QCircuitBench: A Large-Scale Dataset for Benchmarking Quantum Algorithm Design

Rui Yang Ziruo Wang Yuntian Gu Yitao Liang **Tongyang Li**





Peking University

Accepted by NeurIPS 2025

International Workshop on Quantum Boltzmann Machines (IW-QBM)

December 10, 2025

Contents

QCircuitBench

Contents

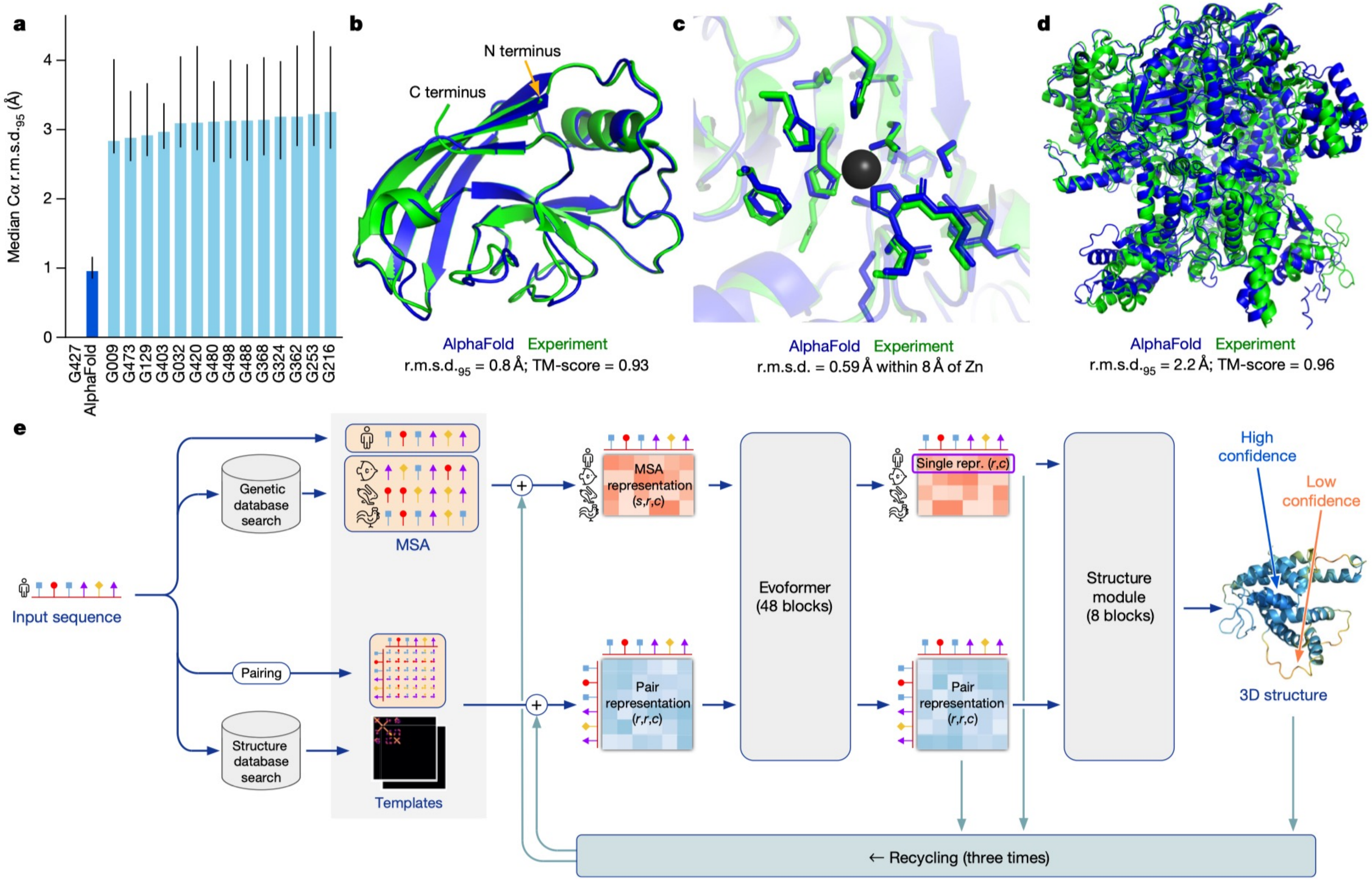
QCircuitBench

Trends in AI applications: AI for Science

AlphaFold

Predicting the 3D structure of proteins based on amino acid sequence.

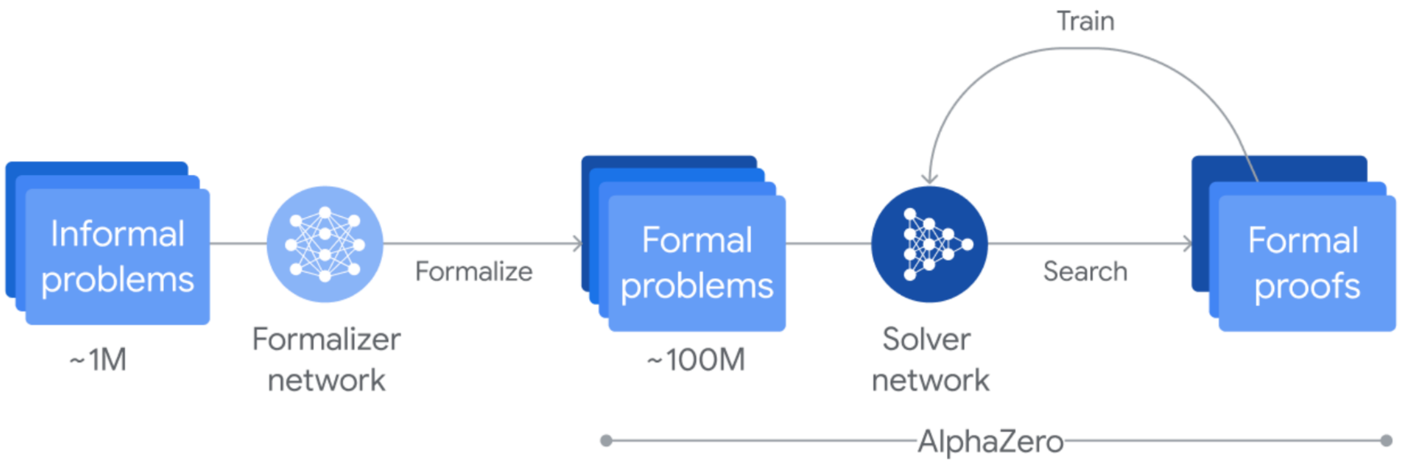
2024 **Nobel Prize** in Chemistry.

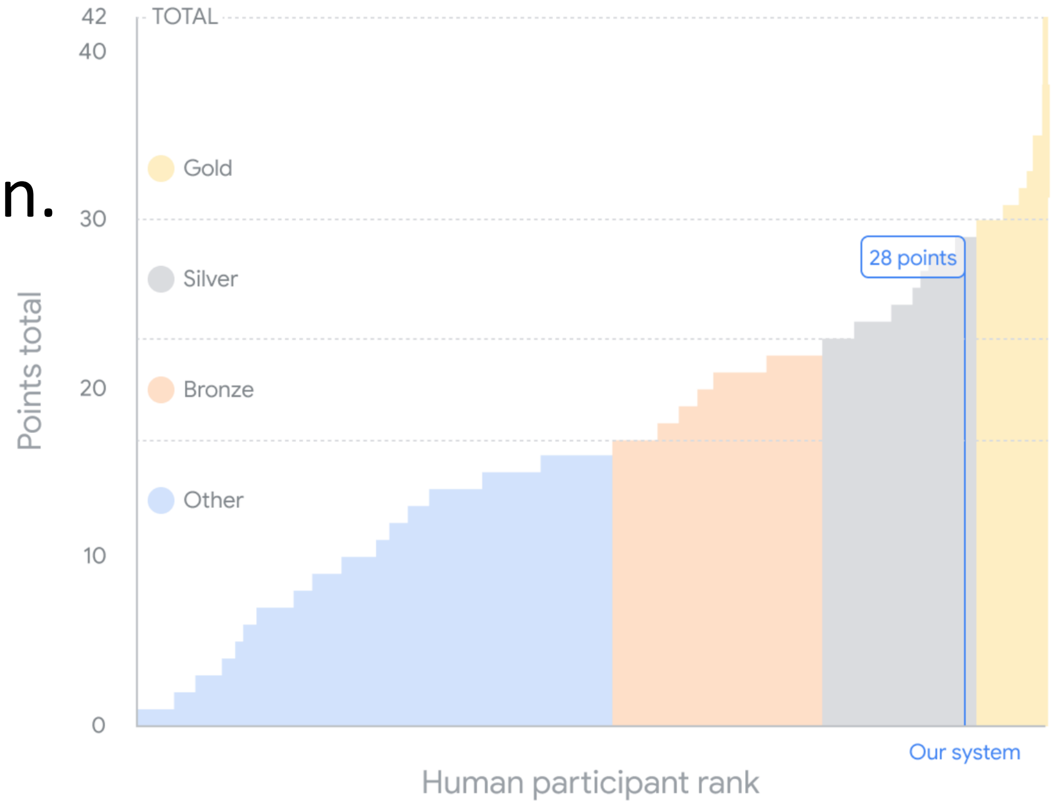


Trends in AI applications: AI for Science

AlphaProof

Achieved a silver medal in the IMO competition.





