Shor 与 Grover 算法的物理原理

[redacted]
钱学森学院 少年班 240[redacted]
西安交通大学
[redacted]
or [redacted]

摘要

Shor 算法和 Grover 算法分别代表了量子算法中两种不同的加速机制。前者把整数分解转化为模指数函数的周期测量，量子相位估计与逆量子 Fourier 变换把周期信息压缩到离散相位峰；后者在由目标态和非目标态叠加态张成的二维子空间内反复旋转振幅，使目标态测量概率接近 1。本文先给出两个算法的数学步骤，再从相干叠加、相位反冲、Fourier 干涉和反射合成的角度解释其物理图像。文末结合一个 Qiskit 小项目：Shor 示例取 $N = 15, a = 2$，8 个计数比特的测量峰给出相位 $1/4$ 并恢复阶 $r = 4$，得到因子 $(3, 5)$；Grover 示例在 3 比特空间中搜索 $|101\rangle$，两次迭代后目标态精确概率为 0.9453125，4096 次抽样中出现 3864 次。

1 引言

量子计算的加速并不是把所有候选答案同时读出。一次测量只能给出一个经典结果，叠加态本身也不能绕开这一限制。Shor 算法和 Grover 算法的共同点在于它们先把问题写成酉演化，再用干涉把可测概率重新分配到少数结果上。不同之处在于，Shor 算法利用函数周期带来的相位结构，Grover 算法利用两次反射在二维空间内形成的振幅旋转。

Shor 在 1994 年给出了离散对数和整数分解的量子算法 [4]，其扩展版本后来发表于 *SIAM Journal on Computing*[5]。算法的量子部分并不直接输出因子，而是求一个模乘算符的阶。Grover 在 1996 年提出无结构数据库搜索算法 [1]，随后用“inversion about average”的语言给出了更物理的表述 [2]。若搜索空间大小为 M，一个目标项的经典随机搜索平均需要 $M/2$ 次查询，而 Grover 迭代需要约 $\pi\sqrt{M}/4$ 次 oracle 查询。

本文的目标不是复述算法步骤，而是把“为什么这些门会把正确答案变成高概率事件”说清楚。为此，正文把每个算法分成三层：算法流程、物理图像、Qiskit 仿真。Dirac 记号、量子 Fourier 变换和相位估计的基本约定采用 Nielsen 与 Chuang 教材中的写法 [3]。数值实验基于 Qiskit 2.1 以上版本搭建；Shor 部分采用教学规模的模乘置换矩阵，适合解释 $N = 15$ 一类小例子，不用于大整数分解。

2 Shor 算法：从因数分解到阶寻找

设 N 为待分解的奇合数。随机选取 $1 < a < N$，若 $\gcd(a, N) > 1$，经典 Euclid 算法已经给出一个非平凡因子。困难情形是 $\gcd(a, N) = 1$。此时考虑 a 模 N 的阶 r：

$$a^r \equiv 1 \pmod{N}, \qquad r = \min\{t > 0 : a^t \equiv 1 \pmod{N}\}. \tag{1}$$

如果 r 为偶数，且 $a^{r/2} \not\equiv -1 \pmod{N}$，则

$$a^r - 1 = (a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \pmod{N}. \tag{2}$$

因此

$$p = \gcd(a^{r/2} - 1, N), \qquad q = \gcd(a^{r/2} + 1, N) \tag{3}$$

通常给出 N 的非平凡因子。量子电路只负责估计 r；随机选取 a、检查互素性、连分数恢复和 gcd 计算仍是经典步骤。图1把这个分工分成两条路径：若 $\gcd(a, N)$ 已经暴露因子，算法不进入量子电路；若 a 与 N 互素，才把阶寻找交给相位估计。

