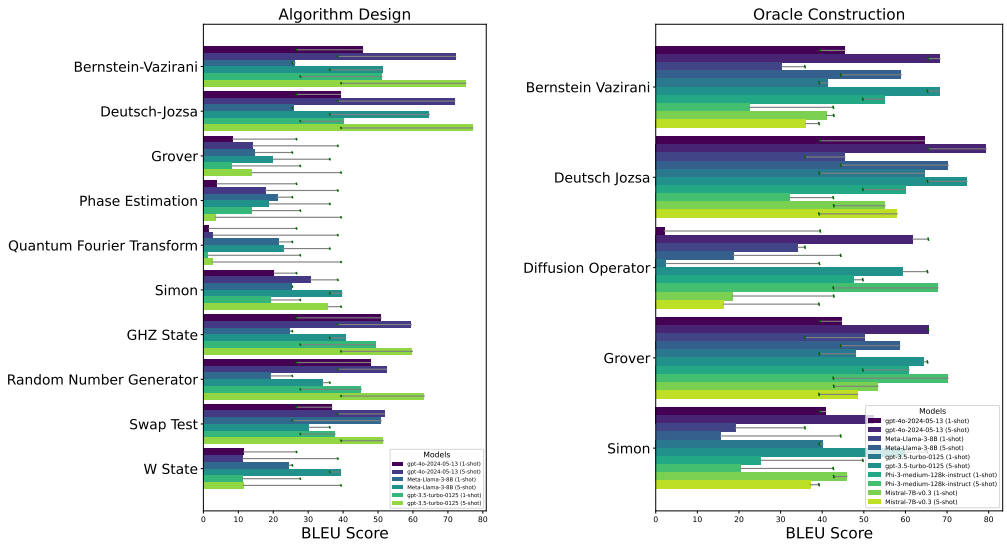

Figure 3: Benchmarking algorithm design and oracle construction in BLEU scores.

score. The figure indicates that GPT-4o significantly outperforms all other models. Additionally, nearly all models demonstrate the ability to learn quantum knowledge from context, as the five-shot prompt performs much better than the one-shot alternative.

The figure also reveals the different difficulty levels for each algorithm. For simple quantum algorithms such as the Bernstein-Vazirani algorithm where directly applying more H gates to the qubits solves the problem, language models tend to perform well. However, for complicated algorithms such as the W state where the parameters vary with qubit number, the models tend to perform poorly.

Table 1: Benchmarking algorithm design in verification function scores.

Model	Shot	Bernstein-Vazirani	Deutsch-Jozsa	Grover	Phase Estimation	Quantum Fourier Transform	Simon	GHZ State	Random Number Generator	Swap Test	W State
gpt-4o-2024-05-13	1	-0.8462	-0.5538	-0.7089	-1.0000	-1.0000	-0.6692	-0.8462	-1.0000	-1.0000	-1.0000
gpt-4o-2024-05-13	5	-0.3054	0.0135	-0.2071	-0.5357	-0.6154	-0.3692	-0.1538	-0.4967	-0.8700	-0.9231
Meta-Llama-3-8B	1	-0.2308	-0.7692	-0.7143	-0.8571	-0.9231	-1.0000	-0.6154	-0.9285	-1.0000	-0.3846
Meta-Llama-3-8B	5	0.0769	-0.2308	-0.5393	-1.0000	-0.7692	-0.8462	-0.3846	-0.7276	-1.0000	-0.1538
gpt-3.5-turbo-0125	1	-0.8462	-0.7154	-0.5679	-1.0000	-1.0000	-0.6231	-0.8462	-1.0000	-1.0000	-1.0000
gpt-3.5-turbo-0125	5	-0.6154	-0.0571	-0.0500	-1.0000	-0.6538	-0.1646	-0.2308	-0.4513	-0.8778	-0.8462

Table 2: Benchmarking oracle construction in verification function scores.

Model	Shot	Bernstein-Vazirani	Deutsch-Jozsa	Diffusion-Operator	Grover	Simon
gpt-4o-2024-05-13	1	-0.3200	-0.0100	-0.8462	-0.9885	-0.4674
gpt-4o-2024-05-13	5	-0.1100	0.0800	-0.3077	-0.9540	-0.0870
Meta-Llama-3-8B	1	-0.7300	-0.5000	-0.3846	-1.0000	-0.6848
Meta-Llama-3-8B	5	-0.0500	0.1700	-0.8462	-1.0000	-0.6413
gpt-3.5-turbo-0125	1	-0.3500	-0.0400	-0.8462	-1.0000	-0.3696
gpt-3.5-turbo-0125	5	-0.1100	0.0200	-0.3077	-0.9770	-0.1087
Phi-3-medium-128k-instruct	1	-0.6800	-0.6100	-0.9231	-1.0000	-0.7500
Phi-3-medium-128k-instruct	5	-0.5400	-0.4300	-1.0000	-1.0000	-0.8370
Mistral-7B-v0.3	1	-0.4000	-0.4300	-0.9231	-0.9540	-0.6087
Mistral-7B-v0.3	5	-0.3700	-0.1300	-1.0000	-0.9195	-0.2391

5.3 Observations and Analysis

The Challenge of LLM for Quantum Algorithm Design. As shown by the experiment results, the integration of LLMs into quantum algorithm design presents several challenges:

1. Lack of data: Unlike classical computing and code generation, where vast datasets and extensive examples exist, the field of quantum computing is still nascent, with limited accessible data. This scarcity hampers the ability of LLMs to learn and generalize effectively.
2. Distinct nature of each algorithm: Quantum algorithms can be seen as unitary maps but in exponential size linear spaces. This distinct nature makes it intractable for LLMs to generalize knowledge from one algorithm to another, posing challenges to transfer learning.
3. Reasoning of underlying mechanism: Quantum algorithms involve deep comprehension of unitary transformations and the evolution of quantum states. Such reasoning goes beyond simple pattern recognition and is difficult for LLMs to grasp and apply accurately.
4. Quantum programming language syntax: The syntax of quantum programming languages, such as Qiskit and OpenQASM, introduces an additional layer of complexity. As shown by the verification scores, the models can barely output circuit / codes with correct syntax, demonstrating that this is a non-trivial task, which challenges the current capabilities of LLMs.

Usage of QCircuitNet Dataset. Our dataset helps provide guidance to address these challenges:

1. Formulate the task: We propose framing algorithm design tasks in circuit or code form rather than natural language descriptions, which can be vague, or mathematical formulas, which are difficult to verify. This provides a concrete framework for LLMs to operate within.
2. Clarify descriptions with concrete examples: The dataset includes detailed descriptions of representative problems in universal quantum algorithms, accompanied by concrete cases, which helps bridge the gap between abstract algorithms and practical implementations.
3. Benchmark for fair evaluation: To improve the capability of LLMs in quantum algorithm design, we need a fair and robust evaluation method first. Our dataset includes metrics and benchmarks for such purpose, providing a foundation for developing and testing novel improvement methods.

Implications for AI Learning. We observe a performance separation between writing general qiskit codes and explicit gate-level circuits in QASM. Since Qiskit provides detailed tutorial with general codes for several algorithms, this may imply a *data contamination* phenomenon where LLMs rely on memorization and retrieval rather than genuine algorithm design. Similarly, current benchmarks for AI code generation and syntax learning may also suffer from this unseen bias. Our dataset, based on QASM files created from scratch, may help circumvent this issue and serve as a stable and fair evaluation method for benchmarking AI syntax learning.

6 Conclusions and Future Work

In this paper, we propose QCircuitNet, the first comprehensive, structured universal quantum algorithm dataset and quantum circuit generation benchmark for AI models. It contains automatic validation and verification functions, allowing for scalable and efficient evaluation methodologies. Benchmarking of QCircuitNet on up-to-date LLMs are systematically conducted.

Our work leaves several open questions for future investigation:

- QCircuitNet is a benchmarking dataset for LLMs. It is of general interest to extend benchmarking to training, which will help LLMs better maneuver quantum algorithm design. This may need implementations of more advanced algorithms to make it a more meaningful training dataset.
- Since quantum algorithms have fundamental difference from classical algorithms, novel fine-tuning methods to attempt quantum algorithm design and quantum circuit implementation, or even development of new quantum algorithms by LLMs are solicited.
- Currently, variational quantum algorithms [Cerezo et al., 2021] can already be implemented on near-term NISQ machines [Preskill, 2018]. It would be also of general interest to extend QCircuitNet to contain the design and implementation of variational quantum algorithms.

References

Quantum algorithm zoo. https://quantumalgorithmzoo.org/, 2024. Accessed: 2024-05-30.

