UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Felipe Augusto Nunes Cintra

Implementação do Algoritmo de Shor como Objeto de Comparação das Tecnologias Quânticas Cirq, Qiskit e PyQuil

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Felipe Augusto Nunes Cintra

Implementação do Algoritmo de Shor como Objeto de Comparação das Tecnologias Quânticas Cirq, Qiskit e PyQuil

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Luís Fernando Faina

Coorientador: Prof. Dr. João Henrique de Souza Pereira

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2025

Felipe Augusto Nunes Cintra

Implementação do Algoritmo de Shor como Objeto de Comparação das Tecnologias Quânticas Cirq, Qiskit e PyQuil

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Prof. Dr. Luís Fernando Faina
Orientador

Prof. Dr. João Henrique de Souza Pereira
Coorientador

Prof. Dr. Ivan da Silva Sendin

Profª. Dra. Maria Adriana Vidigal de Lima

Uberlândia, Brasil
2025

Agradecimentos

Agradeço a Deus por tão grande amor, por ter me sustentado até aqui. Agradeço aos meus pais, que com carinho me conduziram, apoiaram e incentivaram, me permitindo viver tudo isso. Agradeço a minha companheira, que acompanhou-me ao longo de toda esta jornada, dando grande apoio durante os desafios. Agradeço aos meus amigos, que juntos na sala de aula, e mesmo fora dela, tornaram essa trajetória única e suave. Por fim, agradeço aos meus orientadores, que com paciência ouviram, acompanharam, suportaram e tornaram possível a realização deste trabalho.

"Those who can imagine anything, can create the impossible."

Alan Turing

Resumo

Este trabalho apresenta uma análise comparativa de tecnologias de programação quântica tendo por base a implementação do algoritmo de Shor. Foram avaliadas as ferramentas *Cirq*, desenvolvida pelo Google, o *Qiskit*, da IBM, e *PyQuil*, mantido pela Rigetti Computing. A implementação em *Cirq* exigiu a criação manual de portas aritméticas e circuitos personalizados, evidenciando sua flexibilidade, mas também o esforço extra necessário. Já no *Qiskit*, notou-se um maior grau de abstração, com recursos de alto nível para exponenciação modular e Transformada Quântica de Fourier, além de documentação extensa e integração nativa com simuladores e processadores quânticos reais da IBM. Por outro lado, o *PyQuil* se mostrou restrito para este trabalho, em razão da falta de primitivas aritméticas de alto nível e do foco direcionado ao ecossistema da Rigetti Computing. Isso levou à escolha de não implementar sua etapa quântica, embora as etapas pré e pós-processamento tenham sido contempladas. A comparação mostra que, apesar de todas as tecnologias serem adequadas para experimentação acadêmica e desenvolvimento quântico, cada uma possui suas próprias vantagens e limitações: o *Qiskit* se destaca pela maturidade e completude, o *Cirq* pela flexibilidade a nível de circuito, e o *PyQuil* pela integração direta com *hardware* proprietário, ainda que com recursos menos abrangentes.

Palavras-chave: Computação Quântica, Algoritmo de Shor, Cirq, Qiskit, PyQuil

Lista de ilustrações

Lista de tabelas

Lista de algoritmos

Lista de abreviaturas e siglas

QFT	*Quantum Fourier Transform*
QPU	*Quantum Processing Unit*
QSE	*Quantum Software Engineering*
CNOT	*Controlled-NOT*

Sumário

1 Introdução

A Computação tradicional, popularmente conhecida, baseia-se em máquinas que operam segundo os princípios da Física Clássica. Nessas máquinas, os componentes apresentam comportamentos previsíveis e bem compreendidos, funcionando de forma intuitiva, onde o estado de cada elemento pode ser descrito e observado de maneira objetiva.

Com o progresso dos computadores clássicos, muitos problemas puderam ser resolvidos de maneira eficaz. No entanto, ainda há problemas computacionais cuja complexidade torna inviável resolvê-los com máquinas convencionais, mesmo com recursos de processamento avançados. Diante dessas limitações, surgem novas abordagens capazes de explorar paradigmas distintos de computação, entre as quais se destaca a Computação Quântica - um campo interdisciplinar que combina fundamentos da Mecânica Quântica e da Ciência da Computação com o objetivo de desenvolver dispositivos capazes de realizar operações além do alcance dos computadores clássicos.

O primeiro modelo de uma máquina quântica foi proposto por Paul Benioff (BENIOFF, 1980), físico estadunidense considerado pioneiro nos estudos da área, o qual no ano de 1980 em um de seus artigos, propusera um "microscópico modelo de Mecânica Quântica representado por máquinas de Turing". Já em 1981, Richard P. Feynman apresentou problemas clássicos da física que a computação tradicional não seria capaz de representar, mas a quântica sim (FEYNMAN, 1981). Visando superar esses problemas, surgiram os primeiros modelos de circuitos quânticos, similares aos circuitos lógicos dos computadores clássicos, e os primeiros algoritmos quânticos.

Desde então, a área da Computação Quântica vem sendo constantemente desenvolvida, instigando não só pesquisadores, cientistas, mas também investidores. Graças a isso, desenvolveram máquinas quânticas e simuladores destas, nos quais é possível executar algoritmos e avaliar o potencial dessas máquinas. Por exemplo, já existem bibliotecas para algumas linguagens de programação, como o *PyQuil* (KOCH; WESSING; ALSING, 2019) para a linguagem *Python*, que simulam operações de computadores quânticos.

A Computação Quântica tem sido amplamente discutida, especialmente em relação aos seus potenciais impactos em diversos campos, como a Criptografia, bem como aos desafios inerentes à construção e viabilização prática desses dispositivos. Embora o estado atual da tecnologia ainda esteja distante do ideal almejado, observa-se uma evolução significativa e constante nesse campo, impulsionada por avanços teóricos e experimentais.

1.1 Justificativa

A área de Computação Quântica, embora comumente percebida como recente, vem sendo desenvolvida há mais de quatro décadas. Anos após as primeiras formulações teóricas sobre máquinas quânticas, foi construído o primeiro computador quântico funcional, capaz de realizar cálculos elementares (CHUANG; GERSHENFELD; KUBINEC, 1998). Contudo, à época, as limitações impostas pelos recursos computacionais e pelas tecnologias disponíveis restringiam significativamente o avanço prático da área.

Com o progresso contínuo da ciência e da engenharia, surgiram ferramentas mais sofisticadas que viabilizam desde a escrita e execução de códigos que simulam o comportamento de computadores quânticos até o acesso a dispositivos quânticos reais por meio de plataformas em nuvem.A análise e experimentação dessas ferramentas permitem a construção de uma base de conhecimento sólida sobre os recursos já disponíveis, as limitações tecnológicas e a adequação de cada ferramenta a diferentes propósitos. Além disso, esse processo contribui para a identificação de funcionalidades relevantes, pontos fortes e aspectos que ainda demandam aprimoramento.

Diante da diversidade de tecnologias disponíveis no campo da Computação Quântica, justifica-se a elaboração de um trabalho que consolide informações sobre o atual estado das tecnologias. A proposta é reunir dados comparativos que contribuam para o processo de seleção das ferramentas mais adequadas a projetos, pesquisas e aplicações futuras, otimizando tempo e fornecendo uma base sólida para tomadas de decisão.

1.2 Objetivos

Este trabalho tem como objetivo principal avaliar diferentes tecnologias de Computação Quântica, por meio de uma análise comparativa baseada em características funcionais específicas de cada tecnologia, além de revisar e atualizar estudos já realizados na área. Para alcançar esse propósito, são definidos os seguintes objetivos específicos:

- levantar e sistematizar informações sobre tecnologias para a programação quântica;
- apresentar dados atualizados a respeito das tecnologias selecionadas;
- realizar experimentações práticas com as tecnologias escolhidas;
- identificar os recursos disponíveis e limitações associadas a cada tecnologia;
- comparar as tecnologias considerando critérios como desempenho, usabilidade, qualidade da documentação disponível e perspectiva de atualização futura.

1.3 Metodologia

Com o intuito de atingir os objetivos propostos neste trabalho, inicialmente foi conduzido um estudo do estado da arte da Computação Quântica. Essa etapa teve como finalidade identificar trabalhos correlatos que possibilitassem a compreensão dos algoritmos implementados com o uso de tecnologias quânticas, bem como as linguagens de programação, bibliotecas, arquiteturas e plataformas envolvidas nesses projetos. Além disso, foram analisadas as dificuldades enfrentadas durante essas implementações, com o intuito de mapear desafios comuns e soluções adotadas.

Paralelamente, foram realizadas buscas por estudos que compartilhassem objetivos similares aos deste trabalho, especialmente aqueles voltados à avaliação comparativa de ferramentas e plataformas quânticas. Também foram consultadas fontes que oferecessem informações técnicas sobre os principais recursos disponíveis no mercado, com o objetivo de selecionar aqueles que mais contribuiriam para a proposta em questão.

A partir da análise realizada, foram escolhidas três ferramentas distintas, preferindo aquelas de código aberto (*open source*). Cada uma dessas ferramentas foi utilizada para a implementação de um algoritmo clássico da Computação Quântica: o algoritmo de Shor (SHOR, 1994). As implementações serviram como base empírica para a coleta de métricas e informações relevantes à análise comparativa proposta.

Os critérios adotados para a avaliação das ferramentas selecionadas foram: desempenho computacional, usabilidade, nível de documentação disponível e potencial de atualização. Tais critérios foram definidos com base em sua relevância para a análise prática de tecnologias voltadas à Computação Quântica.

No que se refere ao desempenho, compreende-se que a mensuração da performance de operações quânticas apresenta desafios particulares, tendo em vista a complexidade envolvida na execução de algoritmos em dispositivos quânticos (WANG; GUO; SHAN, 2022). Frequentemente, utiliza-se a quantidade de *qubits* disponíveis como um dos indicadores de capacidade da plataforma. Entretanto, também é essencial considerar a quantidade de *qubits* necessária para a execução de determinados algoritmos, uma vez que diferentes tecnologias impõem restrições quanto a esse aspecto. Neste trabalho, optou-se por uma abordagem pragmática: a avaliação do desempenho foi realizada com base no tempo de execução das operações simuladas ou executadas.

Para o critério de usabilidade, foram considerados fatores como a existência de interfaces gráficas para o usuário, a disponibilidade de interfaces de linha de comando, a compatibilidade com diferentes sistemas operacionais e dispositivos, e a facilidade de acesso, instalação e utilização das ferramentas. Essa análise teve como foco a experiência do usuário durante o uso prático de cada tecnologia.