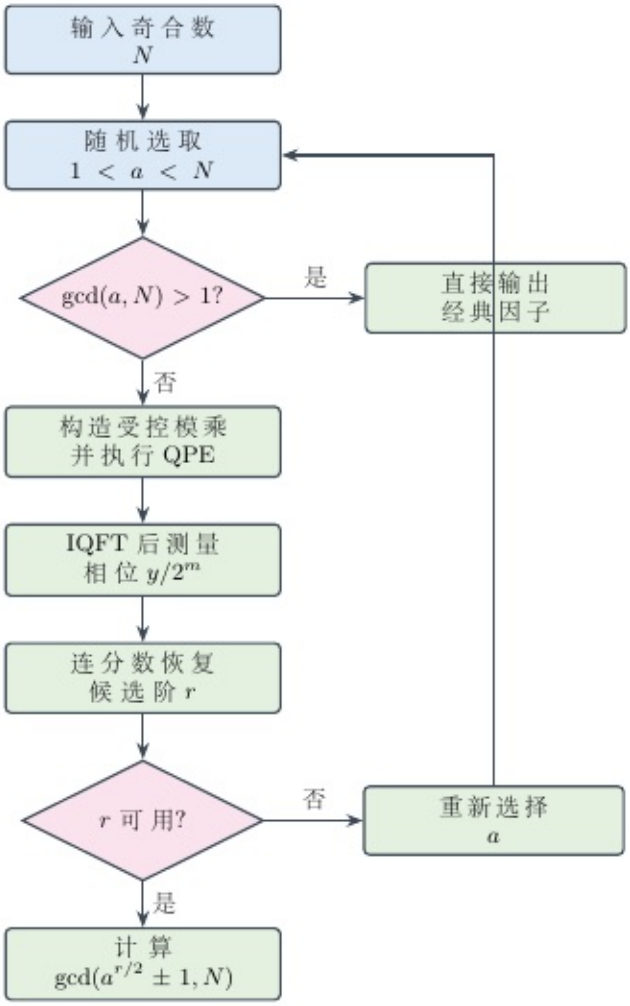

图 1: Shor 算法流程

对 $N = 15, a = 2$，阶为 4，因为 $2^0, 2^1, 2^2, 2^3$ 模 15 分别为 $1, 2, 4, 8$，而 $2^4 \equiv 1 \pmod{15}$。此时 $a^{r/2} = 4$，于是

$$\gcd(4 - 1, 15) = 3, \qquad \gcd(4 + 1, 15) = 5. \tag{4}$$

这个例子足够小，可以在经典计算机上把模乘写成完整酉矩阵；它也足够完整，能展示 Shor 算法中“周期转频谱”的核心结构。

3 Shor 算法的物理图像

Shor 电路的核心是模乘酉算符

$$U_a\,|x\rangle = |ax \bmod N\rangle\,. \tag{5}$$

当 $\gcd(a,N)=1$ 时，U_a 在 $\{0,\dots,N-1\}$ 上是置换，因此可嵌入到量子电路的酉演化中。相位估计并不测量 U_a 怎样作用于每个 x，而是测量其本征相位。若 r 为 a 的阶，则周期轨道上的本征态可写成

$$|u_s\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1}\exp\left(-\frac{2\pi i s k}{r}\right)|a^k \bmod N\rangle\,, \qquad s=0,\dots,r-1, \tag{6}$$

并满足

$$U_a\,|u_s\rangle = \exp\left(\frac{2\pi i s}{r}\right)|u_s\rangle\,. \tag{7}$$

相位估计把本征值中的 s/r 写入计数寄存器。物理上看，受控 $U_a^{2^j}$ 让计数比特的相位随 j 倍增；逆 QFT 把这种相位斜率变成计算基上的峰。图2只保留寄存器级结构：Hadamard 门制备计数叠加，受控模指数写入周期相位，逆 QFT 把相位转成可测比特串。