M. Abdin, S. Ade Jacobs, A. A. Awan, J. Aneja, A. Awadallah, H. Awadalla, N. Bach, A. Bahree, A. Bakhtiari, J. Bao, H. Behl, A. Benhaim, M. Bilenko, J. Bjorck, S. Bubeck, Q. Cai, M. Cai, C. César Teodoro Mendes, W. Chen, V. Chaudhary, D. Chen, D. Chen, Y.-C. Chen, Y.-L. Chen, P. Chopra, X. Dai, A. Del Giorno, G. de Rosa, M. Dixon, R. Eldan, V. Fragoso, D. Iter, M. Gao, M. Gao, J. Gao, A. Garg, A. Goswami, S. Gunasekar, E. Haider, J. Hao, R. J. Hewett, J. Huynh, M. Javaheripi, X. Jin, P. Kauffmann, N. Karampatziakis, D. Kim, M. Khademi, L. Kurilenko, J. R. Lee, Y. T. Lee, Y. Li, Y. Li, C. Liang, L. Liden, C. Liu, M. Liu, W. Liu, E. Lin, Z. Lin, C. Luo, P. Madan, M. Mazzola, A. Mitra, H. Modi, A. Nguyen, B. Norick, B. Patra, D. Perez-Becker, T. Portet, R. Pryzant, H. Qin, M. Radmilac, C. Rosset, S. Roy, O. Ruwase, O. Saarikivi, A. Saied, A. Salim, M. Santacroce, S. Shah, N. Shang, H. Sharma, S. Shukla, X. Song, M. Tanaka, A. Tupini, X. Wang, L. Wang, C. Wang, Y. Wang, R. Ward, G. Wang, P. Witte, H. Wu, M. Wyatt, B. Xiao, C. Xu, J. Xu, W. Xu, S. Yadav, F. Yang, J. Yang, Z. Yang, Y. Yang, D. Yu, L. Yuan, C. Zhang, C. Zhang, J. Zhang, L. Lyna Zhang, Y. Zhang, Y. Zhang, Y. Zhang, and X. Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024. arXiv:2404.14219

A. Ambainis. Quantum search algorithms. *ACM SIGACT News*, 35(2):22–35, 2004. arXiv:quant-ph/0504012

C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing, Bangalore*, pages 175–179, 1984.

C. H. Bennett and S. J. Wiesner. Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Phys. Rev. Lett.*, 69:2881–2884, Nov 1992. doi: 10.1103/PhysRevLett.69.2881. URL https://link.aps.org/doi/10.1103/PhysRevLett.69.2881.

C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters*, 70(13):1895, 1993.

E. Bernstein and U. Vazirani. Quantum complexity theory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, pages 11–20, 1993.

J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017. arXiv:1611.09347

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020. arXiv:2005.14165

M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, and P. J. Coles. Variational quantum algorithms. *Nature Reviews Physics*, 3(9):625–644, 2021. arXiv:2012.09265

K. Chen, W. Fang, J. Guan, X. Hong, M. Huang, J. Liu, Q. Wang, and M. Ying. VeriQBench: A benchmark for multiple types of quantum circuits, 2022. arXiv:2206.10880

C. Ciliberto, M. Herbster, A. D. Ialongo, M. Pontil, A. Rocchetto, S. Severini, and L. Wossnig. Quantum machine learning: a classical perspective. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 474(2209):20170551, 2018. arXiv:1707.08561

A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, and B. R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022. arXiv:2104.14722

A. M. Dalzell, S. McArdle, M. Berta, P. Bienias, C.-F. Chen, A. Gilyén, C. T. Hann, M. J. Kastoryano, E. T. Khabiboulline, A. Kubica, G. Salton, S. Wang, and F. G. Brandao. Quantum algorithms: A survey of applications and end-to-end complexities, 2023. arXiv:2310.03011

D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

F. Fürrutter, G. Muñoz-Gil, and H. J. Briegel. Quantum circuit synthesis with diffusion models. *Nature Machine Intelligence*, pages 1–10, 2024. arXiv:2311.02041

L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, pages 212–219. ACM, 1996. arXiv:quant-ph/9605043

A. J., A. A. Adedoyin, J. J. Ambrosiano, P. M. Anisimov, W. R. Casper, G. Chennupati, C. J. Coffrin, H. N. Djidjev, D. O. Gunter, S. Karra, N. W. Lemons, S. Lin, A. Malyzhenkov, D. D. L. Mascarenas, S. M. Mniszewski, B. T. Nadiga, D. O'Malley, D. A. Oyen, S. D. Pakin, L. Prasad, R. M. Roberts, P. R. Romero, N. Santhi, N. Sinitsyn, P. J. Swart, J. G. Wendelberger, B. Yoon, R. J. Zamora, W. Zhu, S. J. Eidenbenz, A. Bärtschi, P. J. Coles, M. D. Vuffray, and A. Y. Lokhov. Quantum algorithm implementations for beginners. *ACM Transactions on Quantum Computing*, 3(4), 7 2022. doi: 10.1145/3517340. arXiv:1804.03719

A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta. Quantum computing with Qiskit, 2024. arXiv:2405.08810

A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. Singh Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. Renard Lavaud, M.-A. Lachaux, P. Stock, T. Le Scao, T. Lavril, T. Wang, T. Lacroix, and W. El Sayed. Mistral 7B, 2023. arXiv:2310.06825

J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

A. Li, S. Stein, S. Krishnamoorthy, and J. Ang. QASMBench: A low-level quantum benchmark suite for NISQ evaluation and simulation. *ACM Transactions on Quantum Computing*, 4(2):1–26, 2023. arXiv:2005.13018

A. Nakayama, K. Mitarai, L. Placidi, T. Sugimoto, and K. Fujii. VQE-generated quantum circuit dataset for machine learning, 2023. arXiv:2302.09751

M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2000.

OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu,

H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph. GPT-4 technical report, 2024. URL `https://openai.com`. arXiv:`2303.08774`

E. Perrier, A. Youssry, and C. Ferrie. QDataSet, quantum datasets for machine learning. *Scientific Data*, 9(1):582, 2022. arXiv:`2108.06661`

L. Placidi, R. Hataya, T. Mori, K. Aoyama, H. Morisaki, K. Mitarai, and K. Fujii. MNISQ: A large-scale quantum circuit dataset for machine learning on/for quantum computers in the NISQ era, 2023. arXiv:`2306.16627`

J. Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018. arXiv:`1801.00862`

R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022. arXiv:`2112.10752`

M. Saeedi and I. L. Markov. Synthesis and optimization of reversible circuits—a survey. *ACM Computing Surveys (CSUR)*, 45(2):1–34, 2013. arXiv:`1110.2574`

L. Schatzki, A. Arrasmith, P. J. Coles, and M. Cerezo. Entangled datasets for quantum machine learning, 2021. arXiv:`2109.03400`

M. Schuld, I. Sinayskiy, and F. Petruccione. An introduction to quantum machine learning. *Contemporary Physics*, 56(2):172–185, 2015. arXiv:`1409.3097`

T. Shirakawa, H. Ueda, and S. Yunoki. Automatic quantum circuit encoding of a given arbitrary quantum state, 2021. arXiv:2112.14524

D. R. Simon. On the power of quantum computation. *SIAM Journal on Computing*, 26(5):1474–1483, 1997.

J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine learning*, pages 2256–2265. PMLR, 2015. arXiv:1503.03585

Y. Tang, H. Xiong, N. Yang, T. Xiao, and J. Yan. Q-TAPE: A task-agnostic pre-trained approach for quantum properties estimation. In *The Twelfth International Conference on Learning Representations*, 2023.

H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models, 2023a. arXiv:2302.13971

H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. Singh Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023b. arXiv:2307.09288

T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong. Solving Olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

S. M. Xie, A. Raghunathan, P. Liang, and T. Ma. An explanation of in-context learning as implicit bayesian inference. In *International Conference on Learning Representations*, 2021. arXiv:2111.02080

K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024. arXiv:2306.15626

B. Yu, F. N. Baker, Z. Chen, X. Ning, and H. Sun. LlaSMol: Advancing large language models for chemistry with a large-scale, comprehensive, high-quality instruction tuning dataset, 2024. arXiv:2402.09391

Z. Zhang, Y. Zhang, H. Yao, J. Luo, R. Zhao, B. Huang, J. Zhao, Y. Liao, K. Li, L. Zhao, et al. Xiwu: A basis flexible and learnable LLM for high energy physics, 2024. arXiv:2404.08001

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 6.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] Quantum computing is still a nascent technology at the moment. Therefore, our work does not have negative societal impacts from our perspective. In the future, we welieve that our dataset can be beneficial for quantum algorithm design and the field of quantum computing as a whole.

(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A] We do not have theoretical results.
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments (e.g. for benchmarks)...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See supplemental material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [N/A] The experiments do not contain model training.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] Neither random initialization nor stochastic gradient descent is in our experiments. There is no need for repeated experiments.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Section 5.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We cited Qiskit [Javadi-Abhari et al., 2024], OpenQASM [Cross et al., 2022], and QASMBench [Li et al., 2023] in our paper.
   (b) Did you mention the license of the assets? [Yes] The links of the aforementioned assets are given in reference.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] Our dataset is proposed by ourselves.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] Our dataset contains purely quantum circuits and does not contain personally identifiable information or offensive content.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Details of QCircuitNet

The QCircuitNet Dataset, along with its Croissant metadata, is available on Anonymous GitHub at the following link: https://anonymous.4open.science/r/QCircuitNet-DE28/

QCircuitNet has the following directory structure:

```
QCircuitNet
├── Oracle Construction ..... All data for the oracle construction task
│   ├── Quantum Logic Synthesis .....Textbook-level oracles used in the experiments
│   └── Problem Encoding .....Advanced oracles encoding application scenarios
└── Algorithm Design .....All data for the quantum algorithm design task
    ├── Quantum Computing .....Textbook-level universal quantum computing algorithms
    └── Quantum Information .....Textbook-level quantum information tasks and protocols
```