Em relação ao nível de documentação, foram avaliadas tanto as documentações

oficiais disponibilizadas pelas equipes mantenedoras das ferramentas quanto a existência de suporte comunitário, especialmente em fóruns técnicos e plataformas de perguntas e respostas, como o *Stack Overflow*. Parte-se do princípio de que uma maior base de usuários implica uma troca de conhecimento mais ampla, o que contribui diretamente para a robustez da documentação disponível.

Por fim, o potencial de atualização de cada ferramenta foi aferido com base no histórico de versões e atualizações anteriores, na frequência com que novos lançamentos são disponibilizados, no grau de engajamento dos desenvolvedores e mantenedores, bem como na atividade do projeto em plataformas de controle de versão, como o *GitHub*. Esses indicadores permitiram inferir a maturidade e a sustentabilidade do desenvolvimento contínuo das tecnologias analisadas.

2 Fundamentação e Referencial Teórico

Nesse capítulo são apresentados conceitos fundamentais para um melhor entendimento do trabalho. Trata-se desde definições básicas da Física Quântica, propriedades físicas, até pontos que caracterizam a área da Computação Quântica e que serão frequentemente citados aqui. Em seguida, são apresentados trabalhos relacionados, que contribuem para o atual estado da arte da Computação Quântica e ajudam a entender como esse trabalho está posicionado junto aos demais que vêm sendo desenvolvidos.

2.1 Visão Geral da Física Quântica

No início do século XX, os modelos da Física Clássica mostraram-se insuficientes para explicar determinados fenômenos físicos observados em escalas atômicas e subatômicas, como o efeito fotoelétrico. Esses fenômenos apresentavam resultados experimentais diferentes das previsões teóricas então conhecidas. Diante dessas inconsistências, físicos como Max Planck, Albert Einstein e outros pioneiros, começaram a questionar o comportamento da matéria e da energia em níveis microscópicos, propondo novos conceitos que mais tarde dariam origem à Física Quântica.

Os primeiros trabalhos relacionados à área da Física Quântica propuseram novas teorias e entendimentos sobre a interação entre matéria e energia, abordando a quantização dos níveis de energia atômica (PLANCK, 1900), a natureza dual da luz e a estrutura do átomo (BOHR, 1913). A partir desses trabalhos e com o avanço da Física Quântica, consolidaram-se os principais fundamentos que sustentam a teoria quântica moderna, como a Dualidade Onda-Partícula, o Princípio da Superposição, o Princípio da Incerteza e o Entrelaçamento Quântico.

2.1.1 Dualidade Onda-Partícula

Em 1887, o físico Heinrich Hertz obteve resultados que contrariavam teorias já estabelecidas sobre o comportamento da luz (HERTZ, 1887). Anos depois, em 1905, Einstein reforçou os resultados de Heinrich, explicando um fenômeno que ficou conhecido como "Efeito Fotoelétrico". Como consequência disso, verificou-se que a luz e a matéria apresentam tanto características ondulatórias quanto corpusculares, dependendo da forma de análise, originando então o princípio da Dualidade Onda-Partícula.

Na natureza, partículas são entidades com posição definida no espaço, possuindo massa e forma definida. Ao colidirem, podem perder energia e até se dissipar. Em contraste, ondas são perturbações no espaço sem posição fixa ou massa, transportam energia

e, ao interagir, estão sujeitas a fenômenos como refração e difração, podendo se cancelar ou se combinar, alterando sua amplitude.

2.1.2 Princípio da Superposição

O princípio da Superposição estabelece que um objeto pode existir simultaneamente em múltiplos estados. Esse conceito desafia a intuição, pois o cotidiano sugere que os objetos possuem estados bem definidos. Para ilustrar tal princípio, diversos experimentos e analogias foram propostos ao longo dos anos, sendo o experimento mental do gato de Schrödinger um dos mais populares (SCHRÖDINGER, 1935).

Nesse experimento, Erwin Schrödinger buscou demonstrar um paradoxo resultante da aplicação da Mecânica Quântica a objetos do dia a dia. Ele propôs um cenário hipotético onde um gato é colocado em uma câmara selada junto a um frasco de veneno, um mecanismo de disparo capaz de quebrar o frasco e um átomo instável (Fig. 1). O átomo pode passar por uma transformação espontânea que é detectada por um instrumento (Contador Geiger). Se essa transformação ocorrer, o sensor ativa o mecanismo quebrando o frasco e liberando o veneno, resultando na morte do gato. Caso contrário, o frasco permanece intacto e o gato continua vivo.

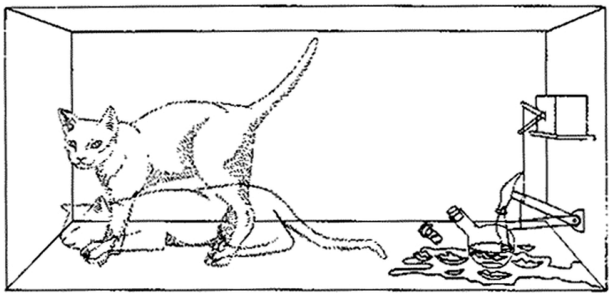

Figura 1 – Sobreposição de Estados no Experimento. Autor: Harrison (1999)

Pelo princípio da Superposição, antes de abrir a câmara, o átomo está em uma condição de superposição de dois possíveis estados - transformado e não transformado. Como a situação do gato depende da transformação do átomo, ele também está em uma superposição de vivo e morto. Ao abrir a câmara, é possível observar a condição do átomo e do gato, avaliando e definindo seus estados, tirando-os da condição de superposição. A partir desse experimento mental, destacou-se a essência deste princípio e impulsionou diversas interpretações sobre o papel da medição de estados na Mecânica Quântica.

2.1.3 Princípio da Incerteza

Algumas grandezas físicas podem ser medidas com facilidade, como o peso de um alimento, a altura de um edifício ou a intensidade de uma música. No entanto, ao medir propriedades de objetos com características ondulatórias (Seção 2.1.1), a complexidade

aumenta. No contexto quântico, todas as partículas, em certas condições, apresentam características de onda.

O Princípio da Incerteza (HEISENBERG, 1927) estabelece que não é possível determinar simultaneamente, com exatidão, a posição e o momento (velocidade) de uma partícula. Quão maior a precisão de uma medida, menor é a precisão da segunda medida. Embora partículas não exibam de forma evidente características de onda em escalas macroscópicas, ao serem observadas em nível molecular, a Dualidade Onda-Partícula se manifesta, revelando que também podem apresentar comportamento ondulatório.

2.1.4 Entrelaçamento Quântico

Descrito por Einstein (BORN et al., 1971) como uma "ação fantasmagórica à distância", o Entrelaçamento Quântico ocorre quando duas ou mais partículas se correlacionam de forma que o estado de uma determina o estado das demais, independente da distância. Assim, não é possível considerar cada partícula isoladamente, pois seus estados permanecem interligados, formando um sistema único. A medição do estado de uma das partículas entrelaçadas define o estado das demais, rompendo a superposição inicial.

Esse comportamento desafia a visão clássica, segundo a qual qualquer interação entre partículas exigiria uma transmissão de informação limitada pela velocidade da luz. No entanto, o entrelaçamento não implica em uma comunicação entre partículas, mas sim uma conexão fundamental entre seus estados quânticos, de forma que a medição de uma delas afeta instantaneamente a descrição da outra, sem, contudo, violar os Princípios da Relatividade de Einstein (YANOFSKY; MANNUCCI, 2008).

O fenômeno do Entrelaçamento Quântico pode ocorrer em diversos contextos, dependendo das interações físicas entre os corpos envolvidos. Por exemplo, elétrons inicialmente independentes podem, ao se colidirem, estabelecer um entrelaçamento quântico, passando a exibir o comportamento descrito.

2.2 Visão Geral da Computação Quântica

A Computação Quântica baseia-se nos princípios da Física Quântica para processar informações de maneira distinta dos computadores clássicos. Muitos conceitos familiares à Computação Clássica não se aplicam integralmente no paradigma quântico, exigindo uma nova abordagem para compreensão.

Esta seção apresenta os fundamentos da Computação Quântica e, sempre que possível, estabelece comparações com a Computação Clássica para destacar as diferenças entre esses dois modelos computacionais.

2.2.1 *Qubits*

Na Computação Clássica, a informação é armazenada em *bits* que podem assumir apenas dois estados distintos, estado 0 ou 1, sem possibilidade de estados intermediários. A Computação Quântica, por sua vez, introduz um conceito de unidade de informação que se comporta de maneira diferente. Conforme Yanofsky e Mannucci (2008), o *qubit*, abreviação para *quantum bit*, baseia-se em princípios da Física Quântica (Seção 2.1) e pode existir em uma superposição dos estados 0 e 1, representados por uma combinação linear desses estados base.

Uma forma intuitiva de visualizar a diferença entre *bits* e *qubits* é através da analogia com interruptores de luz. Enquanto um *bit* se comporta como um interruptor convencional, podendo estar estritamente ligado ou desligado, o *qubit* funciona como um interruptor com controle de intensidade, podendo estar ligado, desligado ou em um estado intermediário representado por uma superposição dos estados base - uma combinação de ligado e desligado. A possibilidade de um *qubit* existir em superposições permite que um único *qubit* represente um espectro mais amplo de informações do que um *bit* tradicional.

A capacidade de um *qubit* assumir vários estados simultaneamente confere à Computação Quântica um alto potencial de processamento. Ao explorar essa característica, máquinas quânticas podem avaliar diversas possibilidades ao mesmo tempo, possibilitando ganhos significativos em eficiência para determinadas classes de problemas quando comparadas às máquinas tradicionais.

2.2.2 Estados Quânticos

Dadas as propriedades apresentadas anteriormente (Seção 2.2.1), a representação do estado de um *qubit* é mais complexa que a de um *bit* clássico. Enquanto um *bit* tradicional pode ser descrito como 0 ou 1, os estados quânticos são expressos matematicamente como vetores. Para representar estados quânticos, a Mecânica Quântica adota a Notação *Bra-ket*, ou Notação de Dirac (DIRAC, 1939), que fornece uma forma concisa e padronizada de descrever estados quânticos.

Na Notação de Dirac, os estados base de um *qubit* são representados por $|0\rangle$ e $|1\rangle$, que correspondem, respectivamente, aos estados clássicos 0 e 1. Os estados base também podem ser escritos de forma matricial (Eq. 2.1).

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.1}$$

Quando escritos usando vetores (Eq. 2.1), cada elemento do vetor representa a contribuição do estado correspondente no espaço vetorial do *qubit*. O primeiro elemento

indica a amplitude, ou "proporção", do estado $|0\rangle$, enquanto o segundo representa a amplitude associada ao estado $|1\rangle$.