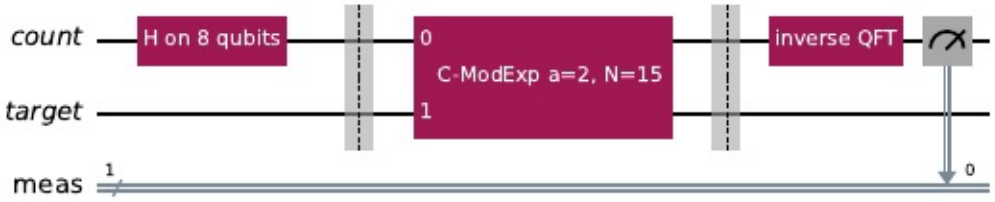

图 2: 阶寻找结构

这个过程可与一维光栅类比。周期函数 $f(x)=a^x \bmod N$ 把不同指数分成若干同余类；同一周期内的振幅在 QFT 后同相叠加，非匹配频率处的振幅相互抵消。图3用 $N=15, a=2$ 展示这一点。左图的残数序列每 4 步重复；右图中 8 个计数比特对应 $2^8=256$ 个离散频率，理想峰落在 $y=0,64,128,192$，即相位 $0, 1/4, 1/2, 3/4$。

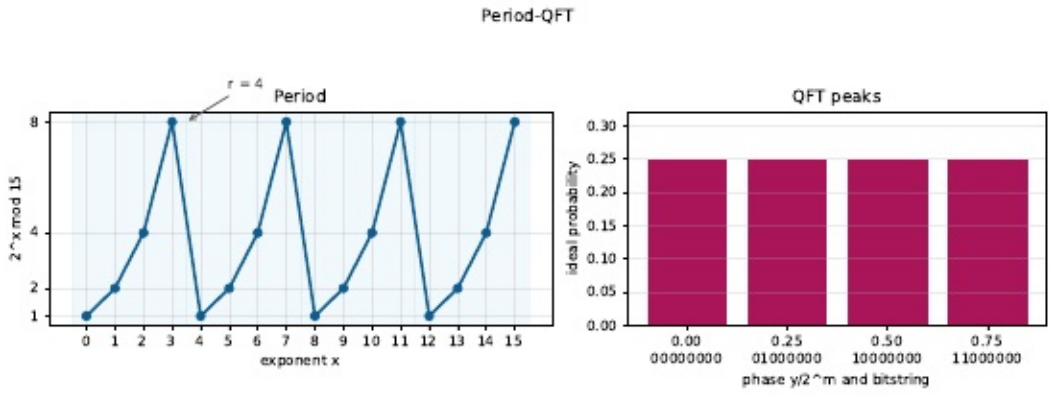

图 3: 周期频谱

如果计数寄存器比特数 m 有限，测量值 y 只能给出 s/r 的近似值 $y/2^m$。Shor 算法之后使用连分数，把 $y/2^m$ 近似为分母不超过 N 的有理数。项目默认例子中 $m=8$，测到 $y=64$ 时有 $64/256=1/4$，连分数不需要进一步近似即可读出 $r=4$。

4 Shor 算法的 Qiskit 仿真

Shor 仿真基于 Qiskit 电路模型搭建阶寻找实验。实验对象不是通用大整数分解器，而是面向 $N=15, a=2$ 的可解释量子电路：8 个计数比特用于记录相位，4 个目标比特用于编码 0 到 14 之间的模乘轨道。电路由三类模块构成：计数寄存器的均匀叠加、受控模乘置换 $M_{a^{2^k}} \bmod N$、计数寄存器上的逆 QFT。目标寄存器用 X 门制备为 $|1\rangle$，计数寄存器经 Hadamard 门制备为

$$\frac{1}{2^{m/2}} \sum_{x=0}^{2^m-1} |x\rangle \,. \tag{8}$$

随后电路施加受控模乘 $M_2 \bmod 15$ 与 $M_4 \bmod 15$；当乘子为 1 时对应模块退化为恒等演化。对 $N=15$，模乘可写成 16 维置换酉矩阵，这一构造用于展示相位估计机制，而不代表大规模 Shor 算法的算术电路成本。图4是论文版紧凑电路，图中保留寄存器规模和关键模块，但不展开每个受控模乘矩阵，也不展开逆 QFT 内部的受控相位门。