In each subdirectory, there is a folder for each specific algorithm. For instance, the folder structure for Simon's algorithm is as follows:

```
Algorithm Design
└── Quantum Computing
    └── simon ..... All data for the Simon's Problem
        ├── simon-dataset.py ..... Dataset creation script
        ├── simon-generation.py ..... Qiskit generation code
        ├── simon-post-processing.py .....Post-processing function
        ├── simon-utils.py .....Utility functions for verification
        ├── simon-verification.py .....Verification function
        ├── simon-description.txt ..... Problem description
        ├── simon-verification.txt ..... Verification results of the data points
        ├── full circuit ..... Raw data of quantum circuits
        │   ├── simon-n2
        │   │   └── simon-n2-s11-k11.qasm ..... Full circuit for a concrete setting
        │   ├── simon-n3
        │   │   ├── simon-n3-s011-k001.qasm
        │   │   ├── simon-n3-s011-k101.qasm
        │   │   ├── simon-n3-s100-k001.qasm
        │   │   └── simon-n3-s100-k101.qasm
        │   └── ...
        ├── test oracle ..... Extracted oracle definitions
        │   ├── n2
        │   │   └── trial1
        │   │       ├── oracle.inc ..... Oracle definition as a .inc file
        │   │       └── oracle-info.txt ..... Oracle information (such as key strings)
        │   ├── n3
        │   │   ├── trial1
        │   │   │   ├── oracle.inc
        │   │   │   └── oracle-info.txt
        │   │   ├── trial2
        │   │   │   ├── oracle.inc
        │   │   │   └── oracle-info.txt
        │   │   ├── trial3
        │   │   │   ├── oracle.inc
        │   │   │   └── oracle-info.txt
        │   │   └── trial4
        │   │       ├── oracle.inc
        │   │       └── oracle-info.txt
        │   └── ...
        ├── simon-n2.qasm ..... Algorithm circuit for model output
        ├── simon-n3.qasm
        ├── simon-n4.qasm
        ├── simon-n5.qasm
        └── ...
```

We expect to extend QCircuitNet under this general structure.

A.1 Format

In this subsection, we provide concrete examples to illustrate the different components of QCircuitNet. We use the case of Simon's Problem throughout the demonstration to achieve better consistency. For further details, please check the code repository.

1. **Problem Description:** this is the carefully hand-crafted description of the task in natural language and latex math formulas. The description is provided as one template for each algorithm, and the concrete settings (such as the qubit number) are replaced when creating the data points in json. The file is named as "{algorithm_name}_description.txt".

Problem Description Template for Simon's Problem

Given a black box function $f : \{0,1\}^n \longmapsto \{0,1\}^n$. The function is guaranteed to be a two-to-one mapping according to a secret string $s \in \{0,1\}^n, s \neq 0^n$, where given $x_1 \neq x_2$, $f(x_1) = f(x_2) \iff x_1 \oplus x_2 = s$. Please design a quantum algorithm to find s. The function is provided as a black-box oracle gate named "Oracle" in the "oracle.inc" file which operates as $O_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$. The input qubits $|x\rangle$ are indexed from 0 to $n-1$, and the output qubits $|f(x)\rangle$ are indexed from n to $2n-1$. Please provide the following components for the algorithm design with n ={qubit number}: 1. the corresponding quantum circuit implementation with {QASM / Qiskit}. 2. the post-processing code run_and_analyze(circuit, aer_sim) in python which simulates the circuit (QuantumCircuit) with aer_sim (AerSimulator) and returns the secret string s according to the simulation results.

2. **Generation Code:** one general Qiskit code to create quantum circuits of different settings. Note that the oracle for the problem is provided as a black-box gate "oracle" here. This code is used to generate the raw data, but can also be used as a testing benchmark for writing Qiskit codes. The file is named as "{algorithm_name}_generation.py".

```python
from qiskit import QuantumCircuit


def simon_algorithm(n, oracle):
    """Generates a Simon algorithm circuit.

    Parameters:
    - n (int): number of qubits
    - s (str): the secret string of length n

    Returns:
    - QuantumCircuit: the Simon algorithm circuit
    """
    # Create a quantum circuit on 2n qubits
    simon_circuit = QuantumCircuit(2 * n, n)

    # Initialize the first register to the |+> state
    simon_circuit.h(range(n))

    # Append the Simon's oracle
    simon_circuit.append(oracle, range(2 * n))

    # Apply a H-gate to the first register
    simon_circuit.h(range(n))

    # Measure the first register
    simon_circuit.measure(range(n), range(n))

    return simon_circuit
```

Listing 1: Qiskit generation code for Simon's algorithm.

3. **Algorithm Circuit:** the OpenQASM 3.0 format file storing the quantum circuit in gate level for each specific setting. Note that the explicit construction of "Oracle" is provided separately in "oracle.inc" file, which guarantees the usage of oracle in a black-box way. This filed is named as "{algorithm_name}_n{qubit_number}.qasm".

```
OPENQASM 3.0;
include "stdgates.inc";
include "oracle.inc";
bit[3] c;
qubit[6] q;
h q[0];
h q[1];
h q[2];
Oracle q[0], q[1], q[2], q[3], q[4], q[5];
h q[0];
h q[1];
h q[2];
c[0] = measure q[0];
c[1] = measure q[1];
c[2] = measure q[2];
```

Listing 2: OpenQASM 3.0 Code for Simon's algorithm with $n = 3$.

4. **Post-Processing Function:** this function simulates the quantum circuit and derives the final answer to the problem. The file is named as "{algorithm_name}_post_processing.py".

```
from sympy import Matrix
import numpy as np
from qiskit import transpile


def mod2(x):
    return x.as_numer_denom()[0] % 2


def solve_equation(string_list):
    """
    A^T | I
    after the row echelon reduction, we can get the basis of the
        ↪ nullspace of A in I
    since we just need the string in binary form, so we can just
        ↪ use the basis
    if row == n-1 --> only one
    if row < n-1 --> get the first one (maybe correct or wrong)
    """
    M = Matrix(string_list).T

    # Augmented  : M | I
    M_I = Matrix(np.hstack([M, np.eye(M.shape[0], dtype=int)]))

    # RREF row echelon form , indices of the pivot columns
    # If x % 2 = 0, it will not be chosen as pivot (modulo 2)
    M_I_rref = M_I.rref(iszerofunc=lambda x: x % 2 == 0)

    # Modulo 2
    M_I_final = M_I_rref[0].applyfunc(mod2)

    # Non-Trivial solution
    if all(value == 0 for value in M_I_final[-1, : M.shape[1]]):
        result_s = "".join(str(c) for c in M_I_final[-1, M.shape[1]
            ↪  :])
```

```
    # Trivial solution
    else:
        result_s = "0" * M.shape[0]

    return result_s


def run_and_analyze(circuit, aer_sim):
    n = circuit.num_qubits // 2
    circ = transpile(circuit, aer_sim)
    results = aer_sim.run(circ, shots=n).result()
    counts = results.get_counts()
    equations = []
    for result, count in counts.items():
        if result != "0" * n:  # We don't use all 0 string
            y = [int(bit) for bit in result]
            equations.append(y)
    if len(equations) == 0:
        prediction = "0" * n
    else:
        prediction = solve_equation(equations)
    return prediction
```

Listing 3: Post-processing code for Simon's algorithm.

5. **Oracle / Gate Definition:** this .inc file provides the definitions of composite gates or oracles. The file is named "customgates.inc" for oracle construction tasks and "oracle.inc" for algorithm design tasks.

```
gate Oracle _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3, _gate_q_4,
    ↪ _gate_q_5 {
  cx _gate_q_0, _gate_q_3;
  cx _gate_q_1, _gate_q_4;
  cx _gate_q_2, _gate_q_5;
  cx _gate_q_2, _gate_q_5;
  x _gate_q_3;
}
```

Listing 4: One test case oracle for Simon's algorithm with $n = 3$.

For algorithm design tasks, this .inc file is accompanied with an "oracle_info.txt" file to describe the encoded information of the oracle. This helps the verification function to check the correctness of the derived answer by the model. The above test case is equipped with the following information text:

oracle_info.txt for Simon's Problem with qubit number 3 and test case 2.

Secret string: 100
Key string: 001

6. **Verification Function:** the function to evaluate the output with grammar validation and test cases verification. The file is named as "{algorithm_name}_verification.py".

```
from simon_utils import *


def check_model(qasm_string, code_string, n):
    """Check the Simon model."""
    # Verify the syntax of the QASM code with the first test case
        ↪ oracle
    t = 1
    with open(f"test_oracle/n{n}/trial{t}/oracle.inc", "r") as file
        ↪ :
```

```
        oracle_def = file.read()
    full_qasm = plug_in_oracle(qasm_string, oracle_def)
    circuit = verify_qasm_syntax(full_qasm)
    if circuit is None:
        return -1
    try:
        exec(code_string, globals())
        aer_sim = AerSimulator()
        total_success = 0
        total_fail = 0
        t_range = min(10, 4 ** (n - 2))
        shots = 10
        for t in range(1, 1 + t_range):
            print(f"    Running Test Case {t}")
            with open(f"test_oracle/n{n}/trial{t}/oracle.inc", "r")
                ↪  as file:
                oracle_def = file.read()
            full_qasm = plug_in_oracle(qasm_string, oracle_def)
            circuit = loads(full_qasm)
            with open(f"test_oracle/n{n}/trial{t}/oracle_info.txt",
                ↪  "r") as file:
                content = file.read()
            match = re.search(r"Secret string: ([01]+)", content)
            if match:
                secret_string = match.group(1)
            else:
                raise ValueError("Secret string not found in the
                    ↪ file.")

            cnt_success = 0
            cnt_fail = 0
            for shot in range(shots):
                prediction = run_and_analyze(circuit.copy(),
                    ↪ aer_sim)
                if not isinstance(prediction, str):
                    raise TypeError("Predicted secret string should
                        ↪  be a string.")
                if prediction == secret_string:
                    cnt_success += 1
                else:
                    cnt_fail += 1
            print(f"        Success: {cnt_success}/{shots}, Fail: {
                ↪ cnt_fail}/{shots}")
            total_success += cnt_success
            total_fail += cnt_fail
        print(f"Total Success: {total_success}; Total Fail: {
            ↪ total_fail}")
        return total_success / (total_fail + total_success)

    except Exception as e:
        print(f"Error: {e}")
        return -1
```

Listing 5: Verification function for Simon's algorithm.