De maneira geral, sendo o estado arbitrário de um *qubit* $|\psi\rangle$, tal estado pode ser expresso como uma combinação linear dos estados base $|0\rangle$ e $|1\rangle$ (Eq. 2.2), com os escalares α e β representando, respectivamente, a amplitude do estado $|0\rangle$ e do estado $|1\rangle$.

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{2.2}$$

As variáveis α e β, amplitudes de probabilidade, são números complexos que determinam a contribuição de cada estado base no estado de superposição de um *qubit*. O valor absoluto ao quadrado das amplitudes, $|\alpha|^2$ e $|\beta|^2$, representa a probabilidade de que ao medir o *qubit* ele colapse para o estado $|0\rangle$ ou $|1\rangle$, respectivamente. Portanto, a soma da probabilidade de um *qubit* assumir um estado base ao ser medido é igual a 1, ou seja, a igualdade $|\alpha|^2 + |\beta|^2 = 1$ é válida.

Até a ação de medição, um *qubit* não está exclusivamente em um dos estados $|0\rangle$ ou $|1\rangle$ mas sim em uma superposição de ambos. Ao ser medido, o *qubit* assume um estado base, deixando a condição de superposição.

2.2.3 Esfera de *Bloch*

A representação dos estados quânticos pode ser abstrata, tornando sua interpretação menos intuitiva. Para facilitar a compreensão, utiliza-se a Esfera de Bloch (ARECCHI et al., 1972), uma representação geométrica que descreve o estado de um *qubit* como um ponto na superfície de uma esfera tridimensional (Fig. 2). Essa abordagem permite visualizar a superposição quântica e a relação entre os estados base $|0\rangle$ e $|1\rangle$.

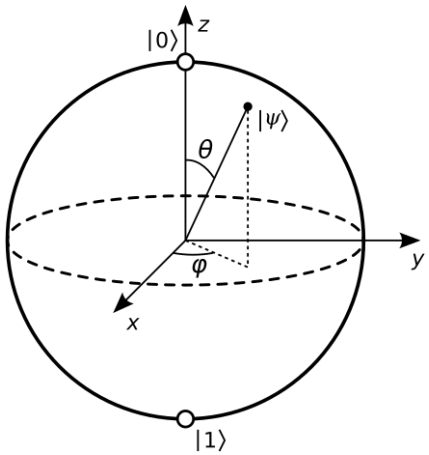

Figura 2 – Esfera de Bloch. Fonte: Wikimedia Foundation (2025)

Na Esfera de Bloch, o polo norte representa o estado base $|0\rangle$ e o polo sul, o estado $|1\rangle$. Qualquer outro ponto na superfície corresponde a um estado de superposição entre

$|0\rangle$ e $|1\rangle$, determinado pelos ângulos *theta* θ e *phi* ϕ. O primeiro mede a inclinação a partir do polo norte, controlando a projeção do estado de um *qubit* no eixo Z, definindo a distribuição de probabilidade entre os estados $|0\rangle$ e $|1\rangle$. O segundo mede a rotação ao longo do equador da esfera, representa a fase relativa entre os estados base e não interfere na medição dos estados. A posição de um ponto na superfície dessa esfera também pode ser escrita matematicamente (Eq. 2.3).

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle \tag{2.3}$$

A equação acima é uma combinação linear dos estados base, onde o valor absoluto ao quadrado das amplitudes (escalares) representa a probabilidade de que ao medir um *qubit* ele colapse para um dos estados $|0\rangle$ ou $|1\rangle$. Ou seja, a chance de medir $|0\rangle$ é igual a $\cos^2(\theta/2)$ e $|1\rangle$, $\sin^2(\theta/2)$.

A Esfera de Bloch torna mais simples a visualização de um estado quântico como uma combinação de estados e facilita a demonstração do princípio da superposição. Além disso, é essencial para o entendimento de portas quânticas, pois operações quânticas podem ser vistas como rotações ao redor dos eixos dessa esfera.

2.2.4 Portas Quânticas

Na Computação Quântica, as portas quânticas desempenham um papel essencial, assim como as portas lógicas na Computação Clássica. Conforme Yanofsky e Mannucci (2008), cada porta representa uma operação diferente aplicada ao estado de um ou mais *qubits*, que pode ser descrita por uma matriz unitária e visualizada como rotações ao redor dos eixos da Esfera de Bloch.

Ao aplicar uma porta quântica a um *qubit*, ocorre uma transformação linear sobre o vetor de estado do *qubit*. Essa transformação pode ser representada através de matrizes, preservando a norma do vetor e garantindo que a soma das amplitudes (Eq. **??**) do estado do *qubit* permaneça igual a 1. Em outras palavras, sendo U a matriz que descreve uma porta quântica, a aplicação dessa porta a um *qubit* com estado $|\psi\rangle$ pode ser representada pelo produto matricial $|\psi'\rangle = U|\psi\rangle$.

A porta quântica X (Eq. 2.4), análoga à porta NOT clássica, inverte o estado de um *qubit*. Sua operação pode ser vista como uma rotação de 180° em torno do eixo X da Esfera de Bloch. Quando aplicada ao estado $|0\rangle$, a matriz que representa a porta X o transforma em $|1\rangle$, e vice-versa (Eq. 2.5).

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{2.4}$$

$$X\left|0\right\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \left|1\right\rangle \tag{2.5}$$

A porta Y (Eq. 2.6) também inverte os estados de *qubits*, porém adiciona um fator de fase imaginário. Sua operação corresponde a uma rotação de 180° em torno do eixo Y da Esfera de Bloch. Quando aplicada ao estado base $\left|0\right\rangle$ resulta em $i\left|1\right\rangle$, enquanto aplicada ao estado $\left|1\right\rangle$ produz $-i\left|0\right\rangle$ (Eq. 2.7).

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \left|0\right\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \left|1\right\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.6}$$

$$Y\left|0\right\rangle = i\left|1\right\rangle, \quad Y\left|1\right\rangle = -i\left|0\right\rangle \tag{2.7}$$

Diferente das portas X e Y, a porta Z (Eq. 2.8) não altera as amplitudes dos estados base $\left|0\right\rangle$ e $\left|1\right\rangle$, apenas adiciona uma fase de π ao estado $\left|1\right\rangle$ (Eq. 2.9). Sua operação equivale a uma rotação de 180° em torno do eixo Z da Esfera de Bloch.

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad \left|0\right\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \left|1\right\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.8}$$

$$Z\left|0\right\rangle = \left|0\right\rangle, \quad Z\left|1\right\rangle = -1\left|1\right\rangle \tag{2.9}$$

A porta *Hadamard* (Eq. 2.10), fundamental para a Computação Quântica, transforma os estados base em uma superposição equilibrada. Quando aplicada ao estado $\left|0\right\rangle$, a porta H produz um estado de superposição de igual probabilidade entre $\left|0\right\rangle$ e $\left|1\right\rangle$ (Eq. 2.11). Sua operação pode ser vista como uma rotação de 180° ao redor da diagonal entres os eixos X e Z da Esfera de Bloch e é crucial para algoritmos como o de Shor.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \left|0\right\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{2.10}$$

$$H\left|0\right\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{\left|0\right\rangle + \left|1\right\rangle}{\sqrt{2}} \tag{2.11}$$

Diferente das demais, que operam em um único *qubit*, a porta CNOT (*Controlled-Not*, Eq. 2.12) manipula dois *qubits* e tem funcionamento similar ao da porta XOR (*Exclusive-OR*) clássica. Em sua operação, o primeiro *qubit* é chamado *qubit* de controle e o segundo, *qubit* alvo. Ao aplicar a porta CNOT, se o *qubit* de controle estiver no estado $\left|1\right\rangle$, então o estado do *qubit* alvo é invertido (Eq. 2.13); senão, nada se altera.

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad |10\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.12}$$

$$CNOT\,|10\rangle = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = |11\rangle \tag{2.13}$$

2.2.5 Circuitos Quânticos

Os circuitos quânticos representam a organização sequencial de operações sobre um conjunto de *qubits*. Cada operação corresponde a uma porta quântica, que realiza transformações no estado dos *qubits*, conforme visto na seção anterior (Seção 2.2.4).

A notação gráfica dos circuitos segue um padrão bem definido. Cada linha horizontal representa um *qubit*, e as operações aplicadas aos *qubits* são dispostas ao longo dessas linhas em ordem temporal, da esquerda para a direita. As portas quânticas são indicadas por símbolos específicos, geralmente retângulos rotulados com a operação correspondente. Conexões verticais representam interações entre *qubits*, como no caso de portas de múltiplos *qubits*, por exemplo, a porta CNOT.

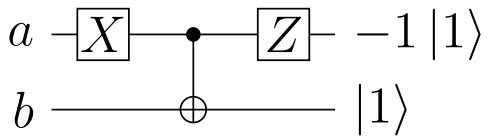

Figura 3 – Pequeno diagrama de circuito quântico

A Figura 3 ilustra um circuito quântico composto por dois *qubits* a e b que estão inicialmente no estado $|0\rangle$, com três portas quânticas - X, CNOT ($\bigoplus$) e Z. De início, o *qubit* a passa pela porta X, que inverte seu estado de $|0\rangle$ para $|1\rangle$. Em seguida, a porta CNOT é aplicada, onde a atua como *qubit* de controle e b como *qubit* alvo. Como a está em $|1\rangle$, b tem seu estado invertido para $|1\rangle$. Por fim, a passa pela porta Z, encerrando o circuito com estado igual à $-1\,|1\rangle$, enquanto b, $|1\rangle$.

2.2.6 Algoritmo de Shor

O algoritmo de Shor, proposto em Shor (1994), é um marco significativo da Computação Quântica. Trata-se de um algoritmo quântico eficiente para a fatoração de inteiros, operando em tempo polinomial, em contraste com os métodos clássicos, que exigem tempo exponencial para a mesma tarefa. Sua relevância se deve ao fato de que a fatoração de números inteiros grandes é um problema computacionalmente difícil para algoritmos clás-

sicos, tornando-se um dos primeiros exemplos concretos da superioridade da Computação Quântica sobre a Computação Clássica.

Com o avanço de tecnologias quânticas, o algoritmo de Shor tem ganho crescente interesse, especialmente devido a seu impacto na segurança da informação. Diversos sistemas criptográficos modernos, incluindo o amplamente utilizado *RSA* (RIVEST; SHAMIR; ADLEMAN, 1978), fundamentam-se na dificuldade de fatoração de grandes números. A capacidade de um computador quântico rodando o algoritmo de Shor realizar essa fatoração de forma eficiente representa uma ameaça direta a esses sistemas, impulsionando pesquisas em criptografia pós-quântica, que busca desenvolver métodos seguros mesmo diante do avanço da Computação Quântica.