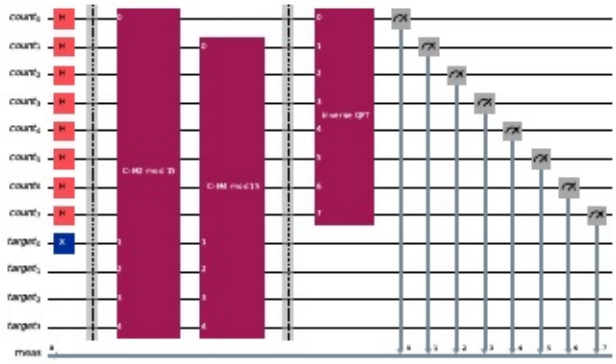

图 4: Shor 电路

仿真采用 `Statevector` 得到精确分布，再用固定随机种子进行 4096 次抽样。图5中四个比特串为 `00000000`、`01000000`、`10000000`、`11000000`，对应 $y=0,64,128,192$。抽样计数为 999、1013、1083、1001，围绕理想概率 1/4 波动。后处理跳过 $y=0$，从 `01000000` 得到相位 1/4，恢复阶 $r=4$，再给出因子 $(3,5)$。

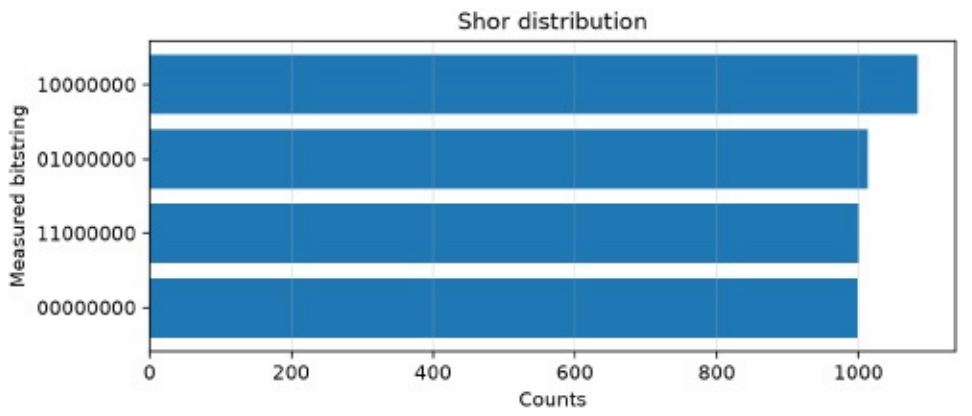

图 5: Shor 仿真

这个仿真也说明了教学实现的边界。项目用 `UnitaryGate` 直接构造小规模模乘置换矩阵，矩阵维度随目标比特数指数增长。真正的大整数分解需要可逆加法、模乘和模指数算术电路；本文只用 $N = 15$ 讨论物理机制。

5 Grover 算法：无结构搜索的迭代

Grover 算法处理的问题是：给定 $M = 2^n$ 个计算基态和一个判定函数 $f(x)$，找出满足 $f(w) = 1$ 的目标态 $|w\rangle$。oracle 不返回 w，而是实现相位标记

$$O_w |x\rangle = (-1)^{f(x)} |x\rangle . \tag{9}$$

算法从均匀叠加态出发：

$$|s\rangle = \frac{1}{\sqrt{M}} \sum_{x=0}^{M-1} |x\rangle . \tag{10}$$

一次 Grover 迭代为

$$G = DO_w, \qquad D = 2 |s\rangle \langle s| - I. \tag{11}$$

其中 D 是关于 $|s\rangle$ 方向的反射，也可理解为对平均振幅的反演 [2]。图6强调一次 Grover 迭代的两步：oracle 先给目标态加负相位，diffusion 再把相位差转化为目标态振幅的增加。