This verification function is accompanied with an "{algorithm_name}_utils.py" file to provide necessary utility functions.

```
from qiskit.qasm3 import loads
from qiskit_aer import AerSimulator
import re


def print_and_save(message, text):
    print(message)
```

```
    text.append(message)


def plug_in_oracle(qasm_code, oracle_def):
    """Plug-in the oracle definition into the QASM code."""
    oracle_pos = qasm_code.find('include "oracle.inc";')
    if oracle_pos == -1:
        raise ValueError("Oracle include statement not found in the
            ↪ file")
    full_qasm = (
        qasm_code[:oracle_pos]
        + oracle_def
        + qasm_code[oracle_pos + len('include "oracle.inc";') :]
    )
    return full_qasm


def verify_qasm_syntax(output):
    """Verify the syntax of the output and return the corresponding
        ↪ QuantumCircuit (if it is valid)."""
    assert isinstance(output, str)
    try:
        # Parse the OpenQASM 3.0 code
        circuit = loads(output)
        print(
            "    The OpenQASM 3.0 code is valid and has been
                ↪ successfully loaded as a QuantumCircuit."
        )
        return circuit
    except Exception as e:
        print(f"    Error: The OpenQASM 3.0 code is not valid.
            ↪ Details: {e}")
        return None
```

Listing 6: Utility functions for verification of Simon's algorithm.

7. **Dataset Creation Script:** this script involves all the code necessary to create the data points from scratch. The file is named as "{algorithm_name}_dataset.py". The main function looks like this:

```
    def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "-f",
        "--func",
        choices=["qasm", "json", "gate", "check"],
        help="The function to call: generate qasm circuit, json
            ↪ dataset or extract gate definition.",
    )
    args = parser.parse_args()
    if args.func == "qasm":
        generate_circuit_qasm()
    elif args.func == "json":
        generate_dataset_json()
    elif args.func == "gate":
        extract_gate_definition()
    elif args.func == "check":
        check_dataset()
```

Listing 7: Main function of the dataset script for Simon's algorithm.

Here the "generate_circuit_qasm()" function generates the raw data of quantum circuits in OpenQASM 3.0 format where the algorithm circuit and the oracle definition are blended, then "extract_gate_definition()" function extracts the definition of oracles and formulates the algorithm circuits into the format suitable for model output. The "check_dataset()" function is used to check

the correctness of the created data points and "generate_dataset_json()" function to combine the data into json format for easy integration with the benchmarking pipeline.

A.2 Discussion of more Tasks

Problem Encoding. In Section 4.1.1, we mentioned another category of oracle construction tasks referred to as "Problem Encoding", which involves applying quantum algorithms, such as Grover's algorithm, to solve practical problems such as SAT and triangle finding. The crux of this process is encoding the problem constraints into Grover's oracle, thereby making this a type of oracle construction task. Unlike quantum logic synthesis, which encodes an explicit function $f(x)$ as a unitary operator U_f, this task involves converting the constraints of a particular problem into the required oracle form. We provide implementations of several concrete problems in this directory as demonstrations and will include more applications in future work.

Quantum Information Protocols. In the "Quantum Information" section of the "Algorithm Design" task, we have also implemented three important quantum information protocols: Quantum Teleportation, Superdense Coding, and Quantum Key Distribution (BB84). A brief introduction to these protocols can be found in Appendix B. We did not include the experiments for these protocols as they involve communication between two parties, which is challenging to characterize with a single OpenQASM 3.0 file. We recommend revising the post-processing function as a general classical function to schedule the communication and processing between different parties specifically for these protocols. The fundamental quantum circuits and processing codes are provided in the repository.

A.3 Datasheet

Here we present a datasheet for the documentation of QCircuitNet.

Motivation

- *For what purpose was the dataset created?* It was created as a benchmark for the capability of designing and implementing quantum algorithms for LLMs.
- *Who created the dataset (e.g., which team, research group) and on behalf of which entity (e.g., company, institution, organization)?* The authors of this paper.
- *Who funded the creation of the dataset?* We will reveal the funding resources in the Acknowledgement section of the final version.

Composition

- *What do the instances that comprise the dataset represent (e.g., documents, photos, people, countries)?* The dataset comprises problem description, generation code, algorithm circuit, post-processing function, oracle / gate definition, verification function, and dataset creation script for various quantum algorithms.
- *How many instances are there in total (of each type, if appropriate)?* The dataset has 5 algorithms for oracle construction task and 10 algorithms for algorithm design task used for experiments. There are 3 quantum information protocols and additional problem encoding tasks not included for experiments.
- *Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set?* The dataset contains instances with restricted qubit numbers due to the current scale of real quantum hardware.
- *What data does each instance consist of?* Qiskit codes, OpenQASM 3.0 codes, python scripts, and necessary text information.
- *Are relationships between individual instances made explicit?* Yes, the way to create different instances are clearly described in Appendix A.1.

- *Are there recommended data splits?* Yes, we recommend splitting the data according to different algorithms in algorithm design task.
- *Are there any errors, sources of noise, or redundancies in the dataset?* There might be some small issues due to the dumping process of Qiskit and programming mistakes (if any).
- *Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)?* The dataset is self-contained.
- *Does the dataset contain data that might be considered confidential (e.g., data that is protected by legal privilege or by doctor-patient confidentiality, data that includes the content of individuals' non-public communications)?* No.
- *Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety?* No.

Collection Process

- *How was the data associated with each instance acquired?* The data is created by first composing Qiskit codes for each algorithm and then converting to OpenQASM 3.0 files using qiskit.qasm3.dump function, with additional processing procedure.
- *What mechanisms or procedures were used to collect the data (e.g., hardware apparatuses or sensors, manual human curation, software programs, software APIs)?* Manual human programming and Qiskit APIs.
- *Who was involved in the data collection process (e.g., students, crowd workers, contractors), and how were they compensated (e.g., how much were crowd workers paid)?* Nobody other than the authors of the paper.
- *Over what timeframe was the data collected?* The submitted version of the dataset was created in May and June 2024.

Uses

- *Has the dataset been used for any tasks already?* It has been used in this paper to benchmark LLM's ability for quantum algorithm design.
- *Is there a repository that links to any or all papers or systems that use the dataset?* The only paper which uses the dataset for now is this paper.

Distribution

- *Will the dataset be distributed to third parties outside of the entity (e.g., company, institution, organization) on behalf of which the dataset was created?* Yes, the dataset will be made publicly available on the Internet after the review process.
- *How will the dataset be distributed (e.g., tarball on website, API, GitHub)?* It will be distributed on the GitHub platform.
- *Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)?* The dataset is distributed under CC BY 4.0.
- *Have any third parties imposed IP-based or other restrictions on the data associated with the instances?* No.
- *Do any export controls or other regulatory restrictions apply to the dataset or to individual instances?* No.

Maintenance

- *Who will be supporting/hosting/maintaining the dataset?* The authors of this paper.

- *How can the owner/curator/manager of the dataset be contacted (e.g., email address)?* The email for contact will be provided after the review process.
- *Is there an erratum?* Not at this time.
- *Will the dataset be updated (e.g., to correct labeling errors, add new instances, delete instances)?* Yes, it will be continually updated.
- *If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?* Yes, they can do so with the GitHub platform.

A.4 Copyright and Licensing Terms

This work is distributed under a CC BY 4.0 license. The implementation of the code references open-source projects such as Qiskit, QuantumKatas, Cirq, and NWQBench. We bear responsibility in case of violation of rights.

B Additional Preliminaries for Quantum Computing and Quantum Information

Quantum circuit diagram. A quantum algorithm is composed of a series of quantum gates. By default, a quantum algorithm starts from the all-0 state $|0^n\rangle$. A quantum algorithm can be illustrated by its quantum gate diagram, drawn from left to right. The initial all-0 state is placed at the left side of the diagram. After that, whenever we apply a quantum gate, it is placed on the corresponding qubits, from left to right. At the end of the quantum gates, we need to measure and read the outputs, and these measurements are placed at the right side of the diagram. See Figure 4 for the quantum gate diagram of Simon's algorithm [Simon, 1997].

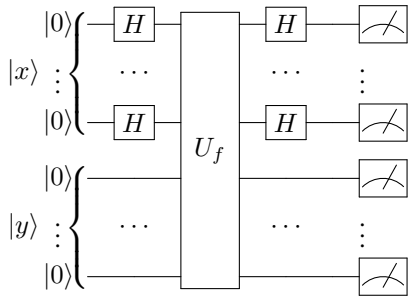

Figure 4: Quantum gate diagram of Simon's algorithm.