Nas seções seguintes, o algoritmo será abordado com maior profundidade, detalhando suas etapas e o princípio matemático que permite sua eficiência na fatoração de inteiros.

2.3 Análise de Tecnologias Quânticas

Com o crescimento acelerado das tecnologias voltadas para a Computação Quântica, tornou-se um desafio escolher quais ferramentas e plataformas utilizar em projetos específicos. Esse cenário motivou diversos pesquisadores a desenvolver estudos comparativos e analíticos sobre as tecnologias disponíveis. A seguir, são apresentados alguns desses trabalhos, destacando suas conclusões e contribuições.

A comparação de computadores clássicos normalmente envolve critérios como capacidade de memória, potência do processador e presença de um *chip* gráfico dedicado. No entanto, quando se trata de computadores quânticos, ainda não existem métricas padronizadas universalmente aceitas para avaliação de desempenho.

Conforme discutido em Wang, Guo e Shan (2022), os autores argumentam que a contagem de *qubits* não é, por si só, um indicador confiável do desempenho de uma máquina quântica, podendo ser até enganosa. Para contornar essa limitação, são propostas três categorias de métricas para comparação, analisando desde aspectos físicos da construção das máquinas quânticas até testes com aplicações quânticas reais. Os autores concluem que ainda não existem formas perfeitas de medir o desempenho dessas máquinas, mas o desenvolvimento de ferramentas de *benchmark* quântico é fundamental para o progresso da Computação Quântica.

No contexto de ferramentas de desenvolvimento, LaRose (2019) realiza uma análise comparativa de quatro populares plataformas - *Forest* (*pyQuil*), *Qiskit*, *ProjectQ* e *Quantum Developer Kit* (Q#). O autor fornece informações detalhadas sobre o processo de instalação, o nível de documentação disponível, a sintaxe das linguagens associadas

a cada ferramenta e os recursos de *hardware* oferecidos. Como conclusão, o autor sugere algumas plataformas de acordo com diferentes casos de uso, como *Forest* (*PyQuil*) e *Qiskit* para usuários familiarizados com a linguagem *Python*, e reforça que todas as plataformas analisadas representam avanços significativos para a Computação Quântica.

Com a evolução das tecnologias quânticas, torna-se inevitável a consolidação de princípios para a Engenharia de Software Quântico (em inglês: *Quantum Software Engineering* - QSE). Nesse contexto, Piattini et al. (2020) propõe o Manifesto de Talavera, que estabelece diretrizes para o desenvolvimento de software quântico de maneira industrializada. O manifesto sugere a adoção de práticas já consagradas na Engenharia de Software Clássico. O autor reforça que a maturidade da QSE dependerá do desenvolvimento de metodologias robustas e de um ecossistema de ferramentas adequado.

Complementando as análises comparativas, Serrano et al. (2022) realiza uma extensa comparação entre diversas tecnologias para Computação Quântica, categorizando-as em três grupos - linguagens de programação, bibliotecas auxiliares e plataformas de desenvolvimento e execução de aplicações. O autor apresenta uma linha do tempo com a evolução das linguagens quânticas e as classifica de acordo com seu paradigma e nível de abstração. São comparadas linguagens como *pyQuil* (SMITH; CURTIS; ZENG, 2016), *Cirq*, *Q#* (SVORE et al., 2018) e *QML* (GAY; NAGARAJAN, 2005).

Além disso, Serrano et al. (2022) avalia bibliotecas auxiliares, analisando aspectos como facilidade de uso, interface gráfica, tipo de licença e recursos disponíveis. Também são discutidas plataformas que permitem a simulação e execução de algoritmos quânticos em hardware real, bem como as principais empresas que desenvolvem dispositivos quânticos. O autor conclui que, devido ao rápido avanço da Computação Quântica, revisões e comparativos assim devem ser atualizados com frequência. Por fim, destaca a importância da criação de repositórios colaborativos para centralizar informações sobre novas ferramentas e tecnologias voltadas à Computação Quântica.

2.4 Métricas para Comparação de Tecnologias Quânticas

A análise comparativa das tecnologias selecionadas neste trabalho será conduzida com base em critérios qualitativos e quantitativos. Alguns desses critérios foram extraídos da análise proposta por Serrano et al. (2022), enquanto outros foram definidos para atender às necessidades deste estudo.

A seguir, apresenta-se a definição de cada critério, bem como os possíveis valores atribuídos a cada um:

- **Tipo:** refere-se à forma como a tecnologia é disponibilizada ao usuário. Pode ser através de uma biblioteca, contendo um conjunto de funções e algoritmos pré-

implementados para uma determinada linguagem de programação, visando a execução de tarefas específicas, ou como um conjunto de ferramentas (*toolkit*), com um conjunto mais abrangente de funcionalidades ou ambos;

- **Licenciamento:** define o modelo de distribuição e uso da tecnologia. Pode ser uma tecnologia de código aberto (*open source*), onde o código-fonte está disponível para inspeção, modificação, contribuição e utilização da comunidade, ou uma tecnologia comercial que requer aquisição de licença ou assinatura para utilizar;
- **Linguagem de Programação:** especifica a linguagem de programação na qual a tecnologia foi implementada ou para a qual foi projetada. Exemplos incluem *Python*, C++, entre outras.
- **Interface:** especifica como o usuário interage com a aplicação. Algumas tecnologias dispõem de linha de comando, interação via terminal, enquanto outras oferecem interface gráfica para o usuário. Algumas disponibilizam ambos;
- **Número de *Qubits*:** refere-se à quantidade de *qubits* disponíveis para uso na execução de algoritmos, seja em um simulador ou hardware real;
- **Documentação:** avalia a disponibilidade e abrangência da documentação oficial e da comunidade considerando três níveis - baixo, escassa ou pouco detalhada; moderado, suficiente para uso básico; alto, documentação extensa, bem detalhada e suportada pela comunidade;
- **Atualização e suporte:** mede a regularidade das atualizações da tecnologia e a existência de um plano de desenvolvimento futuro (*roadmap*), considerando também três níveis - baixo, moderado e alto;
- **Facilidade de Implementar Shor:** avalia o grau de esforço necessário para implementar o algoritmo de Shor utilizando os recursos nativos da tecnologia. Considera a disponibilidade de componentes prontos, como a Transformada de Fourier Quântica (QFT) e exponenciação modular, bem como o nível de abstração oferecido na construção de circuitos aritméticos;
- **Suporte a Portas Aritméticas:** analisa a existência de portas e operações aritméticas prontas, como soma e multiplicação modular, que são essenciais para algoritmos de fatoração e outros baseados em aritmética quântica. Tecnologias sem esse suporte requerem a construção manual de portas compostas, aumentando a complexidade de implementação;
- **Compatibilidade com *Hardware* Real:** indica se a tecnologia permite a execução de circuitos em computadores quânticos reais, e, em caso positivo, com quais provedores é compatível (por exemplo, *IBM Quantum*, *Google Sycamore* ou *Rigetti*).

Esse critério também considera restrições de acesso, como disponibilidade pública ou necessidade de credenciais institucionais;

- **Desempenho de Simulação:** mede a eficiência e escalabilidade dos simuladores oferecidos pela tecnologia, considerando fatores como capacidade máxima de *qubits*, tempo de execução e otimizações internas. Simuladores mais otimizados permitem experimentos mais realistas e rápidos, especialmente em ambientes sem acesso a *hardware* real;

- **Adequação ao Algoritmo de Shor:** avalia a pertinência geral da tecnologia para a implementação do algoritmo de Shor, considerando de forma integrada critérios como suporte aritmético, abstração de alto nível, desempenho de simulação e facilidade de integração entre camadas clássica e quântica. Esse critério sintetiza a capacidade da tecnologia de suportar todas as etapas do algoritmo com eficiência;

- **Outros recursos:** abrange funcionalidades extras oferecidas pela tecnologia como algoritmos pré-implementados, ferramentas de visualização gráficas (diagramas), otimização de código e suporte a paralelismo.

3 Implementação do Algoritmo de Shor

A implementação do Algoritmo de Shor foi realizada utilizando a linguagem de programação *Python*, uma vez que as três tecnologias analisadas - *Cirq* (CIRQ, 2018), Qiskit (CROSS, 2018) e PyQuil (SMITH; CURTIS; ZENG, 2016) - possuem suporte consolidado para ela. Para garantir compatibilidade com as bibliotecas e pacotes mais recentes, adotou-se a versão *Python* 3.11.0.

O gerenciamento das versões, dependências e ambientes virtuais foi conduzido com a ferramenta *pyenv*, permitindo a criação de três ambientes isolados, um para cada tecnologia quântica escolhida. Essa abordagem evita conflitos entre pacotes e assegura que cada tecnologia opere de forma independente.

Por fim, para o desenvolvimento e análise dos experimentos, utilizou-se o *Jupyter Notebook* (KLUYVER et al., 2016), que possibilita a integração de código , visualizações e explicações textuais em um único ambiente. Esse formato facilita a organização da implementação e contribui para a reprodutibilidade dos testes.

3.1 Descrição do Algoritmo de Shor

De forma resumida, o algoritmo proposto em Shor (1994) tem como objetivo fatorar um número inteiro N de maneira eficiente. Para isso, seleciona-se um valor aleatório g, denominado palpite (*guess*), e busca-se determinar o período r da função $f(x) = g^x \bmod N$ (Eq. 3.3). Identificado esse período, é possível calcular $g^{r/2} \pm 1$, obtendo valores que, com grande probabilidade, compartilham fatores comuns com N, possibilitando a fatoração em seus primos constituintes. Vale destacar que este trabalho não tem como objetivo aprofundar-se nas demonstrações matemáticas que fundamentam o algoritmo.

A implementação do algoritmo pode ser organizada em três etapas principais: i) reduzir o problema da fatoração; ii) encontrar o período da função desejada; iii) encontrar os fatores não triviais a partir do período encontrado. O pseudocódigo apresentado que reflete o algoritmo de Shor (Alg. 1) contempla: i) verificações iniciais (pré-processamento), ii) etapa quântica do algoritmo (busca pelo período) e iii) processamento final para encontrar os fatores não triviais de N.

A princípio, verifica-se se N pode ser fatorado de maneira trivial (Alg. 1, linhas 3 a 9), seja por ser um número par ou por ser uma potência perfeita de um número primo. Nesse último caso, diz-se que N é uma potência prima perfeita quando pode ser expresso como $N = b^k$, em que b e k são inteiros, com $b > 1$ e $k \geq 2$, sendo b um número primo. Para identificar essa condição, testam-se valores inteiros de k dentro de um intervalo limitado,

calculando-se a raiz k-ésima de N e verificando se o arredondamento dessa raiz, elevado novamente a k, resulta exatamente em N. Se essa igualdade for satisfeita, b é considerado fator não trivial de N. A verificação de potência prima perfeita (Alg. 1, linhas 7 e 8) se desdobra em diversas instruções a depender da linguagem de programação utilizada.