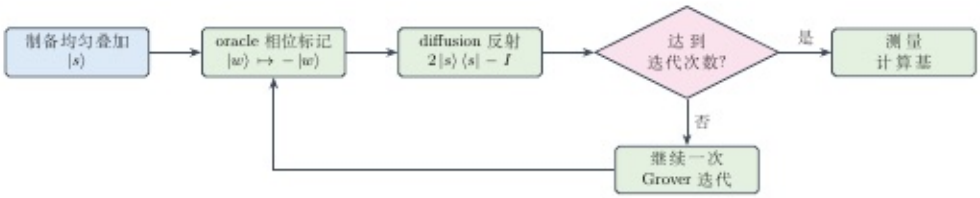

图 6: Grover 流程

若目标态数量为 K，令 $\sin\theta = \sqrt{K/M}$。最常用的迭代次数为

$$T = \left\lfloor \frac{\pi}{4\theta} \right\rfloor . \tag{12}$$

本项目默认 $M = 8, K = 1$，故 $\theta = \arcsin\left(1/\sqrt{8}\right)$，最优整数迭代数为 2。

6 Grover 算法的物理图像

Grover 算法的整个演化可以投影到二维实平面。定义目标子空间方向为 $|w\rangle$，非目标态的归一化叠加为

$$|w_\perp\rangle = \frac{1}{\sqrt{M-1}} \sum_{x \neq w} |x\rangle \,. \tag{13}$$

初态为

$$|s\rangle = \sin\theta\,|w\rangle + \cos\theta\,|w_\perp\rangle \,. \tag{14}$$

oracle 把 $|w\rangle$ 分量变号，相当于关于 $|w_\perp\rangle$ 轴反射；diffusion 关于 $|s\rangle$ 轴反射。两次反射的合成是旋转，每次 Grover 迭代使态矢量向 $|w\rangle$ 方向旋转 2θ。迭代 j 次后目标态振幅满足

$$\big\langle w|G^j s\big|w|G^j s\big\rangle = \sin\big((2j+1)\theta\big), \qquad P_w(j) = \sin^2\big((2j+1)\theta\big). \tag{15}$$

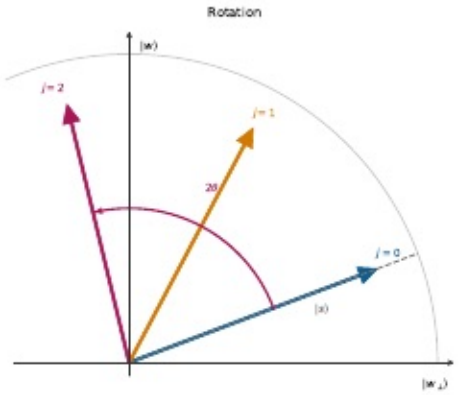

图 7: 振幅旋转

图7画出 $j = 0, 1, 2$ 三根态矢量。第二次迭代后态矢量已经接近 $|w\rangle$ 轴，因此测量目标态的概率接近 1。对 3 比特单目标搜索，$P_w(0) = 1/8$。一次迭代后 $P_w(1) = 25/32 = 0.78125$，两次迭代后 $P_w(2) = 121/128 = 0.9453125$。继续迭代不会单调提高概率，因为旋转会越过目标方向；这也是 Grover 算法必须控制迭代次数的原因。图8给出项目参数下的精确概率曲线。

图 8: 目标态概率

7 Grover 算法的 Qiskit 仿真

Grover 仿真基于 Qiskit 搭建 3 比特相位标记搜索电路，用于验证“相位标记加 diffusion 反射”如何把概率集中到目标态。目标态设为 101。oracle 模块先把需要按 0 控制的量子比特翻转到 1 控制形式，再通过多控 Z 等效结构只改变目标态的相位，最后恢复原来的计算基标号。diffusion 模块实现 $D = 2|s\rangle\langle s| - I$，在门级上对应 Hadamard、$X$、多控相位翻转以及逆序恢复。完整实验先对 3 个量子比特施加 Hadamard 门，然后串接两次 Grover 迭代，最后测量全部量子比特。图9中两个紫色模块对应这两次 Grover 迭代。