Superdense coding. Superdense coding [Bennett and Wiesner, 1992] is a quantum communication protocol that allows Alice to transmit two classical bits of information to Bob by sending only one qubit, given that they share a pair of entangled qubits. The protocol can be divided into five steps:

1. **Preparation:** Charlie prepares a maximally entangled Bell state, such as $|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.
2. **Sharing:** Charlie sends the qubit 1 to Alice and the qubit 2 to Bob. Alice and Bob can be separated by an arbitrary distance.
3. **Encoding:** Depending on the two classical bits $zx \in \{00, 01, 10, 11\}$ that Alice wants to send, she applies the corresponding quantum gate operation to her qubit, transforming the

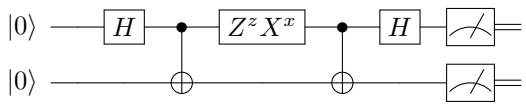

Figure 5: Quantum circuit diagram for superdense coding.

Bell state $|\beta_{00}\rangle$ into one of the four Bell states:

$$|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \text{ if } zx = 00$$
$$|\beta_{01}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \text{ if } zx = 01$$
$$|\beta_{10}\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \text{ if } zx = 10$$
$$|\beta_{11}\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle) \text{ if } zx = 11$$

Alice achieves these transformations by applying the operation $Z^z X^x$ to her qubit, where Z is the phase-flip gate, X is the bit-flip gate. Specifically:

- If $zx = 00$, Alice applies $Z^0 X^0 = I$ (identity gate).
- If $zx = 01$, Alice applies $Z^0 X^1 = X$ (bit-flip gate).
- If $zx = 10$, Alice applies $Z^1 X^0 = Z$ (phase-flip gate).
- If $zx = 11$, Alice applies $Z^1 X^1 = ZX = iY$ gate.

4. **Sending:** Alice sends her qubit to Bob through a quantum channel.
5. **Decoding:** Bob applies a CNOT gate followed by a Hadamard gate to the two qubits, transforming the entangled state into the corresponding computational basis state $|zx\rangle$. By measuring the qubits, Bob obtains the two classical bits zx sent by Alice.

Superdense coding exploits the properties of quantum entanglement to transmit two classical bits of information using only one qubit. The quantum circuit diagram for superdense coding is shown in Figure 5.

Quantum teleportation. Quantum teleportation [Bennett et al., 1993] is a technique for transferring quantum information from a sender (Alice) to a receiver (Bob) using shared entanglement and classical communication. The protocol can be described as follows:

1. **Preparation:** Telamon prepares a maximally entangled Bell state, such as $|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.
2. **Sharing:** Alice has qubit 1 in the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, which she wants to teleport to Bob. Telamon shares qubit 2 with Alice and qubit 3 with Bob, creating the shared entangled state $|\beta_{00}\rangle_{23}$.
3. **Encoding:** Alice wants to teleport an unknown quantum state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ to Bob. She applies a CNOT gate to qubits 1 and 2, with qubit 1 as the control and qubit 2 as the target. Then, she applies a Hadamard gate to qubit 1. The resulting state of the three-qubit system is:

$$|\Psi\rangle = \frac{1}{2}[|\beta_{00}\rangle(\alpha|0\rangle + \beta|1\rangle) + |\beta_{01}\rangle(\alpha|1\rangle + \beta|0\rangle) + |\beta_{10}\rangle(\alpha|0\rangle - \beta|1\rangle) + |\beta_{11}\rangle(\alpha|1\rangle - \beta|0\rangle)].$$

4. **Measurement:** Alice measures qubits 1 and 2 in the Bell basis and obtains one of four possible outcomes: $|\beta_{00}\rangle$, $|\beta_{01}\rangle$, $|\beta_{10}\rangle$, or $|\beta_{11}\rangle$. This measurement collapses the three-qubit

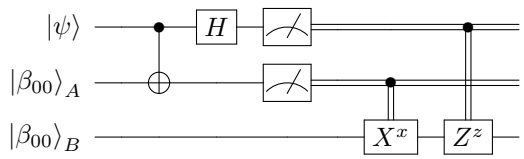

Figure 6: Quantum circuit diagram for quantum teleportation

state into one of the following:

$$|\beta_{00}\rangle \otimes (\alpha|0\rangle + \beta|1\rangle)$$
$$|\beta_{01}\rangle \otimes (\alpha|1\rangle + \beta|0\rangle)$$
$$|\beta_{10}\rangle \otimes (\alpha|0\rangle - \beta|1\rangle)$$
$$|\beta_{11}\rangle \otimes (\alpha|1\rangle - \beta|0\rangle)$$

5. **Classical Communication:** Alice sends the result of her measurement (two classical bits) to Bob via a classical channel.
6. **Reconstruction:** Depending on the classical information received from Alice, Bob applies the operation $Z^z X^x$ to qubit 3, where z and x correspond to the two classical bits sent by Alice:
   - If Alice measured $|\beta_{00}\rangle$, she sends $zx = 00$, and Bob applies $Z^0X^0 = I$ (identity operation).
   - If Alice measured $|\beta_{01}\rangle$, she sends $zx = 01$, and Bob applies $Z^0X^1 = X$ (bit-flip).
   - If Alice measured $|\beta_{10}\rangle$, she sends $zx = 10$, and Bob applies $Z^1X^0 = Z$ (phase-flip).
   - If Alice measured $|\beta_{11}\rangle$, she sends $zx = 11$, and Bob applies $Z^1X^1 = ZX = iY$ (bit-flip and phase-flip).

   After applying the appropriate operation, Bob's qubit 3 will be in the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, which is the original state that Alice wanted to teleport.

The quantum circuit diagram for quantum teleportation is shown in Figure 6.

Quantum key distribution. Quantum key distribution (QKD) [Bennett and Brassard, 1984] is a secure communication protocol that allows two parties, Alice and Bob, to produce a shared random secret key, which can then be used to encrypt and decrypt messages. The security of QKD is based on the fundamental principles of quantum mechanics that measuring a qubit can change its state. One of the most well-known QKD protocols is the BB84 protocol, which works as follows:

1. Alice randomly generates a bit string and chooses a random basis (X or Z) for each bit. She then encodes the bits into qubits using the chosen bases and sends them to Bob through a quantum channel.
2. Bob measures the received qubits in randomly chosen bases (X or Z) and records the results.
3. Alice and Bob communicate over a public classical channel to compare their basis choices. They keep only the bits for which their basis choices coincide and discard the rest.
4. Alice and Bob randomly select a subset of the remaining bits and compare their values. If the error rate is below a certain threshold, they conclude that no eavesdropping has occurred, and the remaining bits can be used as a secret key. If the error rate is too high, they abort the protocol, as it indicates the presence of an eavesdropper (Eve).

The security of the BB84 protocol relies on the fact that any attempt by Eve to measure the qubits during transmission will introduce detectable errors, alerting Alice and Bob to the presence of an eavesdropper.

C Additional Experiment Results

C.1 Metrics

BLEU Score. Bilingual Evaluation Understudy (BLEU) score is a metric used to evaluate the quality of machine-translated text compared to human-translated text. It measures how close the machine translation is to one or more reference translations. The BLEU score evaluates the quality of text generated by comparing it with one or more reference texts. It does this by calculating the n-gram precision, which means it looks at the overlap of n-grams (contiguous sequences of n words) between the generated text and the reference text. Originally the BLEU score ranges from 0 to 1, where 1 indicates a perfect match with the reference translations. Here rescaling the score makes it ranges from 0 to 100.

The BLEU score, originally designed for machine translation, can also be effectively used for evaluating algorithm generation tasks. Just as BLEU measures the similarity between machine-translated text and human reference translations, it can measure the similarity between a generated algorithm and a gold-standard algorithm. This involves comparing sequences of tokens to assess how closely the generated output matches the reference solution. In the context of algorithm generation, n-grams can represent sequences of tokens or operations in the code. BLEU score captures the precision of these n-grams, ensuring that the generated code aligns closely with the expected sequences found in the reference implementation.

The formula for BLEU score is given by:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right).$$

where BP is the acronym for brevity penalty, w_n is the weight for the n-gram precision (typically $\frac{1}{N}$ for uniform weights), p_n is the precision for n-grams. BP is calculated as:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-\frac{r}{c}} & \text{if } c \leq r \end{cases}.$$

where c is the length of the generated text and r is the length of the reference text. Furthermore, n-gram precision p_n is calculated as:

$$p_n = \frac{\sum_{C \in \text{Candidates}} \sum_{n-\text{gram} \in C} \min(\text{Count}(n-\text{gram in candidate}), \text{Count}(n-\text{gram in references}))}{\sum_{C \in \text{Candidates}} \sum_{n-\text{gram} \in C} \text{Count}(n-\text{gram in candidate})}.$$

This formulation ensures that the BLEU score takes into account both the precision of the generated n-grams and the overall length of the translation, providing a balanced evaluation metric.

Byte Perplexity. Perplexity is a measure of how well a probability distribution or a probabilistic model predicts a sample. In the context of language models, it quantifies the uncertainty of the model when it comes to predicting the next element in a sequence. Byte perplexity specifically deals with sequences of bytes, which are the raw binary data units used in computer systems. For our purposes, we consider byte perplexity under UTF-8 encoding, a widely used character encoding standard that represents each character as one or more bytes.

For a given language model, let $p(x_i|x_{<i})$ be the probability of the i-th byte x_i given the preceding bytes $x_{<i}$. If we have a sequence of bytes $x = (x_1, x_2, \ldots, x_N)$, the perplexity $PP(x)$ of the model on this sequence is defined as:

$$PP(x) = 2^{-\frac{1}{N}\sum_{i=1}^{N} \log_2 p(x_i|x_{<i})}.$$

A notable feature of byte perplexity is that, it does not rely on any specific tokenizer, making it versatile for comparing different models. Therefore, byte perplexity can be used to measure the

performance in quantum algorithm generation tasks. In such tasks, a lower byte perplexity indicates a better-performing model, as it means the model is more confident in its predictions of the next byte in the sequence.