Trends in AI applications: AI for Science ⟶ LLM for Math

Generative Language Modeling for Automated Theorem Proving

Stanislas Polu
OpenAI
spolu@openai.com

Ilya Sutskever
OpenAI
ilyasu@openai.com

Abstract

We explore the application of transformer-based language models to automated theorem proving. This work is motivated by the possibility that a major limitation of automated theorem provers compared to humans – the generation of original mathematical terms – might be addressable via generation from language models. We present an automated prover and proof assistant, *GPT-f*, for the Metamath formalization language, and analyze its performance. *GPT-f* found new short proofs that were accepted into the main Metamath library, which is to our knowledge, the first time a deep learning based system has contributed proofs that were adopted by a formal mathematics community.

1 Introduction

Artificial neural networks have enjoyed a spectacularly successful decade, having made considerable advances in computer vision [1, 2], translation [3, 4, 5], speech recognition [6, 7], image generation [8, 9, 10, 11, 12], game playing [13, 14, 15], and robotics [16, 17]. Especially notable is the recent rapid progress in language understanding and generation capabilities [18, 19, 20, 21, 22].

With the possible exception of AlphaGo [13] and AlphaZero [23], reasoning tasks are conspicuously absent from the list above. In this work we take a step towards addressing this absence by applying a transformer language model to automated theorem proving.

deepseek

DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition

Z.Z. Ren*, Zhihong Shao*, Junxiao Song*, Huajian Xin†, Haocheng Wang†, Wanjia Zhao†, Liyue Zhang, Zhe Fu
Qihao Zhu, Dejian Yang, Z.F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao
Daya Guo, Chong Ruan

DeepSeek-AI

https://github.com/deepseek-ai/DeepSeek-Prover-V2

Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction

Yong Lin[1]*, Shange Tang[1 2]*, Bohan Lyu[3]*, Ziran Yang[1]*, Jui-Hui Chung[1]*, Haoyu Zhao[1]*, Lai Jiang[7]*, Yihan Geng[8]*, Jiawei Ge[1], Jingruo Sun[4], Jiayun Wu[3], Jiri Gesi[6]†, Ximing Lu[2], David Acuna[2], Kaiyu Yang[5]‡, Hongzhou Lin[6]*†, Yejin Choi[2 4], Danqi Chen[1], Sanjeev Arora[1], Chi Jin[1]*
[1]Princeton Language and Intelligence, Princeton University [2]NVIDIA
[3]Tsinghua University [4]Stanford University [5]Meta FAIR [6]Amazon
[7]Shanghai Jiao Tong University [8]Peking University

Trends in AI applications:

Quadratic to superpolynomial speedup

AI for Science → AI for $\boxed{\text{Quantum Computing}}$

Challenging to design manually

Dataset for quantum computing is solicited!

QCircuitBench

Contributions

First large-scale benchmark for AI-driven quantum algorithm design

- **Task Formulation**: a carefully designed framework capturing the core aspects of quantum algorithm design.
- **Rich Algorithm Coverage**: covers 3 task suites, 25 algorithms, and 120,290 data points, supporting complex, scalable algorithm implementation.
- **Automatic Verification**: built-in validation tools, enabling human-free, iterative evaluation and interactive reasoning.
- **Training Potential**: demonstrates promise as a training dataset via preliminary fine-tuning experiments.

Contents

QCircuitBench

Challenges

What challenges do we need to tackle?

Formulation: Natural Language? verbose, ambiguous (✗)
Math formulas? precise, but hard to verify automatically (✗)

Oracle Paradox: Theoretically: black-box.
Experimentally: explicit construction with quantum gates.

Classical Procedure: Quantum Algorithm = Quantum Circuit + Interpretation of Measurement Results.

Design Principles

Challenges

Formulation: Natural Language? (✗)
Math formulas? (✗)

Oracle Paradox: Theoretically: black-box.
Experimentally: explicit gates.

Classical Procedure: Quantum Algorithm = Quantum Circuit + Interpretation of Measurement Results.

Solutions

A **code generation** perspective

Represent quantum algorithms with quantum programming languages.

A Separate oracle.inc library

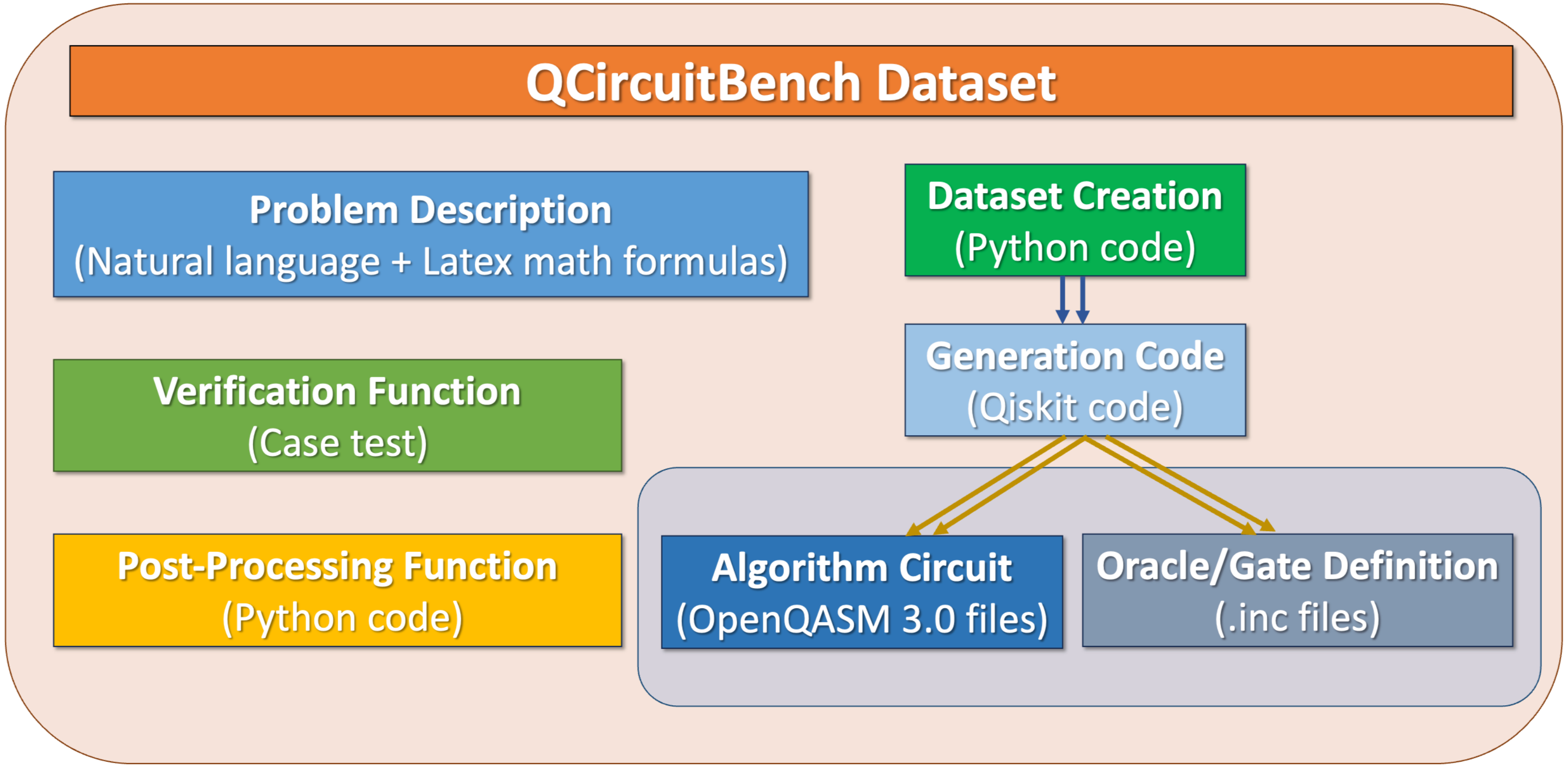
Preserve black-box abstraction while enabling compilation in OpenQASM.