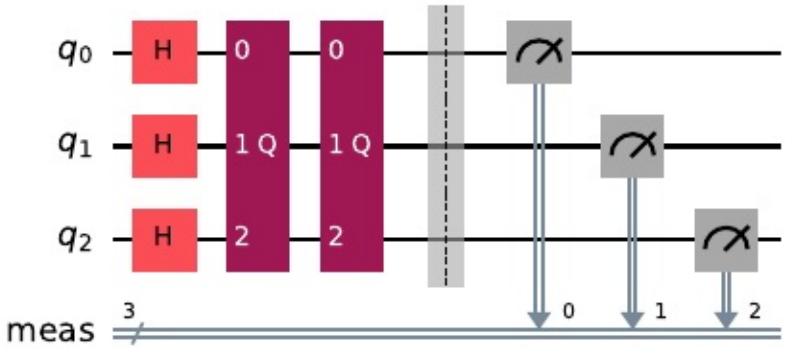

图 9: Grover 电路

精确态矢量仿真给出

$$P(101) = 0.9453125, \tag{16}$$

其余 7 个状态的概率各约为 0.0078125。固定随机种子下 4096 次抽样得到 101 共 3864 次；第二高的 000 只有 42 次。图10显示了这一概率集中现象。与 Shor 的四个相位峰不同，Grover 搜索的目标是把概率尽量集中到单个标记态上。

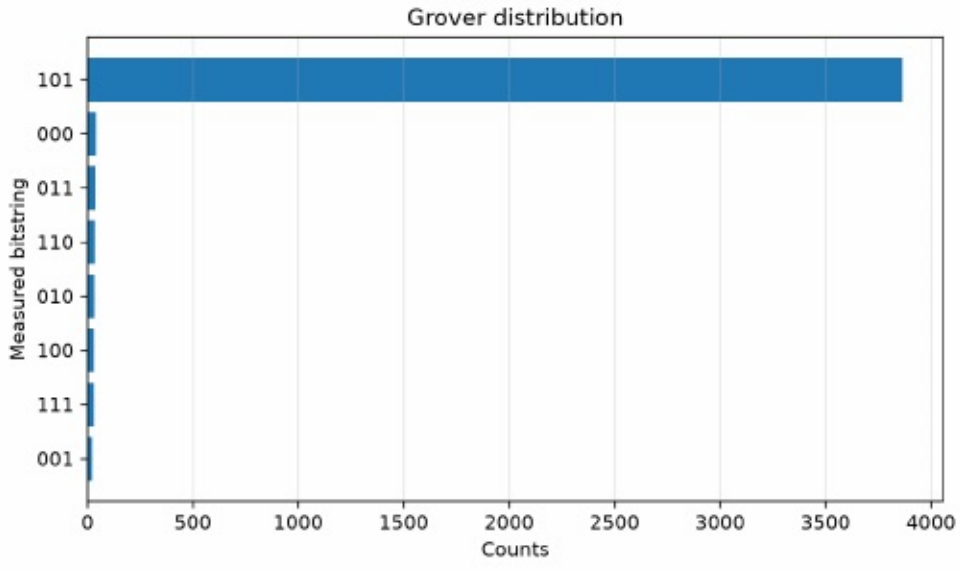

图 10: Grover 仿真

8 总结

Shor 算法和 Grover 算法使用了两种不同的干涉结构。Shor 算法把数论周期写入受控模乘算符的本征相位，逆 QFT 把周期转化为计数寄存器上的频谱峰；后处理再用连分数和 gcd 把阶变成因子。项目中的 $N = 15, a = 2$ 例子测得相位 1/4，恢复 $r = 4$，给出 $(3, 5)$。

Grover 算法没有函数周期可用。它只假设 oracle 能识别目标态，然后在 $|w\rangle$ 与 $|w_\perp\rangle$ 张成的二维子空间内做反射。两次反射合成为固定角度旋转，迭代次数选择为 2 时，3 比特目标 `101` 的概率达到 121/128。从物理角度看，Shor 算法的关键词是相位估计与 Fourier 干涉，Grover 算法的关键词是相位标记与振幅旋转。两者都不把“并行尝试”直接变成答案，而是把相干演化设计成可测概率分布。

参考文献

[1] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 212–219. ACM Press, 1996. doi: 10.1145/237814.237866.

[2] Lov K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physical Review Letters*, 79(2):325–328, 1997. doi: 10.1103/PhysRevLett.79.325.

[3] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 10th anniversary edition, 2010.

[4] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Computer Society Press, 1994. doi: 10.1109/SFCS.1994.365700.

[5] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. doi: 10.1137/S0097539795293172.