C.2 Experiment Results

Due to considerable variances in the experiments, we conducted additional rounds to obtain more representative data.

The BLEU scores for various quantum algorithm design tasks are illustrated in Figure 7. This figure not only displays the average performance of each model but also highlights the differences in performance across individual quantum algorithm tasks. The first notable observation is that the figure clearly demonstrates the varying levels of difficulty among quantum algorithms. For example, models achieve higher BLEU scores on tasks such as Bernstein-Vazirani and Deutsch-Jozsa, whereas they perform significantly worse on tasks like Grover, phase estimation, and quantum Fourier transform. This indicates that the former tasks are considerably easier than the latter ones. Another significant observation is that most models score higher in a five-shot prompt compared to a one-shot prompt, which confirms the large language models' ability to improve performance through contextual learning.

Similar patterns are observed in oracle construction tasks, as illustrated in Figure 8. The figure highlights that the Diffusion Operator task is notably more challenging than the Grover oracle construction task. Interestingly, we found that adding more in-context examples actually reduced the performance of the Phi-3-medium-128k-instruct and Mistral-7B-v0.3 models. This decline in performance could be attributed to the significant differences between each oracle construction task, which may be too out-of-distribution. Consequently, the additional examples might cause the models to overfit to the specific examples provided in the context, rather than generalizing well across different tasks.

In addition to evaluating the BLEU score, we conducted an experiment to measure the correctness of the machine-generated algorithms, and the results are shown in Table 3.[3] By running a verification function, we discovered that phase estimation and the swap test are significantly more challenging than other problems, leading most models to score -1 (indicating they cannot even generate the correct syntax). Notably, the BLEU score for the swap test is above average compared to other algorithms, yet almost none of the models produced a correct algorithm. This discrepancy highlights a critical limitation of using BLEU as a metric for algorithm evaluation. BLEU measures average similarity, but even a single mistake in an algorithm can render it entirely incorrect, thus failing to capture the true accuracy and functionality of the generated algorithms. Another important finding is that in a five-shot setting, GPT-4 and GPT-3.5 surpass all other models by a large margin. This demonstrates their exceptional capabilities, particularly in long-context comprehension and in-context learning. These models not only excel in understanding and generating text based on minimal examples but also maintain high performance over extended sequences, highlighting their advanced architecture and training methodologies.

[3]When we prepare supplementary materials, we observe that these experiments have considerable variances, and they are hence executed by additional rounds to obtain more representative data. As a result, we refreshed the data in Table 1 and Table 2 in the main body, and we also place Table 3 and Table 4 here.

Table 3: Benchmarking algorithm design in verification function scores.

Model	Shot	Bernstein-Vazirani	Deutsch-Jozsa	Grover	Phase Estimation	Quantum Fourier Transform	Simon	GHZ State	Random Number Generator	Swap Test	W State	Average
gpt-4o-2024-05-13	1	-0.8462	-0.5538	-0.7089	-1.0000	-1.0000	-0.6692	-0.8462	-1.0000	-1.0000	-1.0000	-0.8624
gpt-4o-2024-05-13	5	-0.3054	0.0135	-0.2071	-0.5357	-0.6154	-0.3692	-0.1538	-0.4967	-0.8700	-0.9231	-0.4463
Meta-Llama-3-8B	1	-0.2308	-0.7692	-0.7143	-0.8571	-0.9231	-1.0000	-0.6154	-0.9285	-1.0000	-0.3846	-0.7423
Meta-Llama-3-8B	5	0.0769	-0.2308	-0.5393	-1.0000	-0.7692	-0.8462	-0.3846	-0.7276	-1.0000	-0.1538	-0.5575
gpt-3.5-turbo-0125	1	-0.8462	-0.7154	-0.5679	-1.0000	-1.0000	-0.6231	-0.8462	-1.0000	-1.0000	-1.0000	-0.8599
gpt-3.5-turbo-0125	5	-0.6154	-0.0571	-0.0500	-1.0000	-0.6538	-0.1646	-0.2308	-0.4513	-0.8778	-0.8462	-0.4947
Phi-3-medium-128k-instruct	1	-0.8462	-0.7750	-1.0000	-1.0000	-1.0000	-1.0000	-0.3846	-1.0000	-0.8878	-0.8462	-0.8740
Phi-3-medium-128k-instruct	5	-0.6577	-0.3821	-0.8286	-1.0000	-1.0000	-0.6100	-0.9231	-0.3569	-0.8333	-0.8462	-0.7438
Mistral-7B-v0.3	1	-0.8462	-0.8590	-0.7107	-1.0000	-1.0000	-0.9192	-0.7692	-1.0000	-1.0000	-0.6923	-0.8797
Mistral-7B-v0.3	5	-0.6246	-0.6667	-0.4071	-0.8571	-0.9231	-0.9115	-0.6923	-0.8820	-1.0000	-0.5385	-0.7503

The verification results of the oracle construction task, as shown in Table 4, confirm our previous conclusions. In the five-shot setting, GPT-4 and GPT-3.5 consistently outperform all other models. Additionally, this table highlights the inconsistency between BLEU scores and verification scores. For instance, while the Diffusion Operator task achieves the lowest BLEU score, it is the Grover oracle construction that receives the lowest verification score. This discrepancy suggests that BLEU scores may not fully capture the performance of models in certain complex tasks.

Table 4: Benchmarking oracle construction in verification function scores.

Model	Shot	Bernstein-Vazirani	Deutsch-Jozsa	Diffusion-Operator	Grover	Simon	Average
gpt-4o-2024-05-13	1	-0.3200	-0.0100	-0.8462	-0.9885	-0.4674	-0.5264
gpt-4o-2024-05-13	5	-0.1100	0.0800	-0.3077	-0.9540	-0.0870	-0.2757
Meta-Llama-3-8B	1	-0.7300	-0.5000	-0.3846	-1.0000	-0.6848	-0.6599
Meta-Llama-3-8B	5	-0.0500	0.1700	-0.8462	-1.0000	-0.6413	-0.4735
gpt-3.5-turbo-0125	1	-0.3500	-0.0400	-0.8462	-1.0000	-0.3696	-0.5211
gpt-3.5-turbo-0125	5	-0.1100	0.0200	-0.3077	-0.9770	-0.1087	-0.2967
Phi-3-medium-128k-instruct	1	-0.6800	-0.6100	-0.9231	-1.0000	-0.7500	-0.7926
Phi-3-medium-128k-instruct	5	-0.5400	-0.4300	-1.0000	-1.0000	-0.8370	-0.7614
Mistral-7B-v0.3	1	-0.4000	-0.4300	-0.9231	-0.9540	-0.6087	-0.6632
Mistral-7B-v0.3	5	-0.3700	-0.1300	-1.0000	-0.9195	-0.2391	-0.5317

The Byte Perplexity results, shown in Figure 9 and Figure 10, provide valuable insights into the performance of our model. Evaluated in a zero-shot setting, byte perplexity trends closely mirror those observed with BLEU scores. This alignment suggests that our model's predictive capabilities are consistent across Perplexity and BLEU evaluation metrics. Specifically, in the context of quantum algorithm design tasks, the results indicate that the Bernstein-Vazirani and Deutsch-Jozsa algorithms are relatively straightforward for the model, whereas the Simon algorithm presents greater difficulty. This differentiation highlights the varying levels of complexity inherent in these quantum algorithms.

C.3 Case Study

After carefully examining the model's output, we observed several interesting patterns. We present a series of case studies to illustrate these observations and provide possible explanations.

Low Score for GPT-4o in One-Shot Setting. At first glance, it is surprising that GPT-4o performs poorly on many quantum algorithms in the algorithm design task in the one-shot setting compared to Llama3-8B. Given that Llama3-8B has a relatively smaller parameter scale, the results should have been the other way around. A closer examination of the model's output reveals the potential reason: while Llama3-8B closely mimics the input examples, GPT-4o tends to improvise, resulting in outputs that are not well captured by the current syntax support. Here are several concrete examples.