Require post-processing functions

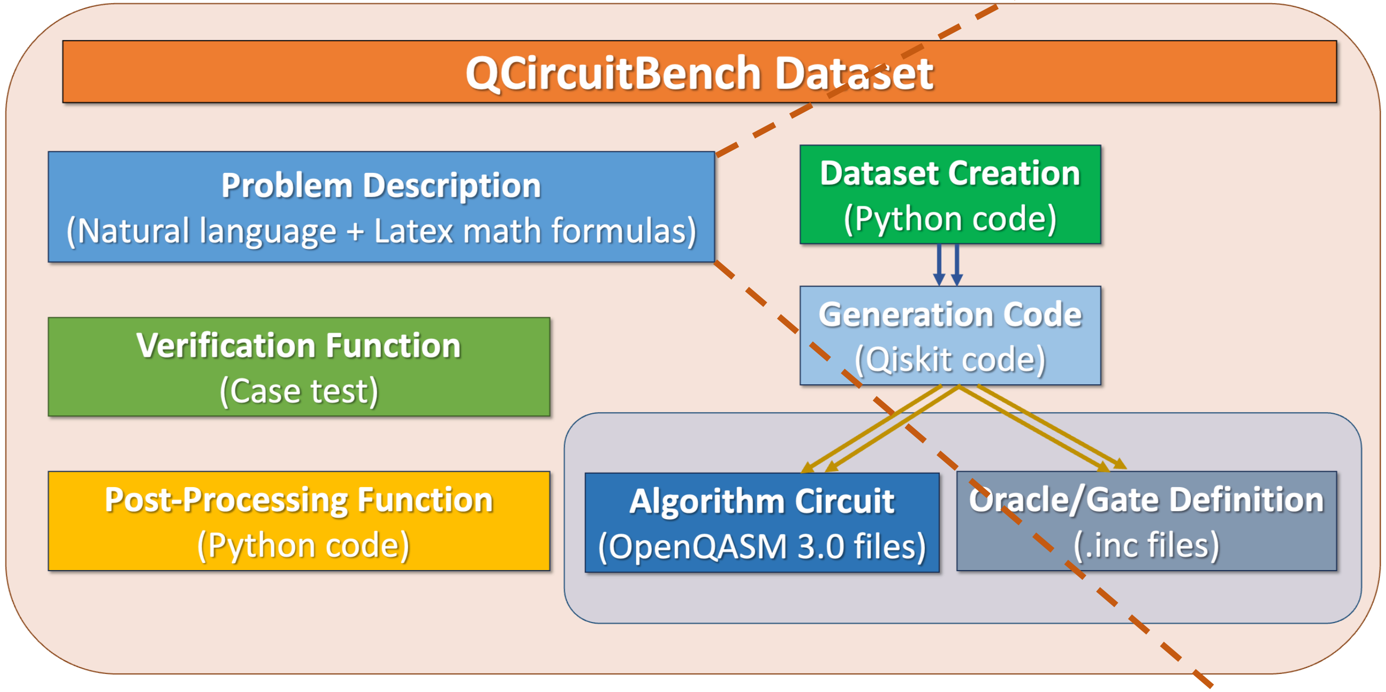
Include number of shots to characterize query complexity.

QCircuitBench Framework

A general framework which formulates the key features of quantum algorithm design task for Large Language Models.



QCircuitBench Framework

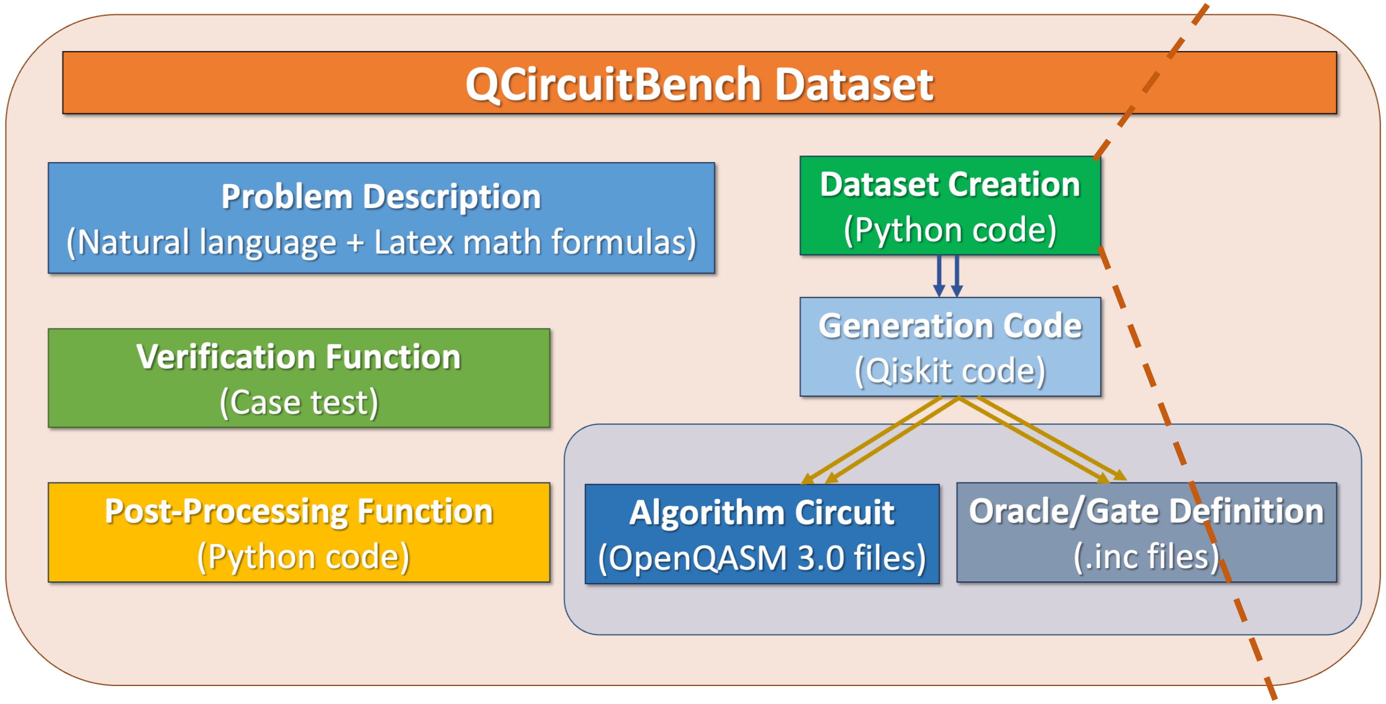


1. Problem Description

- Carefully hand-crafted prompts.
- Natural language + latex math formulas.
- Interfaces of quantum oracle or composite gates.

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 $n = 3$: 1. the corresponding quantum circuit implementation with QASM. 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.