附录

A 核心代码片段

下面列出 Grover 与 Shor 实验中最能体现技术逻辑的核心片段，分别对应相位 oracle、Grover 迭代电路、模乘酉门、阶寻找电路和经典后处理。

Grover 相位 oracle

```python
def create_phase_oracle(marked_states: str | Iterable[str]) -> QuantumCircuit:
    states = normalize_marked_states(marked_states)
    num_qubits = len(states[0])
    oracle = QuantumCircuit(num_qubits, name="Oracle")

    for state in states:
        little_endian_state = state[::-1]
        zero_indices = [
            i for i, bit in enumerate(little_endian_state) if bit == "0"
        ]

        oracle.x(zero_indices)
        if num_qubits == 1:
            oracle.z(0)
        else:
            controls = list(range(num_qubits - 1))
```

```
            target = num_qubits - 1
            oracle.h(target)
            oracle.mcx(controls, target)
            oracle.h(target)
        oracle.x(zero_indices)

    return oracle
```

Grover 完整电路

```
def build_grover_circuit(
    marked_states: str | Iterable[str],
    iterations: int | None = None,
    *,
    with_measurements: bool = True,
) -> QuantumCircuit:
    states = normalize_marked_states(marked_states)
    num_qubits = len(states[0])
    oracle = create_phase_oracle(states)
    grover_op = grover_operator(oracle)

    if iterations is None:
        iterations = optimal_iterations(num_qubits, len(states))

    qc = QuantumCircuit(num_qubits, name="Grover")
    qc.h(range(num_qubits))
    qc.compose(grover_op.power(iterations), inplace=True)

    if with_measurements:
        qc.measure_all()
    return qc
```

Shor 模乘酉门

```
def modular_multiplication_gate(multiplier: int, modulus: int) -> UnitaryGate:
    if gcd(multiplier, modulus) != 1:
        raise ValueError("multiplier and modulus must be coprime.")

    num_target_qubits = ceil(log2(modulus))
    dimension = 2**num_target_qubits
    unitary = np.zeros((dimension, dimension), dtype=complex)

    for x in range(dimension):
        if x < modulus:
            y = (multiplier * x) % modulus
        else:
            y = x
        unitary[y, x] = 1.0

    gate = UnitaryGate(unitary)
    gate.name = f"M_{multiplier}_mod_{modulus}"
    return gate
```

Shor 阶寻找电路

```
def build_order_finding_circuit(
    modulus: int,
    base: int,
    *,
    counting_qubits: int | None = None,
    with_measurements: bool = True,
) -> QuantumCircuit:
    num_target_qubits = ceil(log2(modulus))
    if counting_qubits is None:
        counting_qubits = 2 * num_target_qubits

```

```
    count = QuantumRegister(counting_qubits, name="count")
    target = QuantumRegister(num_target_qubits, name="target")
    classical = ClassicalRegister(counting_qubits, name="meas")
    qc = QuantumCircuit(count, target, classical, name="ShorOrderFinding")

    qc.x(target[0])
    qc.h(count)

    for k in range(counting_qubits):
        multiplier = pow(base, 2**k, modulus)
        if multiplier == 1:
            continue
        gate = modular_multiplication_gate(multiplier, modulus).control(1)
        qc.compose(gate, qubits=[count[k], *target], inplace=True)

    inverse_qft = synth_qft_full(
        counting_qubits,
        do_swaps=True,
        approximation_degree=0,
        insert_barriers=True,
        inverse=True,
        name="IQFT",
    )
    qc.compose(inverse_qft, qubits=count, inplace=True)

    if with_measurements:
        qc.measure(count, classical)
    return qc
```

Shor 经典后处理

```
def try_recover_factors(
    counts: dict[str, int],
    *,
    modulus: int,
    base: int,
    counting_qubits: int,
) -> dict[str, object] | None:
    for bitstring, shots in sorted(counts.items(), key=lambda item: item[1], reverse=True):
        y = int(bitstring, 2)
        if y == 0:
            continue

        phase = y / (2**counting_qubits)
        rational = Fraction(phase).limit_denominator(modulus)
        order = rational.denominator

        if order <= 0 or pow(base, order, modulus) != 1:
            continue
        if order % 2 != 0:
            continue

        x = pow(base, order // 2, modulus)
        if x == modulus - 1:
            continue

        factor1 = gcd(x - 1, modulus)
        factor2 = gcd(x + 1, modulus)
        if 1 < factor1 < modulus and 1 < factor2 < modulus:
            return {
                "bitstring": bitstring,
                "shots": shots,
                "phase": phase,
                "phase_fraction": str(rational),
                "order": order,
                "factors": (factor1, factor2),
            }

    return None
```