This is the OpenQASM 3.0 code output for the W state with $n = 7$. In this code, GPT-4o uses the advanced "for" loop syntax newly introduced in OpenQASM 3.0 to create the circuit. Although the code fails to produce the W state, it is syntactically correct. However, the Qiskit.qasm3 import module, which converts OpenQASM 3.0 files to QuantumCircuit objects and is used in our verification

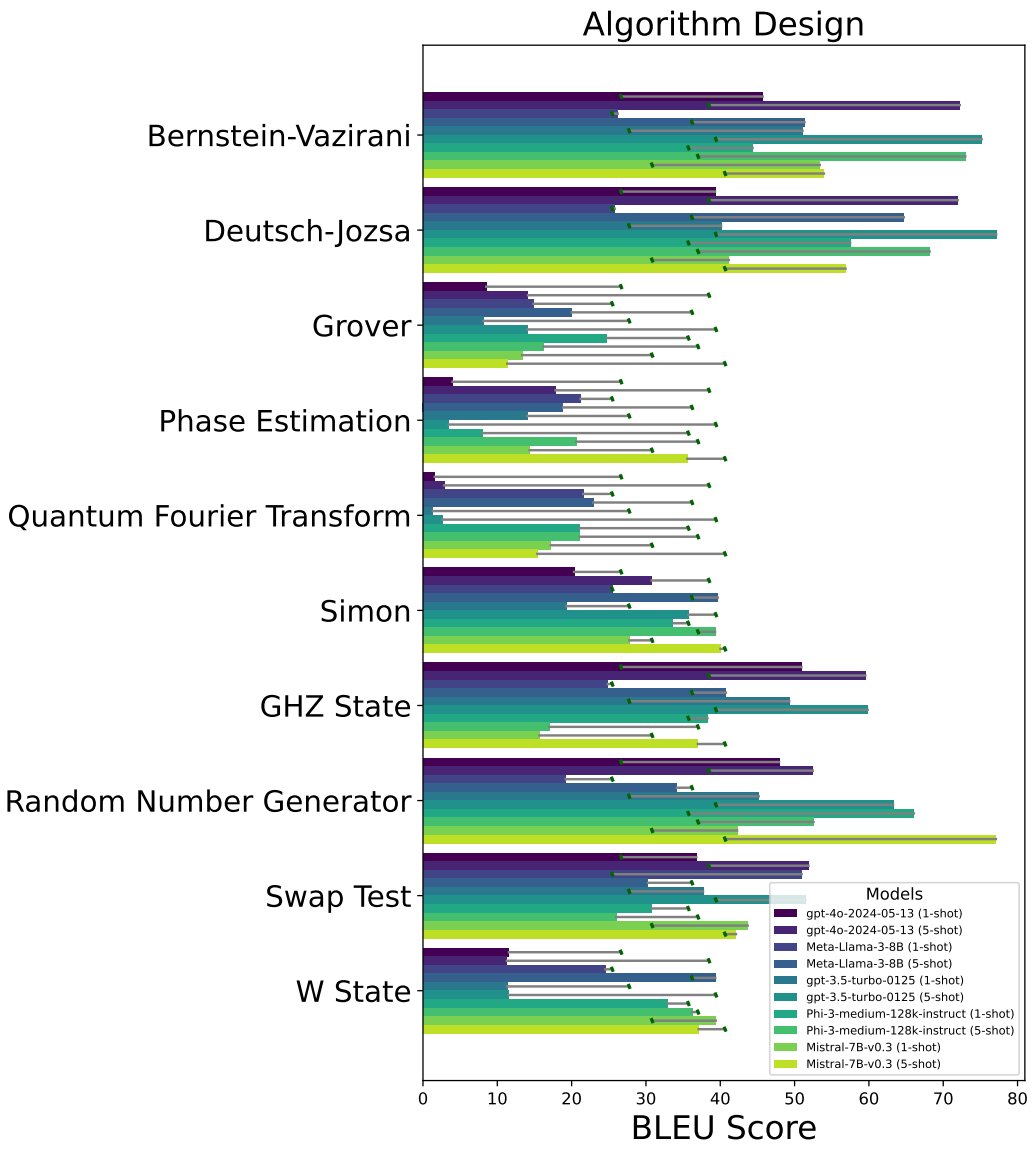

Figure 7: Benchmarking algorithm design in BLEU scores. The green dots represent each model's mean BLEU score across 10 algorithms, while the gray lines show how much its score on each algorithm deviates from this mean.

function to check the correctness of the syntax of output OpenQASM codes, is still in the experimental stage and does not support many of OpenQASM 3.0's advanced features. As a result, GPT-4o's use of these features causes the code to fail syntax validation, resulting in a score of -1.

```
OPENQASM 3.0;
include "stdgates.inc";
qubit[7] q;
h q[0];
for i in[1:6] {
    cx q[i-1], q[i];
}
```

Listing 8: OpenQASM 3.0 Code output by GPT-4o for W state with $n = 7$.

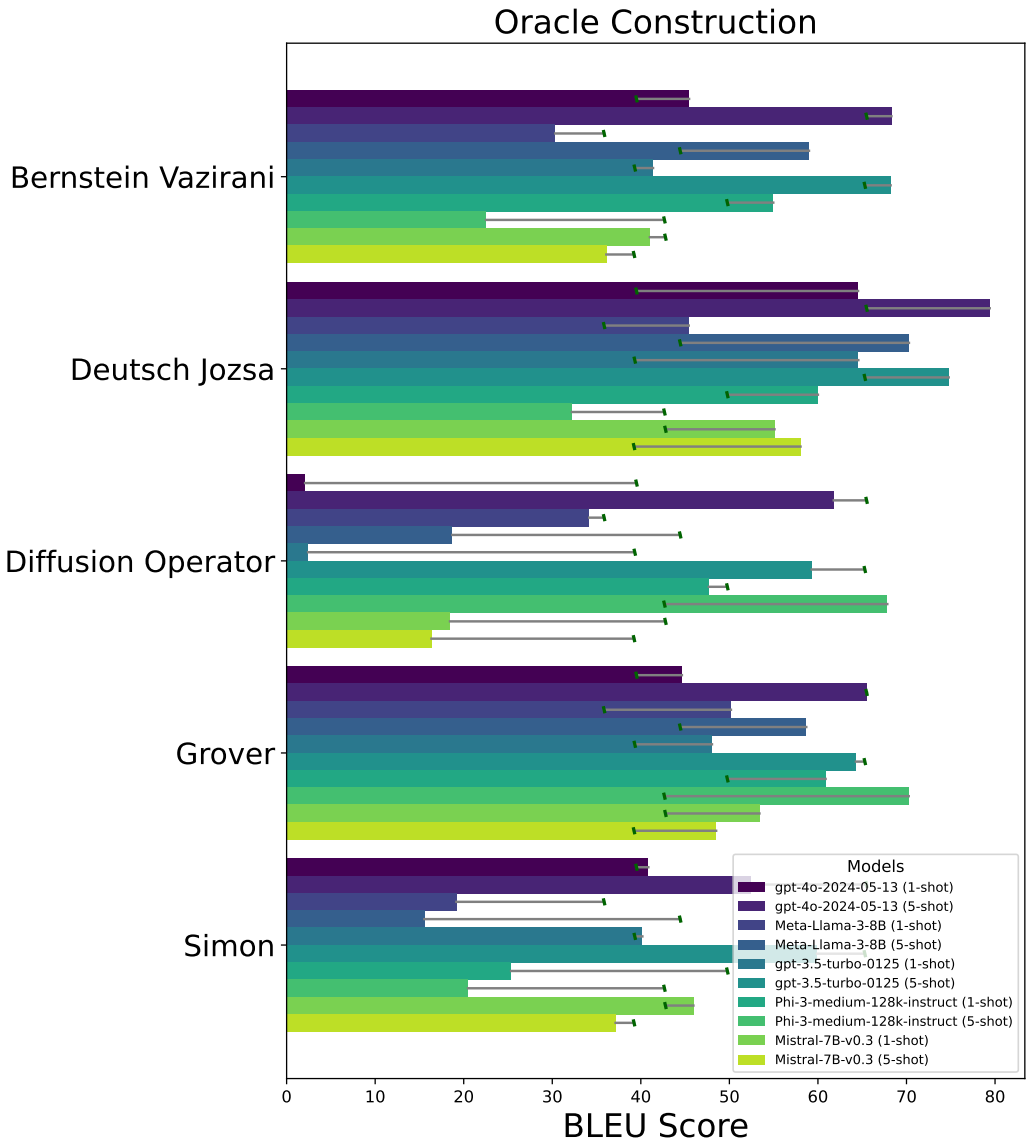

Figure 8: Benchmarking oracle construction in BLEU scores. The green dots represent each model's mean BLEU score across 5 oracles, while the gray lines show how much its score on each oracle deviates from this mean

Here is another example where GPT-4o decides to assign novel names to its qubit registers, leading to a conflict in the symbol table in Scope.GLOBAL. If we substitute all the registers x, y, and s with new names, the code can pass syntax validation successfully and is close to the correct solution.

```
OPENQASM 3.0;
include "stdgates.inc";
include "oracle.inc";
bit[9] s;
qubit[10] x;
qubit[11] y;
h x[0];
h x[1];
h x[2];
h x[3];
```