QCircuitBench Framework



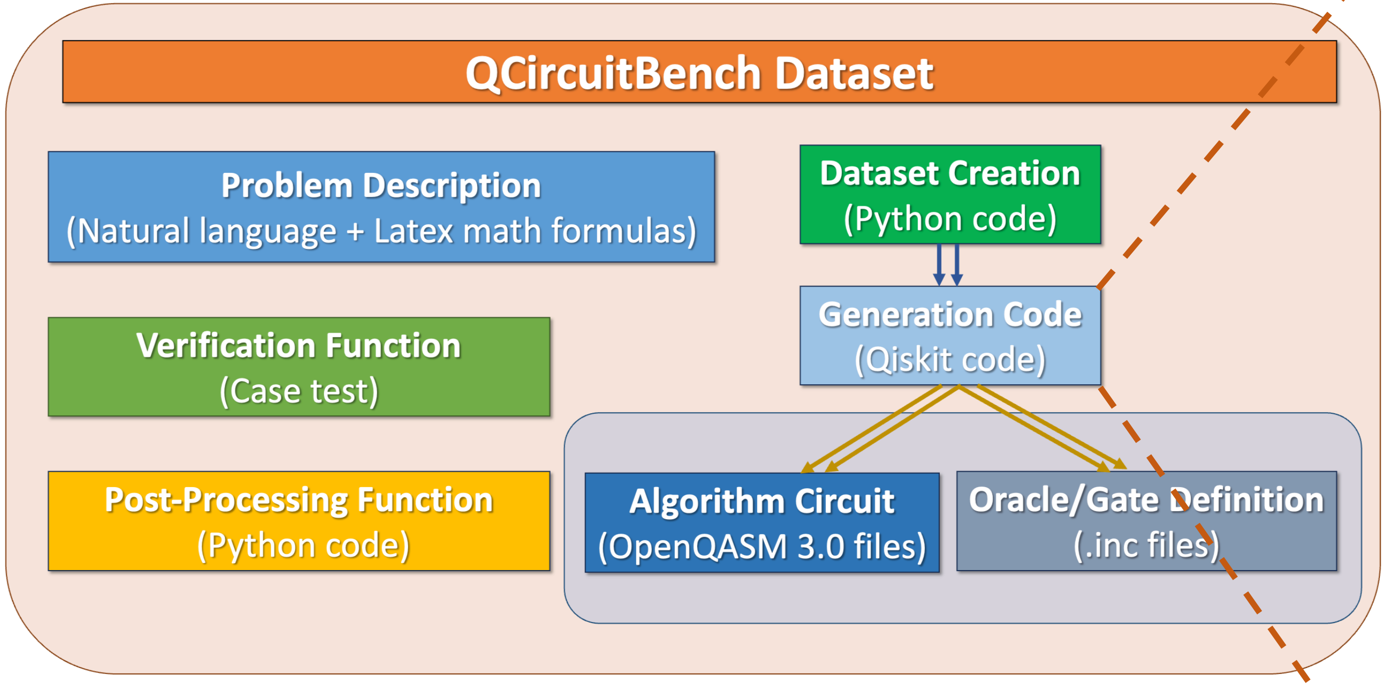
2. Dataset Creation Script

Create the dataset from scratch:

- Generate primitive QASM circuits.
- Extract gate definitions.
- Validate the data points.
- Create benchmark pipeline.

```python
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()
```

QCircuitBench Framework

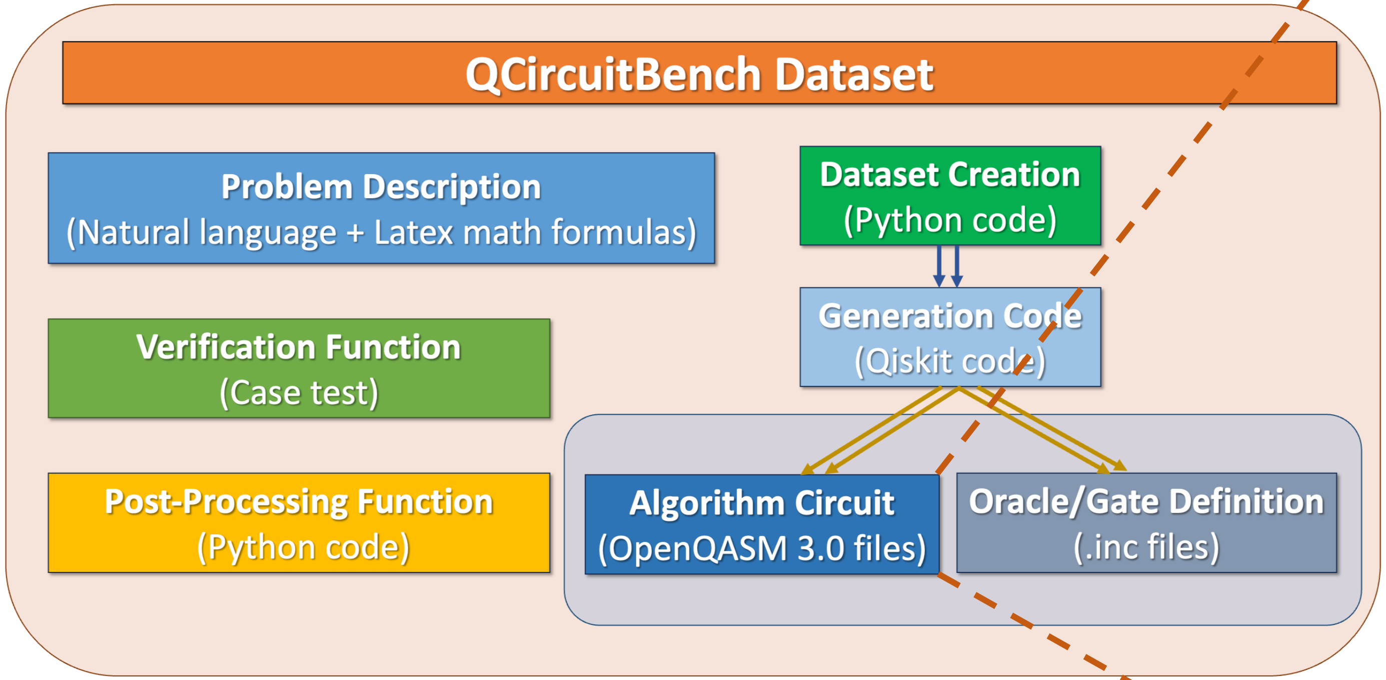


3. Generation Code

- Create quantum circuits for algorithms of different settings (secret strings / qubit numbers).

```python
from Qiskit import QuantumCircuit
def simon_algorithm(n, oracle):
    # 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
```

QCircuitBench Framework

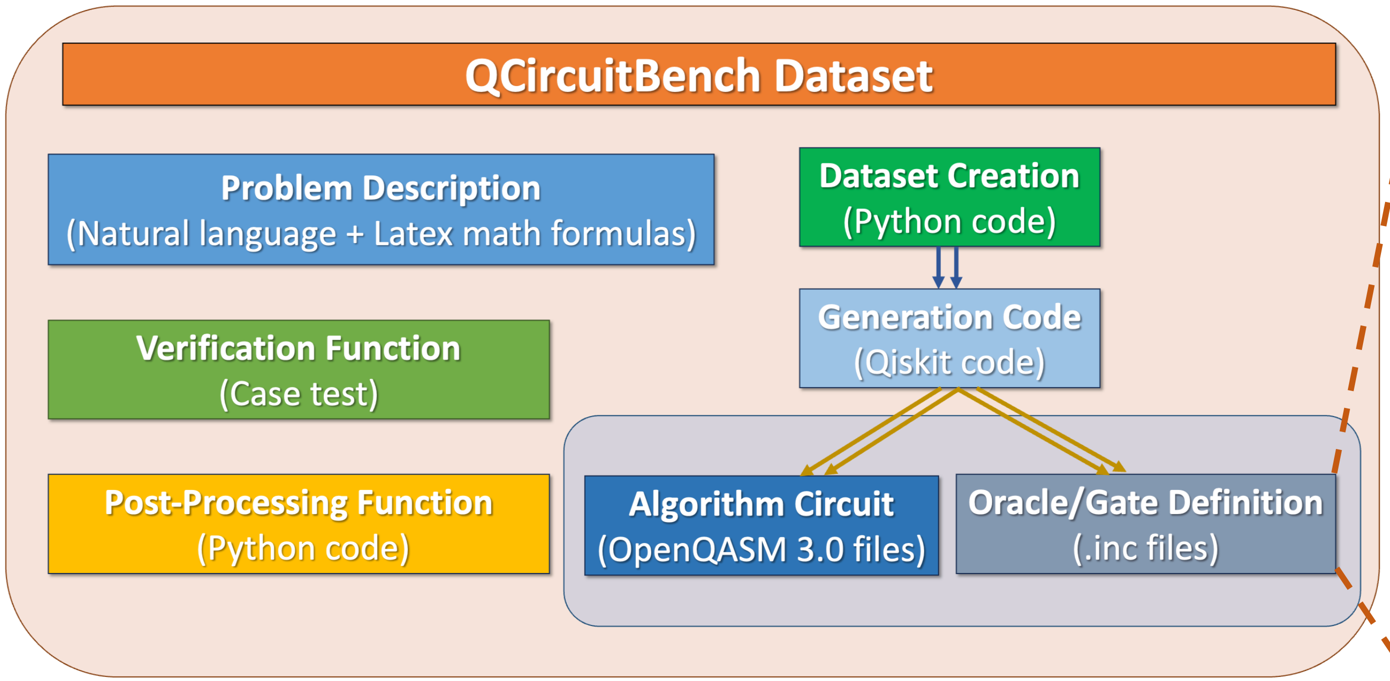


4. Algorithm Circuit

- A .qasm file storing the quantum circuit for each specific setting.
- Adopt OpenQASM 3.0 to explicitly save the circuits at gate level.

```
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];
```