Caso nenhuma dessas condições seja atendida, o algoritmo prossegue selecionando um palpite inicial g, escolhido aleatoriamente no intervalo $1 < g < N$. Esse valor deve ser coprimo com N, isto é, o máximo divisor comum entre g e N deve ser igual a 1. Se, entretanto, o valor escolhido de g for um fator não trivial de N, o processo de fatoração termina imediatamente com sucesso. Do contrário, o algoritmo parte para a etapa de processamento quântico para encontrar o período r da função $f(x) = g^x \bmod N$. Uma vez obtido o período, ele é utilizado na etapa final para o cálculo dos fatores não triviais de N. Cada uma dessas etapas é detalhada nas próximas seções.

Algoritmo 1 Pseudocódigo do algoritmo de Shor para fatoração de inteiros

1: **Entrada:** Inteiro N a ser fatorado.
2: **Saída:** Fatores não triviais (p, q) tal que $N = p * q$
▷ **Pré-processamento**
3: **Se** N é par **então**
4: **Retorne** (2, N/2)
5: **fim Se**
6: **Se** N é uma potência prima na forma b^k **então**
7: Encontre b tal que $N = b^k$
8: Reinicie o algoritmo com $N = b$
9: **fim Se**
▷ **Início do algoritmo**
10: **Enquanto** não encontrar os fatores de N **faça**
11: $g \leftarrow x \mid 1 < x < N$
12: $d \leftarrow MDC(g, N)$
13: **Se** $d > 1$ **então**
14: **Retorne** $(d, N/d)$
15: **fim Se**
▷ **Etapa Quântica**
16: Inicialize dois registradores quânticos
17: Aplique a porta Hadamard ao primeiro registrador
18: **Para** $x = 0$ até $x = N^2 - 1$ **faça**
19: Calcule g^x mod N e armazene no segundo registrador
20: **fim Para**
21: Aplique a Transformada Quântica de Fourier ao primeiro registrador
22: Meça o primeiro registrador para obter y
▷ **Etapa Final**
23: Use frações contínuas para encontrar r a partir de y
24: **Se** r é ímpar **então**
25: Continue para a próxima iteração
26: **fim Se**
27: **Se** $g^r \equiv -1 \pmod N$ **então**
28: Continue para a próxima iteração
29: **fim Se**
30: $f1 \leftarrow MDC(g^{r/2} + 1, N)$
31: $f2 \leftarrow MDC(g^{r/2} - 1, N)$
32: **Se** $1 < f1 < N$ **então**
33: **Retorne** $(f1, N/f1)$
34: **Senão Se** $1 < f2 < N$ **então**
35: **Retorne** $(f2, N/f2)$
36: **fim Se**
37: **fim Enquanto**

3.1.1 Reduzir o Problema de Fatoração

O princípio fundamental da aritmética estabelece que todo número inteiro maior que 1 pode ser decomposto de maneira única como o produto de números primos elevados a determinadas potências, por exemplo, $484 = 2^2 \times 11^2 = (2 \times 2) \times (11 \times 11) = 4 \times 121$. Independentemente da ordem dos fatores, o número 484 sempre será representado como o produto dos primos 2 e 11, cada um elevado à segunda potência. A fatoração de um número inteiro N consiste, portanto, em determinar o conjunto de fatores primos cujo produto seja igual a N.

Para números grandes, essa tarefa se torna computacionalmente complexa. No entanto, explorando propriedades da aritmética modular e da teoria de grupos, é possível reformular o problema da fatoração como a busca pelo período de uma função (YANOFSKY; MANNUCCI, 2008). Essa reformulação constitui o ponto de partida do algoritmo de Shor.

Dado um número composto N, o objetivo é encontrar fatores não triviais de N, ou seja, valores d_i tais que $1 < d_0, d_1, \cdots, d_i < N$ e que dividam N sem deixar resto. Em muitos casos, especialmente quando N é o produto de dois primos, esses fatores corresponderão diretamente aos números primos a e c, tais que $N = a \times c$. No entanto, se N possuir mais de dois fatores primos, o algoritmo poderá encontrar um fator composto, que exigirá fatorações adicionais para encontrar todos os fatores primos.

Para alcançar esse objetivo, escolhe-se aleatoriamente um número g no intervalo $1 < g < N$ e verifica-se se g é coprimo de N utilizando o Máximo Divisor Comum (MDC). Se $MDC(g, N) \neq 1$, então g já é um fator de N, e o algoritmo se encerra. Caso contrário, se $MDC(g, N) = 1$, então g é coprimo de N e o processo segue para a etapa seguinte (Alg. 1, linhas 11 a 15).

A escolha de g como coprimo de N é essencial, pois garante a existência de um grupo cíclico multiplicativo módulo N. Nesse contexto, a sequência de potências sucessivas de g módulo N eventualmente retorna ao valor 1, formando um ciclo repetitivo cuja periodicidade é chamada de ordem ou período. A ordem de $g \bmod N$ é o menor inteiro positivo r tal que $g^r \bmod N = 1$, conforme mostra a Equação 3.1.

$$g^r \equiv 1 \pmod{N} \Rightarrow g^r \bmod N = 1 \tag{3.1}$$

Essa relação decorre diretamente do Teorema de Euler, segundo o qual, se g e N são coprimos, então $g^{\phi(N)} \bmod N = 1$, onde $\phi(N)$ representa a função totiente de Euler, que indica a quantidade de inteiros positivos menores que N que são coprimos a ele. A ordem r de g módulo N sempre será um divisor de $\phi(N)$, garantindo que a congruência $g^r \equiv 1 \pmod{N}$ seja satisfeita para algum valor de r. Assim, ao chegar nessa relação, reduz-se o problema da fatoração de N à determinação do período r da função periódica $f(x) = g^x \bmod N$, que é precisamente o objetivo da etapa quântica do algoritmo de Shor.

$$g^{\phi(N)} \bmod N = 1 \tag{3.2}$$

3.1.2 Encontrar o Período de uma Função

Na subseção anterior, demonstrou-se que o problema da fatoração de um número composto N pode ser reduzido à tarefa de determinar o período r de uma função periódica da forma $f(x) = g^x \bmod N$. Esse resultado é fundamental para o algoritmo de Shor, pois permite transformar um problema clássico de fatoração em um problema de determinação de período, que pode ser resolvido de forma eficiente em um computador quântico. Assim, a etapa de busca pelo período constitui o núcleo quântico do algoritmo de Shor, sendo a parte responsável pela vantagem computacional do método proposto por Shor.

Após essa redução conceitual, o próximo passo consiste em encontrar o período r da função de exponenciação modular (Eq. 3.3), pois, a partir desse valor, será possível extrair os fatores de N. Esta etapa (Alg. 1, linhas 16 a 22) é onde o paralelismo quântico é explorado, permitindo avaliar simultaneamente múltiplos valores de x e, consequentemente, obter informações sobre o comportamento periódico de $f(x)$ de forma exponencialmente mais rápida que os algoritmos clássicos conhecidos.

$$f(x) = g^x \bmod N \tag{3.3}$$

Para encontrar o período pretendido, são utilizados dois registradores quânticos. Cada registrador consiste em um conjunto de *qubits* que representa um número binário. Por exemplo, um registrador de três *qubits* pode representar simultaneamente todos os números de 0 (000_2) a 7 (111_2). O primeiro registrador, comumente chamado de registrador de expoentes, armazena valores de x no intervalo $0 \leq x < Q$, enquanto o segundo registrador, denominado registrador alvo ou de resultados, contém os valores calculados pela função de exponenciação modular (Eq. 3.3) aplicada aos valores de x.

O parâmetro Q define o limite superior do domínio de x e é escolhido de modo que $Q \leq N^2$. A escolha desse limite assegura que o espaço de busca possua tamanho suficiente para que o período r possa ser identificado com precisão após a aplicação da Transformada Quântica de Fourier. Em termos práticos, Q garante que a fração s/r, estimada a partir da medição quântica, consiga reconstruir corretamente o período r por meio do algoritmo de frações contínuas (presente no pós-processamento).

O procedimento inicia-se aplicando a porta *Hadamard* (Eq. 2.10) a todos os *qubits* do primeiro registrador. Essa operação transforma o estado inicial $|0\rangle^{\otimes n}$, em que todos os *qubits* estão em zero, em uma superposição uniforme de todos os estados base possíveis. Em outras palavras, o registrador passa a representar simultaneamente todos os valores

binários de x no intervalo $0 \leq x < Q$, cada um com a mesma amplitude de probabilidade. O estado resultante pode ser representado pela Equação 3.4.

$$|\psi_1\rangle = \frac{1}{\sqrt{Q}} \sum_{x=0}^{Q-1} |x\rangle \tag{3.4}$$

Em seguida, aplica-se o circuito de exponenciação modular controlada, responsável por calcular $f(x) = g^x \bmod N$ para cada x presente na superposição do primeiro registrador, armazenando os no segundo registrador. Essa operação, realizada de forma unitária e simultânea sobre todos os componentes da superposição, gera um entrelaçamento entre os registradores. O estado combinado do sistema pode ser expresso como na Equação 3.5.

$$|\psi_2\rangle = \frac{1}{\sqrt{Q}} \sum_{x=0}^{Q-1} |x\rangle\,|f(x)\rangle \tag{3.5}$$

Como a função de exponenciação modular é periódica com período r, múltiplos valores de x levam ao mesmo resultado de $f(x)$. Ao medir o segundo registrador e obter um valor específico de $f(x_0)$, o sistema colapsa para um subespaço em que o primeiro registrador mantém apenas os estados compatíveis com aquele resultado, isto é, os valores de x tais que $f(x) = f(x_0)$. Assim, o primeiro registrador permanece em superposição dos estados da forma $x_0, x_0 + r, x_0 + 2r, \ldots$, refletindo diretamente a periodicidade da função.

Para extrair r do primeiro registrador, que contém os múltiplos de r, aplica-se a ele a Transformada Quântica de Fourier (em inglês: *Quantum Fourier Transform* - QFT). A transformação realça a estrutura periódica dos valores armazenados e permite estimar r com alta precisão. O resultado da medição final fornece um valor relacionado a r, que pode ser refinado por meio do algoritmo de frações contínuas para determinar o valor exato do período e, então, ser usado para encontrar os fatores.