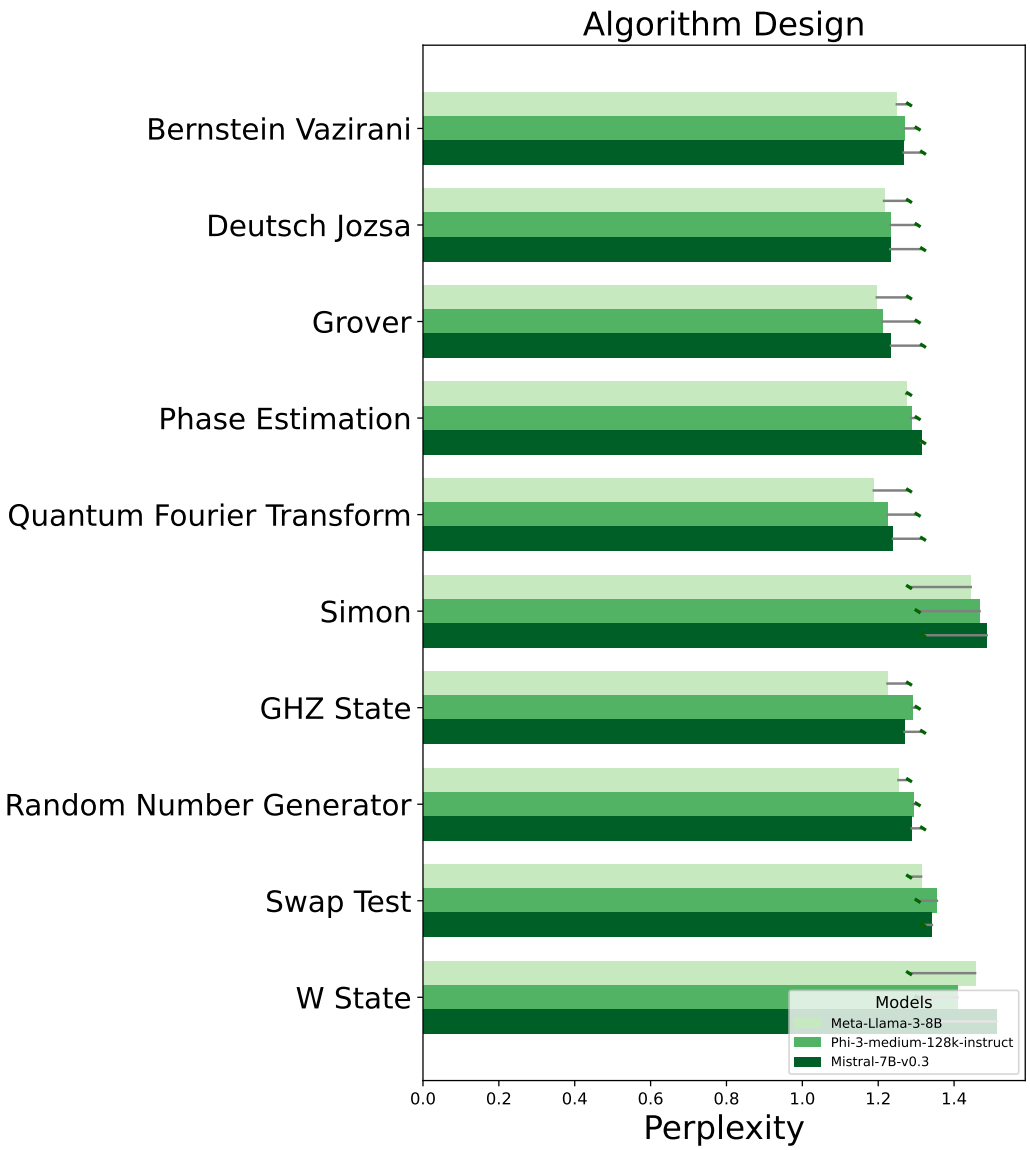

Figure 9: Benchmarking algorithm design in perplexity. The green dots represent each model's mean perplexity score across 10 algorithms, while the gray lines show how much its score on each algorithm deviates from this mean.

```
h x[4];
h x[5];
h x[6];
h x[7];
h x[8];
Oracle x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], y;
h x[0];
h x[1];
h x[2];
h x[3];
h x[4];
h x[5];
h x[6];
h x[7];
```

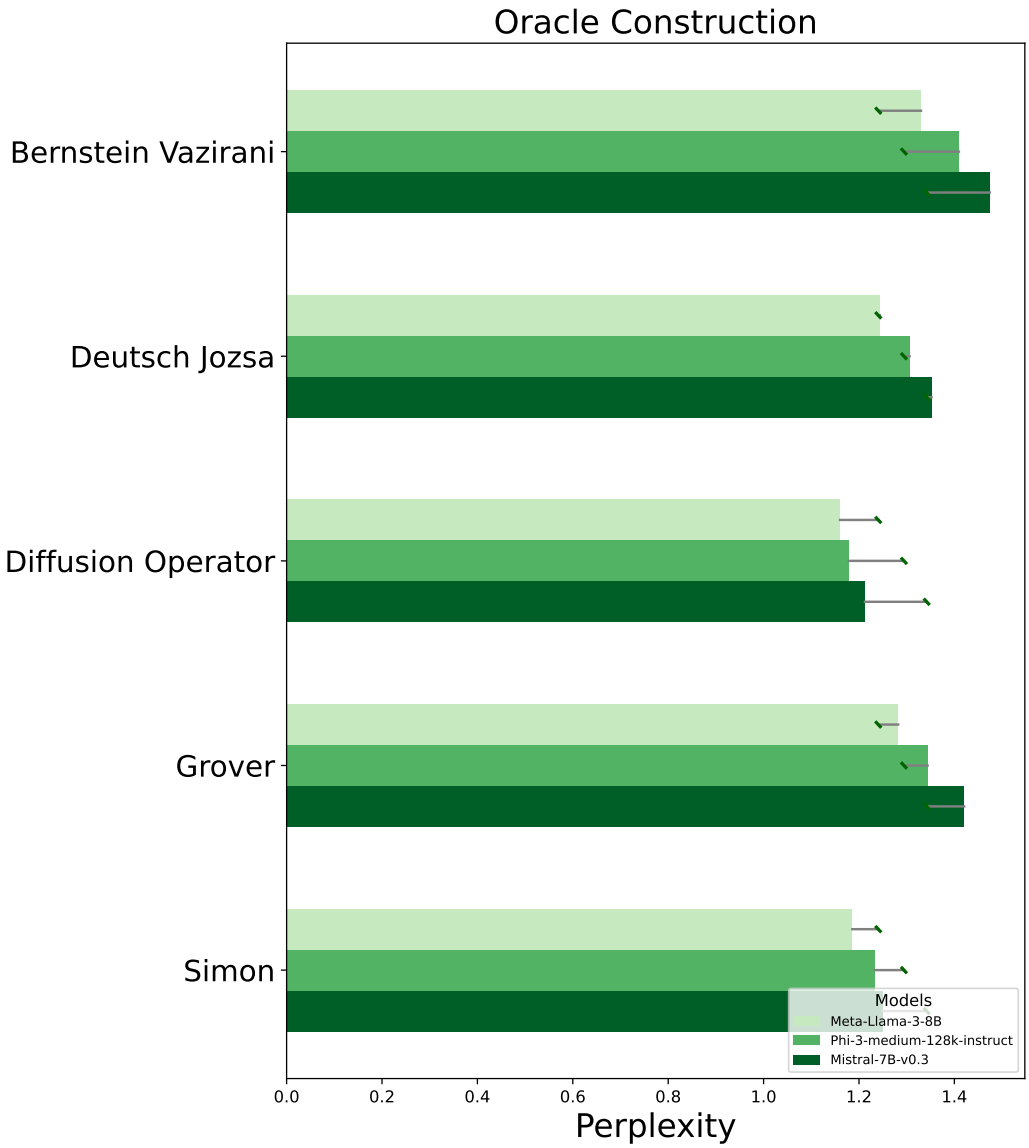

Figure 10: Benchmarking oracle construction in perplexity. The green dots represent each model's mean perplexity score across 5 oracles, while the gray lines show how much its score on each oracle deviates from this mean.

```
h x[8];
measure x[0] -> s[0];
measure x[1] -> s[1];
measure x[2] -> s[2];
measure x[3] -> s[3];
measure x[4] -> s[4];
measure x[5] -> s[5];
measure x[6] -> s[6];
measure x[7] -> s[7];
measure x[8] -> s[8];
```

Listing 9: OpenQASM 3.0 Code output by GPT-4o for Bernstein-Vazirani Problem with $n = 9$.

Since Llama3-8B tends to follow the provided example more closely, it avoids errors like this. These factors may help explain why GPT-4o performs poorly in the one-shot setting compared to Llama3-8B.

High Score for GPT-4o in Five-Shot Setting. In contrast to its relatively low score in the one-shot setting, GPT-4o achieves the best overall performance in the five-shot setting for both oracle construction and algorithm design tasks.

Unlike in the one-shot setting, where GPT-4o frequently uses advanced OpenQASM 3.0 features, it learns from the examples in context and uses simpler syntax in the five-shot setting. For the W state task, here is an output in the five-shot setting:

```
OPENQASM 3.0;
include "stdgates.inc";
qubit[3] q;
h q[0];
cx q[0], q[1];
h q[1];
cx q[1], q[2];
h q[2];
```

Listing 10: OpenQASM 3.0 Code output by GPT-4o for W state with $n = 3$.

This output avoids the use of "for" loops and successfully passes the syntax validation test, although it still cannot generate the W state correctly.

In addition to adapting to plain syntax through in-context learning, GPT-4o achieves outstanding performance on more complicated tasks such as phase estimation. Here is the model output of GPT-4o on the phase estimation task with qubit number $n = 2$.

```
OPENQASM 3.0;
include "stdgates.inc";
include "oracle.inc";
bit[2] c;
qubit[2] q;
Psi q[0];
h q[1];
CU_0 q[0], q[1];
h q[1];
c[0] = measure q[1];
c[1] = measure q[0];
```

Listing 11: OpenQASM 3.0 Code output by GPT-4o for Phase Estimation with $n = 2$.

```
from qiskit import transpile


def run_and_analyze(circuit, aer_sim):
    """Run the circuit and analyze the result."""
    circ = transpile(circuit, aer_sim)
    result = aer_sim.run(circ, shots=1).result()
    counts = result.get_counts()
    phase_str = list(counts.keys())[0]
    phase = int(phase_str, 2) / 2**2
    return phase
```

Listing 12: Post-processing code output by GPT-4o for Phase Estimation with $n = 2$.

This suite of OpenQASM 3.0 circuits and post-processing functions successfully outputs the phase within the required precision for the test case, resulting in an impressive verification score of 1.0. Despite the small number of qubits and differences from the reference implementation, the accuracy achieved is noteworthy.

These phenomena reflect that GPT-4o has impressive in-context learning abilities and overall better capabilities in designing and implementing quantum algorithms.