QCircuitBench Framework

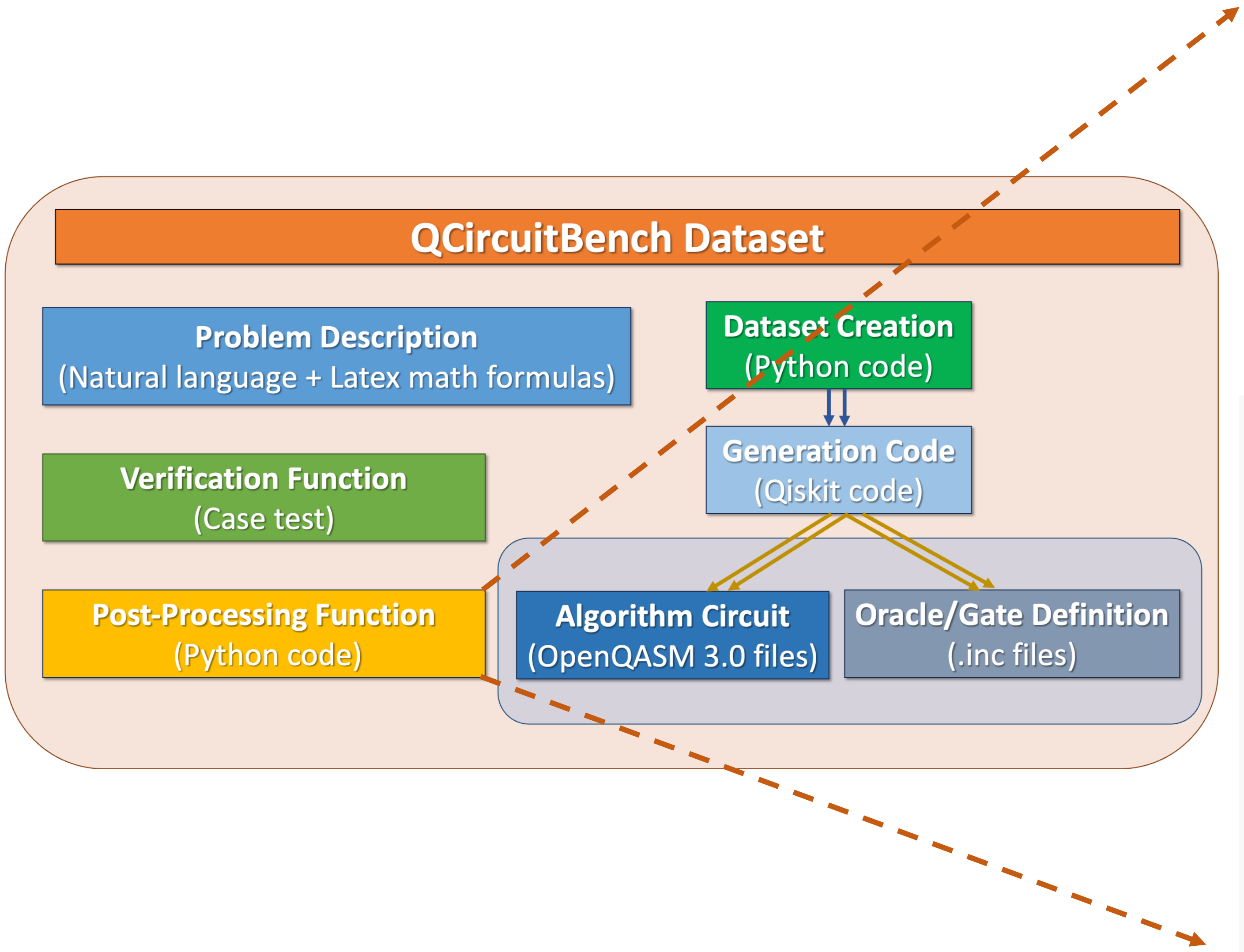


5. Oracle / Gate Definition

- A .inc file to provide definitions of oracles or composite gates.
- Delivers the oracle in a black-box way.

```
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_4;
}
```

QCircuitBench Framework



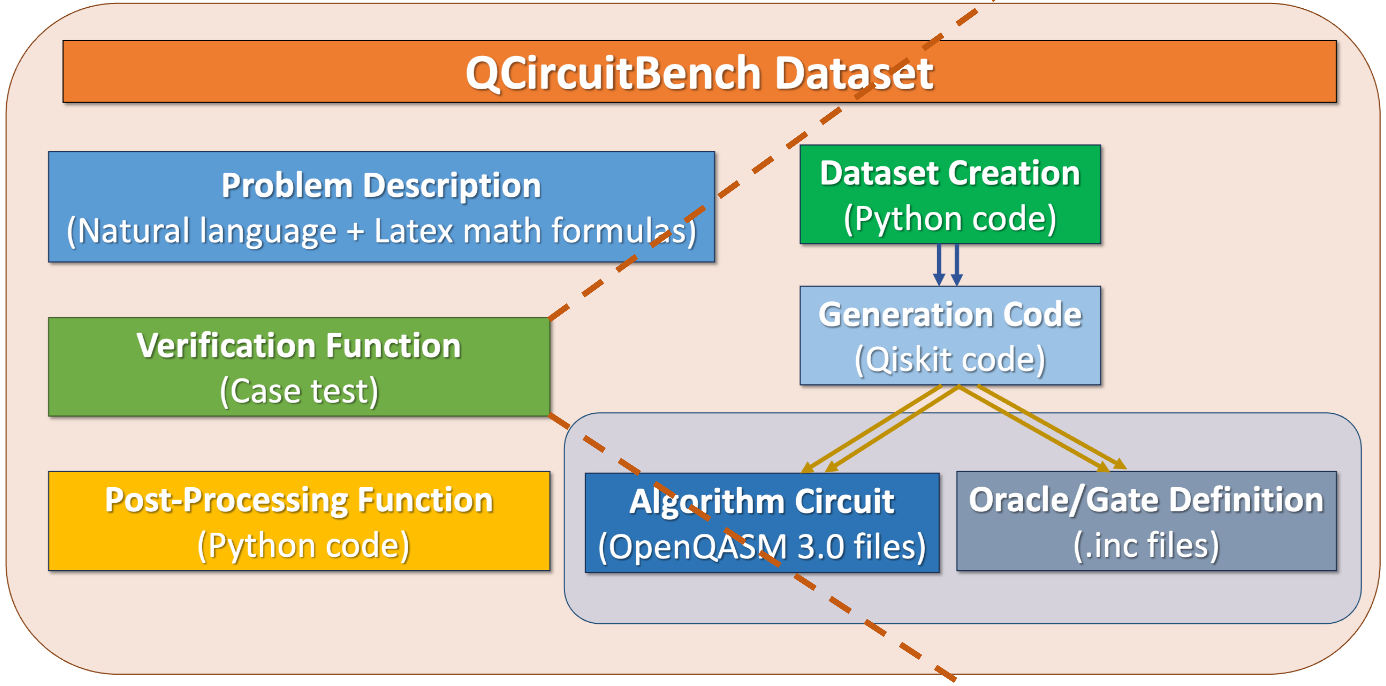
6. Post-Processing Function

- For Algorithm Design task only.
- Uses Qiskit AerSimulator to execute the quantum circuit, and returns the answer to the original problem.

```python
def solve_equation(string_list):
    M = Matrix(string_list).T
    M_I = Matrix(np.hstack([M, np.eye(M.shape[0], dtype=int)]))
    M_I_rref = M_I.rref(iszerofunc=lambda x: x % 2 == 0)
    M_I_final = M_I_rref[0].applyfunc(mod2)
    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] :])
    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 = [list(map(int, result)) for result in counts if result != "0" * n]
    prediction = solve_equation(equations) if len(equations) > 0 else "0" * n
    return prediction
```