3.1.3 Encontrar os Fatores Desejados

Após determinar o período da função de exponenciação modular (Eq. 3.3) na etapa anterior, resta agora utilizar esse valor para encontrar os fatores primos de N. A relação matemática apresentada na primeira etapa (Eq. 3.1), que envolve o palpite inicial g e o período r encontrado na segunda parte do algoritmo, pode ser reescrita de forma que N seja representado como o produto de dois termos contendo g e r (Eq. 3.6).

$$\begin{aligned} g^r &\equiv 1 \pmod{N} \\ g^r - 1 &\equiv 0 \pmod{N} \\ (g^{r/2} - 1)(g^{r/2} + 1) &\equiv 0 \pmod{N} \end{aligned} \tag{3.6}$$

Com base na equação acima, N deve necessariamente dividir o produto dos dois termos do lado esquerdo, ou seja, N compartilha pelo menos um fator com $(g^{r/2}-1)$ ou com $(g^{r/2}+1)$. Portanto, se r for um número par, então os fatores de N podem ser obtidos calculando o MDC entre N e esses termos (Eq. 3.7). Por outro lado, se r for um número ímpar, deve-se escolher um novo valor para o palpite g e reiniciar o algoritmo, pois $g^{r/2}$ não resultaria em um valor inteiro (Alg. 1, linhas 23 a 36).

$$\begin{aligned} &MDC(g^{r/2}-1, N) \\ &MDC(g^{r/2}+1, N) \end{aligned} \tag{3.7}$$

Se pelo menos um dos cálculos retornar um fator p não trivial $(1 < p < N)$, então um fator primo de N foi encontrado. Caso contrário, um novo valor de g deve ser escolhido e o algoritmo repetido. Felizmente, a teoria dos números garante que, para a maioria das escolhas de g, o método funciona corretamente com alta probabilidade (SHOR, 1994).

3.2 Implementação do Algoritmo de Shor com *Cirq*

O *Cirq* (CIRQ, 2018) é um *framework* de código aberto desenvolvido pelo Google, voltado especificamente para aplicações em Computação Quântica. Em sua documentação oficial, o *Cirq* é descrito como uma biblioteca para a linguagem de programação *Python*, projetada para facilitar a escrita, manipulação, otimização e execução de circuitos quânticos em simuladores ou em dispositivos quânticos reais. Lançado em julho de 2018 como parte das iniciativas do Google para fomentar avanços na área, o projeto é distribuído sob a licença Apache 2.0, o que permite seu uso em contextos acadêmicos e comerciais.

Além disso, o *Cirq* recebe atualizações de forma regular e contínua, refletindo seu caráter de projeto ativo em desenvolvimento. O Google mantém um plano de evolução com novas versões sendo publicadas periodicamente, incluindo correções, melhorias de desempenho e a introdução de recursos que expandem o suporte a diferentes dispositivos quânticos. Essa dinâmica de manutenção demonstra o compromisso da comunidade e da própria empresa com a maturidade da ferramenta, assegurando que ela permaneça relevante tanto em pesquisas acadêmicas quanto em aplicações práticas emergentes.

Como se trata de uma biblioteca *Python*, a instalação do *Cirq* é bem simples e compatível com os principais sistemas operacionais, como *Windows*, *MacOS* e *Linux*. O processo é realizado por meio de gerenciadores de pacotes, como o *pip*, e conta com ampla documentação, incluindo instruções detalhadas para instalação e resolução de eventuais problemas técnicos (CIRQ, 2024a).

Diferentemente de outras plataformas, o *Cirq* não possui uma interface gráfica nativa. A principal forma de interação com a ferramenta ocorre por meio de programas

escritos em *Python*, geralmente executados em linha de comando. Entretanto, o *Cirq* possui integração nativa com o *Jupyter Notebook* (KLUYVER et al., 2016). Esse ambiente de computação interativa facilita a visualização de circuitos quânticos e permite a combinação fluida de trechos de código e explicações.

No contexto deste trabalho, a biblioteca *Cirq* é utilizada exclusivamente nos trechos relacionados à Computação Quântica da implementação do algoritmo de Shor. As demais etapas, de natureza clássica, são desenvolvidas eficientemente com recursos convencionais da linguagem *Python*. Tal abordagem híbrida reflete a própria natureza do algoritmo de Shor, que combina elementos clássicos com quânticos. Importante destacar que, embora o *Cirq* permita a execução de circuitos em dispositivos quânticos reais, neste trabalho as operações quânticas foram executadas por meio de emulações em *hardware* clássico. Ou seja, os efeitos quânticos foram simulados, possibilitando a análise funcional do algoritmo sem a necessidade de acesso a um processador quântico físico.

Vale ressaltar que na simulação em *hardware* clássico é possível utilizar milhares de *qubits*, sendo isso limitado apenas pela memória e capacidade computacional do sistema executante. Para execuções em *hardware* quântico real, a integração principal se dá com os processadores quânticos (em inglês: *Quantum Processing Unit* - QPU) do Google, como o *Sycamore* (ARUTE et al., 2019), que possuem até 54 *qubits*. Porém, a utilização desse *hardware* quântico é restrita, ocorrendo através de plataformas de computação em nuvem e depende de acesso concedido por meio de colaborações ou programas de pesquisa.

Adicionalmente, tanto na parte clássica quanto quântica da implementação, são empregadas práticas e estruturas comuns da programação moderna, como laços de repetição, comandos condicionais, orientação a objetos, definição de classes, uso de abstrações e modularização do código. Tais aspectos tornam o desenvolvimento mais organizado, favorecem a reutilização de componentes e facilitam a manutenção do trabalho como um todo. Assim, mesmo as seções que exploram conceitos da Computação Quântica permanecem acessíveis a quem já possui familiaridade com paradigmas de programação tradicionais.

A comunidade de usuários do *Cirq* apesar de ativa, não é tão grande quanto a outras, como a do *Qiskit*. Porém, a documentação oficial do *Cirq* é vasta, clara, com inúmeros exemplos e tutoriais, inclusive um exemplo completo de implementação do algoritmo de Shor (CIRQ, 2024b), o qual é usado como base para os experimentos apresentados nesta seção. O exemplo consiste em um programa *Python* que recebe como entrada o número inteiro N a ser fatorado e como saída um fator não trivial de N, quando encontrado. As etapas implementadas seguem de forma geral o pseudocódigo apresentado no Algoritmo 1, cujos detalhes serão explorados a seguir.

3.2.1 Pré-processamento com *Cirq*

No exemplo oficial de implementação do algoritmo de Shor utilizando o *Cirq* (CIRQ, 2024b), são incluídas etapas iniciais de pré-processamento com o objetivo de otimizar a execução do algoritmo. Estas validações consistem em três verificações fundamentais: i) se o número N a ser fatorado é par, ii) se N é um número primo e iii) se N é uma potência de um número primo. Essas etapas correspondem às linhas 3 a 9 do Algoritmo 1, sendo executadas de forma inteiramente clássica, sem a utilização do *Cirq* e de circuitos quânticos, apenas com recursos convencionais da linguagem *Python*.

O trecho de código apresentado no Algoritmo 2 implementa essas verificações de forma clara e objetiva. Inicialmente, define-se a função `checa_potencia_prima` que verifica se N pode ser representado como a^k, sendo a e k inteiros e $k \geq 2$ (Alg. 2, linhas 2 a 11). Para isso, percorre valores possíveis de k e calcula a raiz k-ésima de N, testando se a potência de seu piso ou teto corresponde a N. Caso algum valor satisfaça essa condição, ele é retornado como fator potencial.

Em seguida, o código realiza a verificação da paridade de N com o operador módulo (linha 14). Caso N seja divisível por 2, este já é retornado como fator não trivial. Depois, utiliza-se a função `isprime` da biblioteca *SymPy* para determinar se N é primo (l. 18). Caso positivo, não há fatores não triviais a serem encontrados, então o algoritmo é interrompido com a devida notificação. Por fim, a função `checa_potencia_prima` é chamada e, caso encontre um valor válido, retorna-o como solução (linhas 23 a 25).

Algoritmo 2 Pré-processamento com *Cirq*

```
1  #Retorna fator não trivial de N se N é uma potência prima, senão retorna None.
2  def checa_potencia_prima(N: int) -> Optional[int]:
3      for k in range(2, math.floor(math.log2(N)) + 1):
4          c = math.pow(n, 1 / k)
5          c1 = math.floor(c)
6          if c1**k == N:
7              return c1
8          c2 = math.ceil(c)
9          if c2**k == N:
10             return c2
11     return None
12
13 # Se N é par, 2 é um fator não trivial de N.
14 if N % 2 == 0:
15     return 2
16
17 # Se N é primo, não tem fatores não triviais.
18 if sympy.isprime(N):
19     print("N primo!")
20     return None
21
22 # Se N é uma potência prima, pode-se achar um fator não trivial eficientemente.
23 c = checa_potencia_prima(N)
24 if c is not None:
25     return c
```

3.2.2 Etapa Quântica com *Cirq*

A implementação da etapa quântica (Alg. 1, linhas 16 a 22) consiste na criação de um circuito capaz de computar a função de exponenciação modular $f(x) = g^x \bmod N$ e extrair seu período por meio da QFT. Essa é a parte fundamental do algoritmo de Shor, pois permite explorar a periodicidade dessa função, possibilitando a fatoração eficiente de N. Lembrando que N não é um número primo, caso contrário o código haveria se encerrado conforme visto nas linhas 18 a 20 do Algoritmo 2.

A biblioteca *Cirq* oferece ferramentas de alto nível para a construção de circuitos quânticos. Entre seus recursos destacam-se as portas quânticas básicas, como *Hadamard* e CNOT, bem como portas mais complexas, como a QFT, operadores aritméticos e recursos para a construção de portas personalizadas. No caso específico da exponenciação modular, é necessário implementar, com *Cirq*, uma porta quântica customizada que represente essa função matemática como uma transformação unitária aplicada sobre *qubits* (Eq. 3.5).

Para isso, define-se uma nova classe denominada `ModularExp` (Alg. 3), que herda atributos e métodos da classe `cirq.ArithmeticGate` - uma abstração fornecida pelo *Cirq* que auxilia na criação de portas quânticas para representar operações aritméticas. Nesta classe, são definidos alguns métodos abstratos herdados, como o método `apply`, responsável por descrever a operação que será executada sobre os *qubits*, como também o método `__init__` que atua como o construtor da classe.