B 完整的 Shor 电路结构展示

图11给出展开后的阶寻找电路。正文采用紧凑图，是因为展开图主要用于检查寄存器连接、受控模乘模块和逆 QFT 位置。

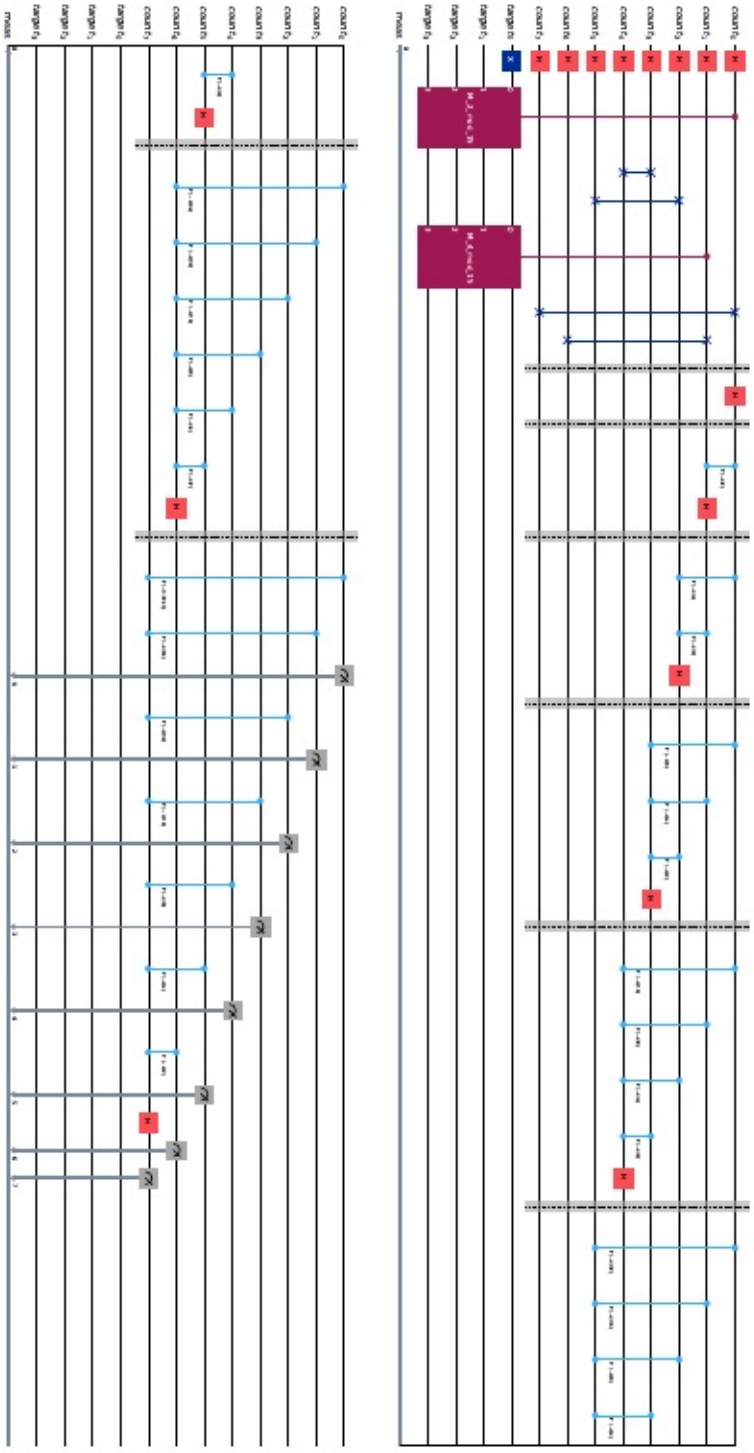

图 11: Shor 全电路