QCircuitBench Framework



7. Verification Function

- Evaluate the implemented algorithm.
- The function returns two scores: **syntax** score and **semantic** score.
- If the program fails to run successfully, a detailed error message is provided as feedback.

```python
def check_model(qasm_string, code_string, n):
    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
```

Task Suite

❖ Oracle Construction

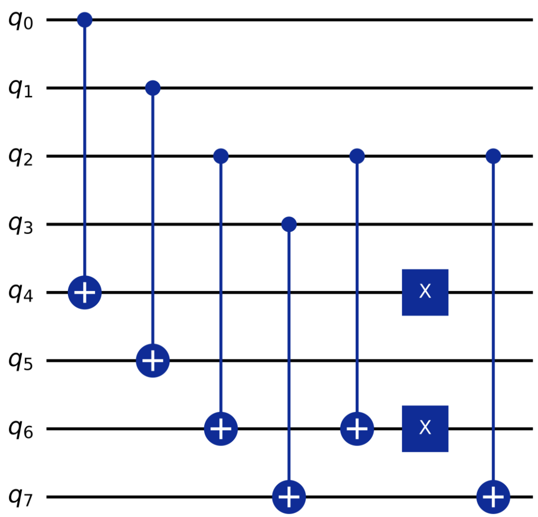
Encode Boolean function f as an oracle U_f such that $U_f|x\rangle|z\rangle = |x\rangle|z \oplus f(x)\rangle$.

❖ Quantum Algorithm Design

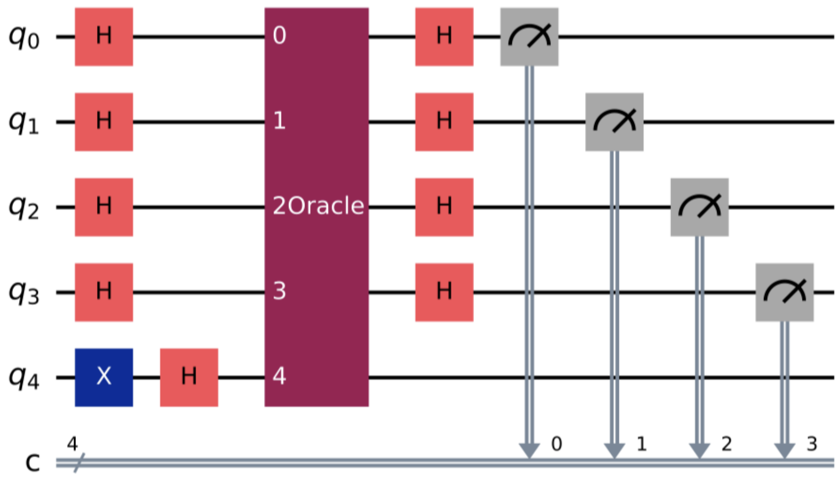
Covers textbook-level algorithms to advanced applications.

❖ Random Circuit Synthesis

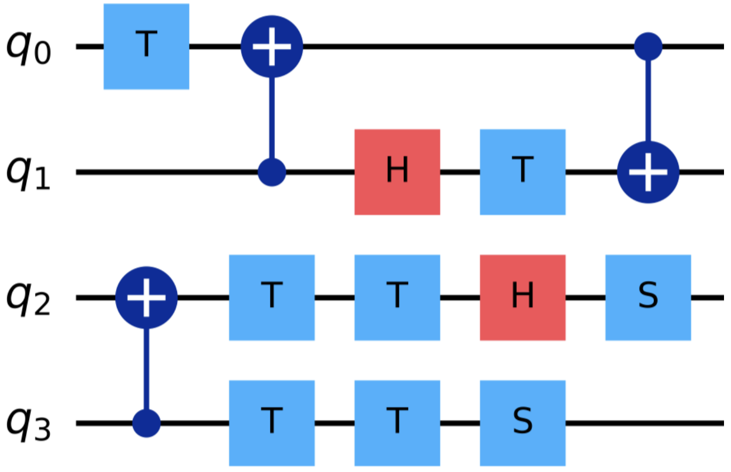
Reproduce quantum states from Clifford set {H, S, CNOT} / universal set {H, S, T, CNOT}.



(a) Simon's Problem (s=1100)



(b) Deutsch-Jozsa Algorithm



(c) Universal Circuits

Task Suite

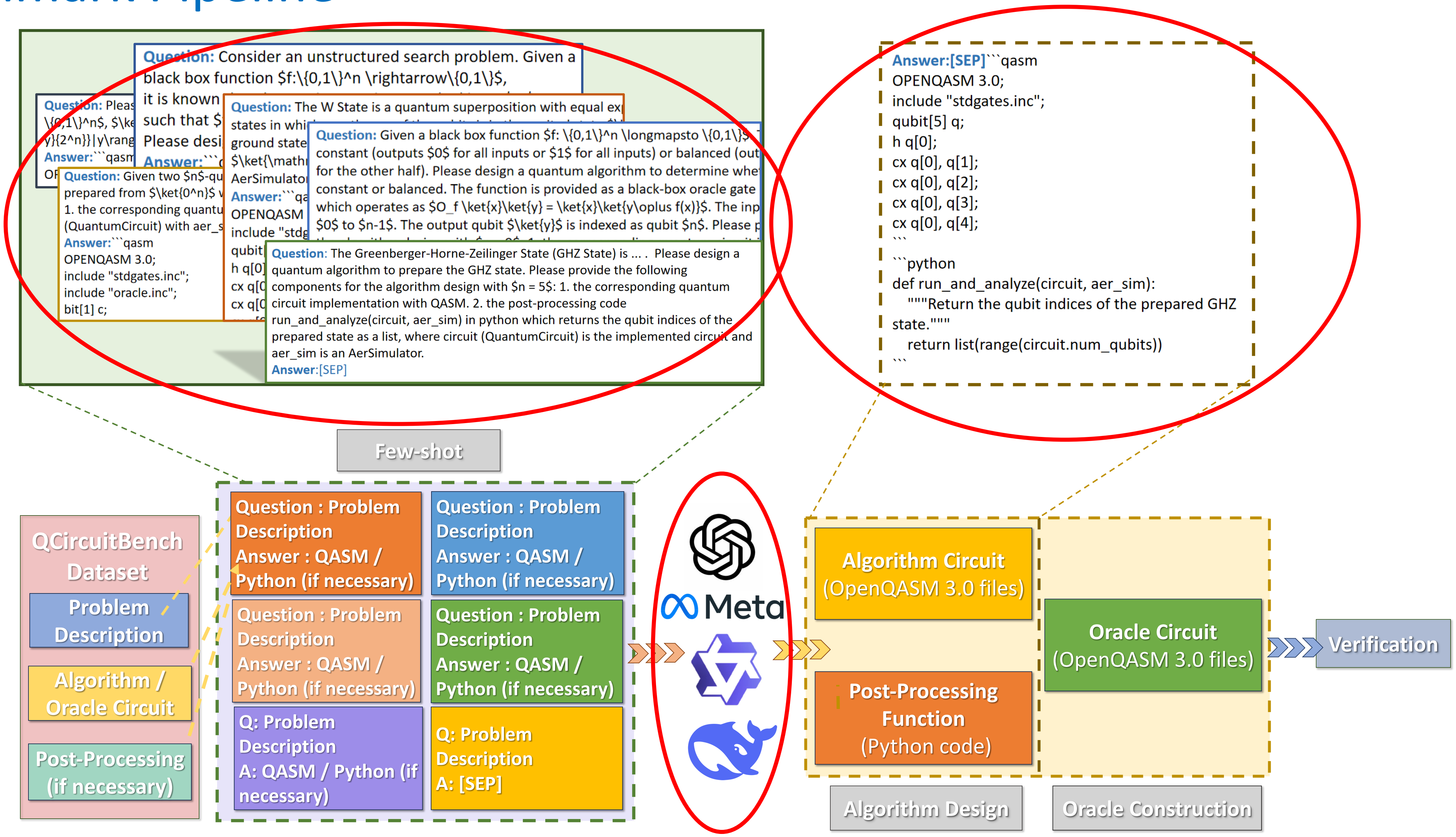
Quantum Algorithms

- **Textbook-Level Algorithms**: Bernstein-Vazirani problem, Deutsch-Jozsa problem, Simon's problem, Grover's algorithm, phase estimation, quantum Fourier transform, Shor's algorithm, etc.
- **Generalized Simon's Problem**: Intuitively, it extends Simon's Problem from binary to p-ary bases and from a single secret string to a subgroup of rank k.
- **Quantum Information Protocols**: GHZ state preparation, W state preparation, swap test, quantum teleportation, superdense coding, quantum key distribution, etc.
- **Variational Quantum Algorithms**: VQE for ground-state energy estimation, QAOA for combinatorial optimization, etc.