Algoritmo 3 Definição da porta quântica ModularExp com *Cirq*

```
1  class ModularExp(cirq.ArithmeticGate):
2      #Exponenciação modular quântica.
3      def __init__(
4          self,
5          alvo: Sequence[int], # Registrador
6          expoente: Union[int, Sequence[int]], # Registrador
7          base: int, # Constante
8          modulo: int # Constante
9      ) -> None:
10         if len(alvo) < modulo.bit_length():
11             raise ValueError(
12                 f'Registrador alvo com {len(alvo)} qubits pequeno demais para modulo'
13                 f' {modulo}'
14             )
15         self.alvo = alvo
16         self.expoente = expoente
17         self.base = base
18         self.modulo = modulo
19
20     def apply(self, *valores_registradores: int) -> int:
21         #Aplica exponenciação modular aos registradores.
22
23         assert len(valores_registradores) == 4
24         alvo, expoente, base, modulo = valores_registradores
25         if alvo >= modulo:
26             return alvo
27         return (alvo * base**expoente) % modulo
28
29     ...
```

O método `__init__` (Alg. 3, linhas 3 a 18), construtor da classe `ModularExp`, é responsável por inicializar os atributos necessários para configurar corretamente a porta de exponenciação modular desejada. Esse método recebe quatro parâmetros principais: i) `alvo`, representando o registrador onde é armazenado o resultado da exponenciação; ii) `expoente`, registrador que contém o expoente da função exponencial; iii) `base`, constante inteira que representa a base da exponenciação (isto é, o valor do palpite g); iv) `modulo`, constante inteira que representa N, o valor a ser fatorado.

Vale ressaltar que a palavra-chave `self`, utilizada como primeiro parâmetro tanto no método `__init__` quanto no método `apply`, não está diretamente relacionada à lógica da operação de exponenciação modular. Trata-se, na verdade, de uma convenção da linguagem *Python* no contexto de programação orientada a objetos. Sua função é fazer referência à instância atual da classe - neste caso, à própria instância da classe `ModularExp` (Alg. 3), `self` está apenas associando os parâmetros recebidos aos atributos internos do objeto, preparando a estrutura necessária para sua aplicação posterior em circuitos.

No escopo do *Cirq*, os parâmetros `alvo` e `expoente` são tratados como registradores quânticos, definidos como sequências ordenadas de *qubits*. Na prática, esses registradores são representados por listas ou sequências (*arrays*) de inteiros, em que cada inteiro representa um *qubit* específico no circuito. Por exemplo, a sequência $[q_0, q_1, q_2]$ indica um registrador de 3 *qubits*, formado pelos *qubits* q_0, q_1 e q_2.

Além de armazenar os valores recebidos, o método construtor realiza uma verificação de consistência (Alg. 3, linhas 10 a 14). Verifica-se se o número de *qubits* no registrador `alvo` é suficiente para representar valores até o `modulo` especificado. Caso contrário, é lançada uma exceção, garantindo que os cálculos não resultem em perda de informação. Essa verificação protege a integridade da operação de exponenciação modular durante a execução em um circuito quântico.

Uma vez definidos os atributos da classe, o funcionamento da porta é descrito por meio do método `apply` (Alg. 3, linhas 20 a 27). Esse método representa o núcleo funcional da transformação aritmética que será aplicada aos valores armazenados nos registradores durante a execução do circuito. Ele é automaticamente invocado pelo *Cirq* quando a porta quântica `ModularExp` é aplicada a um conjunto de *qubits*.

A assinatura do método utiliza a notação `*valores_registradores`, característica da linguagem *Python*, para indicar que um número variável de argumentos inteiros será aceito. No caso específico desta implementação, espera-se exatamente quatro valores inteiros (verificados na linha 23), em ordem: i) o valor atual do registrador `alvo`, ii) o valor do registrador `expoente`, iii) a `base` da exponenciação e iv) o `modulo`. Embora base e módulo tenham sido definidos no construtor, eles são novamente passados ao método para que este seja genérico e compatível com as regras internas de execução do *Cirq*.

A lógica implementada no corpo do método `apply` é simples, porém fundamental. Primeiramente, verifica-se se o valor atual do registrador `alvo` é maior ou igual ao `modulo`; se for, o valor é retornado inalterado. Caso contrário, realiza-se a operação de exponenciação modular conforme a seguinte fórmula: $(\texttt{alvo} * \texttt{base}^{\texttt{expoente}}) \bmod \texttt{modulo}$. O resultado dessa operação representa o novo estado do registrador `alvo` após a aplicação da porta. Apesar dessa operação ser executada em ambiente clássico (durante a simulação), o *Cirq* a traduz em uma transformação unitária correspondente, garantindo fidelidade ao comportamento de um circuito quântico real.

Uma vez definida a porta quântica de exponenciação modular, torna-se possível utilizá-la na construção do circuito responsável por extrair o período da função $f(x) = g^x \bmod N$. Para isso, define-se uma função denominada `criar_circuito_busca_periodo`, que recebe como parâmetros dois inteiros: g, representando o palpite inicial, e N, o inteiro a ser fatorado. Como retorno, a função entrega um circuito quântico configurado com as operações necessárias para a realização da etapa quântica do Algoritmo de Shor (Alg. 4). Essa etapa é semelhante ao proposto no Algoritmo 1 nas linhas 16 a 22.

Algoritmo 4 Definição de circuito do algoritmo de Shor com *Cirq*

```
1   # Retorna circuito que computa a ordem de g modulo N.
2   def criar_circuito_busca_periodo(g: int, N: int) -> cirq.Circuit:
3       L = N.bit_length()
4       # Registrador alvo com L qubits
5       alvo = cirq.LineQubit.range(L)
6       # Registrador expoente com 2L+3 qubits
7       expoente = cirq.LineQubit.range(L, 3 * L + 3)
8
9       # Cria uma porta ModularExp definida anteriormente
10      mod_exp = ModularExp([2] * L, [2] * (2 * L + 3), g, N)
11
12      return cirq.Circuit(
13          cirq.X(alvo[L - 1]),
14          cirq.H.on_each(*expoente),
15          mod_exp.on(*alvo, *expoente),
16          cirq.qft(*expoente, inverse=True),
17          cirq.measure(*expoente, key='expoente'),
18      )
```

Inicialmente, a função calcula o número de *bits* necessários para representar N, armazenando o valor em L (Alg. 4, linha 3). Com esse valor, dois registradores são criados: o registrador `alvo`, com L *qubits*, e o registrador `expoente`, com $2L + 3$ *qubits*. Esses registradores são formados utilizando o método `cirq.LineQubit.range`, que cria listas sequenciais de *qubits* - abordagem comum em implementações com *Cirq*.

Na linha 10 (Alg. 4), é instanciada a porta quântica `ModularExp`, responsável por computar a função de exponenciação modular. Os dois primeiros parâmetros - `[2] * L` e `[2] * (2 * L + 3)` - indicam a dimensão dos registradores alvo e expoente, respectivamente. Esses valores não definem exatamente quais *qubits* serão utilizados, mas a quantidade de *qubits* esperada para cada registrador na operação. Já os parâmetros finais

(g e N) correspondem ao palpite e ao inteiro a ser fatorado.

Em seguida, constrói-se o circuito quântico por meio do construtor `cirq.Circuit` (Alg. 4, linhas 12 a 17), adicionando, em ordem, as seguintes operações: i) inicialização do registrador `alvo`, aplica-se a porta X ao *qubit* mais significativo do registrador alvo, colocando-o no estado $|1\rangle$ e evitando que o registrador inicie em 0, o que anularia o resultado da exponenciação modular; ii) criação de superposição no registrador `expoente`, aplicando a porta *Hadamard* (H) a todos os *qubits* do expoente; iii) aplicação da porta de exponenciação modular aos dois registradores; iv) aplicação da Transformada Quântica Inversa de Fourier (QFT^{-1}) ao registrador expoente, visando extrair o período da função exponencial; v) medição do registrador expoente ao fim do circuito.

Nesse circuito, os *qubits* do registrador alvo são denotados como $t_0, t_1, ..., t_{L-1}$ enquanto os *qubits* do expoente são representados por $e_0, e_1, ..., e_{2L+2}$, totalizando $3L + 3$ *qubits*. O circuito resultante implementa de maneira eficiente a etapa quântica do algoritmo de Shor, permitindo que a periodicidade da função $f(x)$ seja explorada.

Ao final da construção do circuito quântico, é possível visualizar seu funcionamento prático por meio de um exemplo específico. Considerando os parâmetros $g = 5$ e $N = 6$, em que se deseja fatorar o número 6 utilizando o palpite 5. A função `criar_circuito_busca_periodo`, apresentada anteriormente, pode ser utilizada com esses valores para gerar o circuito correspondente.

O resultado, impresso diretamente com o comando `print(circuito)`, revela todas as portas quânticas aplicadas ao longo do tempo sobre os qubits do circuito. O Algoritmo 5 mostra essa estrutura de forma textual, como fornecida pela simulação com o *Cirq*.

Algoritmo 5 Exemplo de circuito construído

```
circuito = criar_circuito_busca_periodo(g=5, N=6)
print(circuito)

#Resultado
#0: --------ModularExp(t*5**e % 6)----------------------------
#           |
#1: --------t1-----------------------------------------------
#           |
#2: ----X---t2-----------------------------------------------
#           |
#3: ----H---e0-----------------------qft^-1---M('expoente')---
#           |                       |        |
#4: ----H---e1-----------------------#2-------M---------------
#           |                       |        |
#5: ----H---e2-----------------------#3-------M---------------
#           |                       |        |
#6: ----H---e3-----------------------#4-------M---------------
#           |                       |        |
#7: ----H---e4-----------------------#5-------M---------------
#           |                       |        |
#8: ----H---e5-----------------------#6-------M---------------
#           |                       |        |
#9: ----H---e6-----------------------#7-------M---------------
#           |                       |        |
#10: ---H---e7-----------------------#8-------M---------------
#           |                       |        |
#11: ---H---e8-----------------------#9-------M---------------
```

No exemplo acima, o circuito é composto pelos registradores `alvo`, contando 3 *qubits* (de t_0 `a` t_2), e expoente, contendo 9 *qubits* (de e_0 a e_8). Observa-se a aplicação de uma porta `X` ao último *qubit* do registrador `alvo`, seguida da porta *Hadamard* aplicada a todos os *qubits* do `expoente`. A operação principal do circuito, a exponenciação modular $t * g^e \bmod N$, é representada pela porta `ModularExp`, aplicada sobre os dois registradores. Posteriormente, a Transformada Quântica de Fourier inversa é executada sobre o registrador `expoente`, finalizando com a medição de todos os seus *qubits*. Cada linha no diagrama representa a evolução temporal de um *qubit*, evidenciando o fluxo do circuito construído.

Com esse circuito definido, é possível simular sua execução utilizando o simulador clássico do *Cirq*. É importante enfatizar que essa simulação não envolve *qubits* físicos reais: em vez disso, comportamentos quânticos como superposição e emaranhamento são emulados por algoritmos rodando em *hardware* clássico. Embora essa abordagem seja extremamente útil para fins educacionais e experimentais, ela apresenta limitações de escalabilidade, dado o custo computacional crescente relativo ao número de *qubits*.