Contents

QCircuitBench

Benchmark Pipeline



BLEU Score

- Measures similarity between model-generated output and reference code.

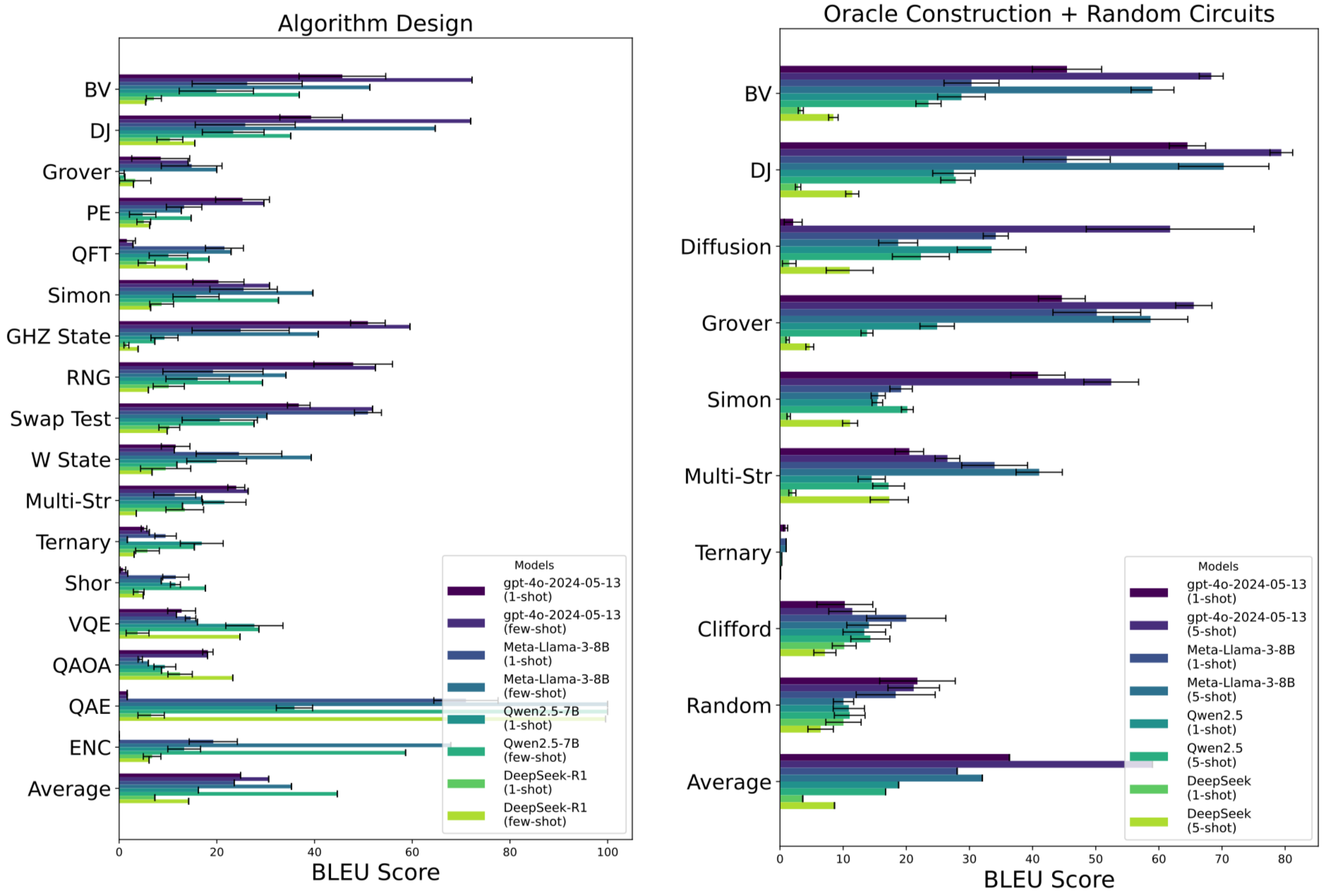


Figure 3: Benchmarking algorithm design and oracle construction tasks in BLEU scores.

Verification Score

Table 1: QASM syntax score for benchmarking quantum algorithm design.

Model	Shot	Bernstein Vazirani	Deutsch Jozsa	Grover	Phase Estimation	QFT	Simon	GHZ	Random Number Generator	Swap Test	W State	Generalized Simon (multi-str)	Generalized Simon (ternary)	Shor	VQE	QAOA	QAE	ENC	Avg
GPT-4o	1	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	1.0000 (±0.0000)	1.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.2308 (±0.0843)	1.0000 (±0.0000)	0.8333 (±0.0904)	0.5833 (±0.1486)	0.2734
GPT-4o	5	1.0000 (±0.0000)	1.0000 (±0.0000)	0.0000 (±0.0000)	0.6154 (±0.1404)	0.5385 (±0.1439)	0.9231 (±0.0769)	0.5714 (±0.2020)	1.0000 (±0.0000)	1.0000 (±0.0000)	0.4444 (±0.1757)	0.0769 (±0.0769)	0.1111 (±0.1111)	0.0000 (±0.0000)	0.2308 (±0.0843)	0.7222 (±0.1086)	1.0000 (±0.0000)	0.5833 (±0.1486)	0.5775
Llama3	1	0.1538 (±0.1042)	0.2308 (±0.1216)	0.3077 (±0.1332)	0.4615 (±0.1439)	0.0000 (±0.0000)	0.1538 (±0.1042)	0.1429 (±0.1429)	0.4615 (±0.1439)	0.1429 (±0.0971)	0.3333 (±0.1667)	0.5385 (±0.1439)	0.4444 (±0.1757)	0.0000 (±0.0000)	0.2574 (±0.0285)	0.1667 (±0.0544)	0.0000 (±0.0000)	0.3438 (±0.0853)	0.2435
Llama3	5	0.5385 (±0.1439)	0.3846 (±0.1404)	0.6154 (±0.1404)	0.5385 (±0.1439)	0.3846 (±0.1404)	0.1538 (±0.1042)	0.2857 (±0.1844)	0.9231 (±0.0769)	0.5000 (±0.1387)	0.3333 (±0.1667)	0.8462 (±0.1042)	0.3333 (±0.1667)	0.0000 (±0.0000)	0.2363 (±0.0277)	0.9375 (±0.0353)	0.0000 (±0.0000)	0.8125 (±0.0701)	0.4602
Qwen 2.5	1	0.0769 (±0.0769)	0.1538 (±0.1042)	0.0000 (±0.0000)	0.0769 (±0.0769)	0.0769 (±0.0769)	0.3077 (±0.1332)	0.4286 (±0.2020)	0.2308 (±0.1216)	0.2857 (±0.1253)	0.2222 (±0.1470)	0.5385 (±0.1439)	0.1111 (±0.1111)	0.0000 (±0.0000)	0.4515 (±0.0324)	0.8750 (±0.0482)	0.0000 (±0.0000)	1.0000 (±0.0000)	0.2844
Qwen 2.5	5	0.3077 (±0.1332)	0.6154 (±0.1404)	0.1538 (±0.1042)	0.3077 (±0.1332)	0.2308 (±0.1216)	0.1538 (±0.1042)	0.4286 (±0.2020)	0.6154 (±0.1404)	0.5714 (±0.1373)	0.2222 (±0.1470)	0.4615 (±0.1439)	0.2222 (±0.1470)	0.0000 (±0.0000)	0.3544 (±0.0311)	0.9583 (±0.0291)	1.0000 (±0.0000)	0.7188 (±0.0808)	0.4307
DeepSeek-R1	1	0.0000 (±0.0000)	0.0769 (±0.0769)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.1429 (±0.1429)	0.0769 (±0.0769)	0.0714 (±0.0714)	0.0000 (±0.0000)	0.1538 (±0.1042)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.07173 (±0.0168)	0.2292 (±0.0613)	0.0000 (±0.0000)	0.1563 (±0.0652)	0.0576
DeepSeek-R1	5	0.3846 (±0.1404)	0.0769 (±0.0769)	0.0000 (±0.0000)	0.0769 (±0.0769)	0.0769 (±0.0769)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.1538 (±0.1042)	0.1429 (±0.0971)	0.0000 (±0.0000)	0.2308 (±0.1216)	0.0000 (±0.0000)	0.0000 (±0.0000)	0.0084 (±0.0060)	0.4167 (±0.0719)	1.0000 (±0.0000)	0.4375 (±0.0891)	0.1768
Human	-	0.5000	1.0000	0.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	1.0000	1.0000	0.0000	0.5000	1.0000	1.0000	1.0000	0.6667	0.6862

QASM Syntax Check

Is the QASM code syntactically valid?

Python Syntax Check

Is the post-processing script valid?

Semantic Accuracy

Does simulation output match the expected result?

Observations

- **Few-shot > One-shot** in most cases *(e.g., Qwen 2.5 improves by 0.2854 on Deutsch-Jozsa)*.
- **Best Performers:** GPT-4o excels at in-context learning and long-code understanding.
- **Harder tasks** *(QFT, Shor's algorithm)* remain challenging even for GPT-4o w. few-shot.
- **VQE and QAOA** are especially difficult due to *hybrid classical-quantum structure* *(e.g. DeepSeek-R1 achieves all zero scores on semantic score of VQE and QAOA)*.
- **BLEU vs. Verification** sometimes diverge *(e.g., high BLEU but poor semantic in GHZ state)*.
- **Open-book setting** shows unguided retrieval may introduce noise or distract from task-specific structure.

Error Types

Improvisation Error

- GPT-4o uses unsupported QASM 3.0 features *(e.g., custom namespaces)*, causing failures.
- Better adherence in few-shot settings.

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

GPT-4o uses the "for" loop syntax in OpenQASM 3.0. Qiskit.qasm3 import module does not support this advanced feature, resulting failure in syntax validation.

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

GPT-4o assigns novel names to the qubit registers, leading to a conflict in the symbol table. Substitution with new names delivers the correct solution.

Error Types

Counting Error

- GPT-4o fails to identify ‘1’ bits in the secret string *(e.g., for Bernstein-Vazirani)*.
- Suggests difficulty in binary indexing and tokenization.

GPT4o’s answer for counting indices

In the secret string s="000101", the bits that equal 1 are found at the following indices (0-based):
Index 2
Index 5
These indices correspond to the positions where the string has a ’1’.

```
OPENQASM 3.0;
include "stdgates.inc";

gate Oracle _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3, _gate_q_4, _gate_q_5, _gate_q_6 {
  // Apply CX gates according to the secret string "000101"
  cx _gate_q_2, _gate_q_6;
  cx _gate_q_5, _gate_q_6;
}
```

Error Types

Data Contamination

- LLMs recall Qiskit tutorials well but struggle with gate-level QASM synthesis from scratch.
- QCircuitBench helps mitigate this with fresh, custom-built QASM data.

Text Answer by GPT4o

Here's a Python code using Qiskit to implement the Bernstein-Vazirani algorithm with $n = 5$ qubits. The secret string used in this example is $s = 10101$ (you can change this to any string of your choice).

```
# Create a Quantum Circuit with n+1 qubits (one ancillary qubit)
bv_circuit = QuantumCircuit(n+1, n)

# Apply Hadamard gates to all qubits except the ancillary qubit
bv_circuit.h(range(n))

# Apply X and Hadamard gate to the ancillary qubit
bv_circuit.x(n)
bv_circuit.h(n)

# Construct the Oracle
for i, bit in enumerate(s):
    if bit == '1':
        bv_circuit.cx(i, n)

# Apply Hadamard gates to the first n qubits
bv_circuit.h(range(n))
```

```
h q[2];
h q[3];
h q[4];

// Apply Hadamard to the ancillary qubit and set it to |1>
x q[5];
h q[5];

// Step 2: Oracle implementation (CNOT gates based on secret string s = 10110)
cx q[0], q[5]; // Secret string bit 1
// No cx for q[1] because the secret string bit is 0
cx q[2], q[5]; // Secret string bit 1
cx q[3], q[5]; // Secret string bit 1
// No cx for q[4] because the secret string bit is 0

// Step 3: Apply Hadamard gates to the input qubits again
h q[0];
h q[1];
h q[2];
h q[3];
```

Fine-tuning Results

- LoRA-based fine-tuning on LLaMA3-8B *(8-bit quantized)*.
- Improves scores, especially better at counting ‘1’ bits *(Bernstein-Vazirani)*.
- Scores drop on random circuits, indicating challenge of encoding quantum state vectors within a language model and overfitting on tasks with high output diversity.

Table 2: Fine-tuning oracle construction scores.

Score	Model	Setting	Bernstein-Vazirani	Deutsch-Jozsa	Grover	Simon	Clifford	Universal	Avg
BLEU	gpt4o	few-shot(5)	95.6388 (±0.3062)	91.0564 (±0.6650)	92.0620 (±0.6288)	80.3390 (±2.0900)	39.5469 (±3.6983)	33.3673 (±3.1007)	72.0017
	Llama3	few-shot(5)	53.5574 (±5.2499)	69.8996 (±5.7812)	61.3102 (±5.4671)	26.3083 (±2.0048)	13.0729 (±0.9907)	13.4185 (±1.2299)	39.5945
	Llama3	finetune	76.0480 (±7.9255)	71.8378 (±2.4179)	67.7892 (±7.8900)	43.8469 (±3.2998)	10.8978 (±0.6169)	7.1854 (±0.5009)	46.2675
Verification	gpt4o	few-shot(5)	0.0000 (±0.0246)	0.4300 (±0.0590)	0.0000 (±0.1005)	-0.0200 (±0.0141)	-0.0333 (±0.0401)	-0.1023 (±0.0443)	0.0457
	Llama3	few-shot(5)	-0.2700 (±0.0468)	0.0900 (±0.0668)	-0.5200 (±0.0858)	-0.6600 (±0.0476)	-0.7303 (±0.0473)	-0.5056 (±0.0549)	-0.4327
	Llama3	finetune	-0.1300 (±0.0485)	-0.2000 (±0.0402)	-0.3300 (±0.0900)	-0.7400 (±0.0441)	-0.8741 (±0.0343)	-0.9342 (±0.0262)	-0.5347
PPL	Llama3	few-shot(5)	1.1967 (±0.0028)	1.1174 (±0.0015)	1.1527 (±0.0021)	1.1119 (±0.0017)	1.4486 (±0.0054)	1.4975 (±0.0051)	1.2541
	Llama3	finetune	1.0004 (±0.0002)	1.1090 (±0.0014)	1.0010 (±0.0006)	1.1072 (±0.0011)	1.2944 (±0.0053)	1.3299 (±0.0055)	1.1403

Contents

QCircuitBench

Takeaways

- **Novelty**
  - First large-scale benchmark for LLM-driven quantum algorithm design.
- **Dataset Design**
  - A perspective from code generation.
  - Modular and extensible structure.
  - Automatic verification functions.
- **Experiments**
  - QCircuitBench poses significant challenges to SOTA LLMs.
  - Fine-tuning experiments demonstrate early promise.

Open Challenges

- **Data Bottleneck**
  - Few existing quantum algorithms → **limited dataset diversity**

How can we construct large-scale, high-quality datasets for LLMs in quantum algorithm design?

- **Fine-tuning for Design**
  - Move from **benchmarking** to enabling **new quantum algorithm synthesis**

Which fine-tuning methods are best for quantum data? What metrics best reflect model capability?

- **Evaluation Bottlenecks**
  - Classical simulation of quantum circuits is computationally expensive

How to develop efficient, scalable automatic evaluation suitable for long/deep circuits?

Thanks!