Ao simular o circuito, o resultado de cada execução é uma cadeia de bits (*bitstring*) que representa o estado final medido dos *qubits* do registrador expoente. Essas *bitstrings* são posteriormente convertidas para inteiros em base decimal e utilizadas na etapa seguinte do algoritmo, em que se tenta reconstruir o período da função por meio de frações contínuas, viabilizando assim a obtenção dos fatores não triviais de N. O Algoritmo 6 mostra o processo completo de simulação do circuito e exibição de seus resultados.

Algoritmo 6 Resultados após simulação do circuito

```
1   circuito = criar_circuito_busca_periodo(g=5, N=6)
2   resultado = cirq.sample(circuito, repetitions=3)
3
4   print(resultado)
5   print(resultado.data)
6
7   """Resultado bruto do registrador "expoente" após medição"""
8   expoente=110, 000, 000, 000, 000, 000, 000, 000, 000
9
10  """Resultado final de cada repetição"""
11     expoente
12  0       256
13  1       256
14  2         0
```

No exemplo acima (Alg. 6), o circuito foi simulado três vezes (`repetitions=3`). A saída bruta exibe as *bitstrings* obtidas nas medições, enquanto a tabela final apresenta os valores inteiros equivalentes, já decodificados pelo *Cirq*. Cada uma das triplas resultantes presentes no registrador `expoente` (Alg. 6, linha 8) corresponde ao estado final de um *qubit* - de e_8 a e_0, da esquerda para a direita - ao fim das três repetições. Ou seja, a primeira tripla `110` expressa que o *qubit* e_8, pertencente ao registrador `expoente`, foi medido no estado 1, 1 e 0, na primeira, segunda e terceira repetição, respectivamente.

Sendo assim, a *bitstring* resultante da primeira repetição é composta pelo primeiro inteiro de cada tripla, onde a primeira tripla representa o *qubit* mais significativo, e o último, o menos significativo. Note que o valor `256`, obtido em duas das três repetições, corresponde à *bitstring* `100000000`, que representa a posição do *qubit* `e8` em estado `1`, enquanto todos os outros *qubits* estão em `0`. Já a terceira repetição resultou na *bitstring* `000000000`, que corresponde ao valor `0` (Alg. 6, linha 14).

A presença de valores distintos entre as repetições é esperada e decorre da natureza probabilística das medições quânticas. Esses valores finais, como o `256` mostrado no exemplo, são então utilizados na etapa clássica subsequente, na qual se aplica o algoritmo de frações contínuas com o objetivo de estimar a relação s/r onde s é o valor medido e r o período buscado. A partir disso, tenta-se deduzir r e, se encontrado corretamente, calcular os fatores não triviais de N.

3.2.3 Pós-processamento com *Cirq*

A partir dos valores medidos no registrador `expoente` durante a etapa de simulação do circuito quântico, o principal objetivo agora é extrair uma fração racional da forma s/r, na qual r representa o período da função $f(x) = g^x \bmod N$ (Eq. 3.3), e s é um número inteiro tal que $0 \leq s < r$. Essa etapa, equivalente às linhas 23 a 36 do Algoritmo 1, é inteiramente realizável em um ambiente com *hardware* clássico e é fundamental para que se possa recuperar os fatores não triviais de N.

Para isso, parte-se do valor decimal obtido a partir da medição do registrador `expoente`. Em simulações com múltiplas repetições, geralmente considera-se o resultado da primeira amostra. Esse valor medido é então dividido pela base $B = 2^m$, onde m representa o número de *qubits* do registrador `expoente`. No exemplo anterior (Alg. 6), o valor medido foi 256, e o registrador possuía 9 *qubits*, ou seja, $B = 2^9 = 512$. Portanto, o valor fracionário obtido será $256/512 = 0.5$.

Esse número é interpretado como uma aproximação da fração s/r, que representa a chamada "fase própria" (*eigenphase*) associada ao operador quântico aplicado no circuito. O passo seguinte consiste em empregar o método das frações contínuas para tentar recuperar uma aproximação racional de s/r, isto é, valores inteiros s e r que satisfazem aproximadamente a equação $s/r \approx 0.5$. A biblioteca `fractions` da linguagem *Python* permite obter essa fração de forma direta, com a função `limit_denominator(N)`, que busca denominadores r razoáveis, limitando-os ao valor de N.

O Algoritmo 7 mostra a implementação dessa etapa. Primeiramente, extrai-se o valor decimal medido do registrador `expoente` e a quantidade de *qubits* envolvidos na medição. Em seguida, calcula-se a *eigenphase*, que é justamente a razão entre o valor medido e a base 2^m. Essa razão é convertida em uma fração reduzida, e, se seu numerador for igual a zero, a tentativa de reconstrução do período falhou. Caso contrário, extrai-se o denominador da fração obtida e o considera como um candidato ao período r. Por fim, verifica-se a validade de r, testando se $g^r \bmod N = 1$. Se a condição for satisfeita, o valor de r é retornado.

Algoritmo 7 Extração do período após medir circuito com *Cirq*

```
1  def processa_saida_circuito(resultado: cirq.Result, g: int, N: int) -> Optional[int]:
2      expoente_inteiro = resultado.data["expoente"][0]
3      num_bits_expoente = resultado.measurements["expoente"].shape[1]
4      eigenphase = float(expoente_inteiro / 2**num_bits_expoente)
5
6      # Frações contínuas para aproximar frac = s / r
7      frac = fractions.Fraction.from_float(eigenphase).limit_denominator(N)
8
9      if frac.numerator == 0:
10         return None
11
12     r = frac.denominator
13     if g**r % N != 1:
14         return None
15     return r
```

Com o valor do período r, a última etapa do algoritmo consiste em utilizar a Equação 3.7 para calcular os possíveis fatores de N com base nos valores $MDC(g^{r/2} \pm 1, N)$. Se pelo menos um dos dois resultados retornar um valor p tal que $1 < p < N$, então foi encontrado um fator não trivial de N, e o algoritmo pode ser encerrado com sucesso.

3.3 Implementação do Algoritmo de Shor com *Qiskit*

Disponibilizado publicamente no ano de 2017, o *Qiskit* é um conjunto de ferramentas de código aberto (licença Apache 2.0) voltado para o desenvolvimento de algoritmos e aplicações em Computação Quântica (CROSS, 2018). Escrito para a linguagem de programação *Python*, oferece ferramentas para a criação, simulação e execução de circuitos quânticos tanto em simuladores clássicos quanto em processadores quânticos reais disponibilizados pela própria fundadora, a companhia norte-americana IBM, via nuvem. Desde seu lançamento, o *Qiskit* consolidou-se como uma das plataformas mais populares para ensino, pesquisa e prototipagem de algoritmos quânticos, sustentado por uma comunidade ativa e uma documentação oficial extensa, com tutoriais e exemplos práticos.

A instalação do *Qiskit* é simples e multiplataforma, suportando sistemas operacionais como *Windows*, *macOS* e *Linux*. A forma mais comum de instalação é via *pip* (***p**ackage **i**nstaller for **P**ython*), o gerenciador de pacotes do *Python*, e a documentação oficial do projeto fornece instruções detalhadas para configurações específicas, resolução de erros comuns e integração com simuladores locais e remotos (IBM Quantum, 2025). Além disso, o *Qiskit* oferece acesso gratuito a uma série de dispositivos quânticos reais da IBM, mediante a criação de uma conta no serviço IBM *Quantum Experience*.

Ao contrário de bibliotecas que se restringem a prover um simulador, o *Qiskit* pode ser classificado tanto como uma biblioteca quanto como um conjunto de ferramentas (*toolkit*). Enquanto biblioteca, fornece classes e métodos que permitem a construção modular de circuitos quânticos; como *toolkit*, abrange todo o ciclo de desenvolvimento de algoritmos quânticos, incluindo compilação, tradução para diferentes dispositivos, execuções em simuladores ou máquinas reais, análise de resultados e até integração com técnicas de otimização. Tudo isso amplia a versatilidade da ferramenta e justifica seu protagonismo no ecossistema da Computação Quântica.

Outro ponto relevante é a documentação do *Qiskit*. Além de manuais técnicos detalhados, inclui exemplos práticos, tutoriais interativos, um livro-texto oficial amplamente utilizado em cursos introdutórios e avançados de Computação Quântica. A comunidade de desenvolvedores e usuários é ativa, com frequente contribuição em fóruns, repositórios e canais de suporte, favorecendo a rápida resolução de dúvidas e evolução da plataforma. Tais características também são reforçadas pela regularidade dos lançamentos. O *Qiskit* possui um plano de desenvolvimento (*roadmap*) público e detalhado, no qual são divulgadas as funcionalidades em desenvolvimento e atualizações previstas.

No entanto, também é importante citar limitações práticas que afetam o uso do *Qiskit*. Embora a execução de algoritmos em simuladores clássicos como o `AerSimulator` permita testar implementações com até dezenas de *qubits*, o custo computacional cresce exponencialmente, fazendo com que seja inviável simular grandes instâncias do algoritmo

de Shor. Já no caso de execução em QPUs reais disponibilizadas pela IBM, existem restrições quanto ao número de *qubits* disponíveis (poucas dezenas).

No contexto deste trabalho, o *Qiskit* é utilizado para reproduzir o algoritmo de Shor com base nos mesmos princípios da implementação feita em *Cirq*. A implementação segue uma abordagem híbrida, com partes clássicas desenvolvidas em *Python* puro e a parte quântica expressa por meio dos circuitos quânticos construídos com o *Qiskit*. Foram utilizadas ferramentas como `QuantumCircuit`, `AerSimulator`, além de operadores aritméticos compostos personalizados para representar as etapas de exponenciação modular e o uso de registradores separados para entrada, saída e auxílio.

Um aspecto importante é que, apesar do *Qiskit* oferecer suporte à execução em QPUs reais, os experimentos deste trabalho foram realizados em modo simulado, utilizando o simulador *AerSimulator*. Isso permitiu avaliar o comportamento e os resultados esperados do algoritmo sem depender da disponibilidade de *qubits* nas QPUs reais.

Por fim, embora a versão oficial do *Qiskit* não inclua um exemplo completo e pronto do algoritmo de Shor na biblioteca principal, sua estrutura modular permite que o algoritmo seja implementado de forma direta, e este trabalho propõe uma versão adaptada para o fatoramento de pequenos inteiros. Essa implementação utiliza circuitos de exponenciação modular e QFT, alinhando-se ao pseudocódigo apresentado no Algoritmo 1 e permitindo, assim, uma comparação direta com a implementação realizada em *Cirq*.

3.3.1 Pré-processamento com *Qiskit*

Na implementação do algoritmo de Shor com *Qiskit*, também são realizadas etapas de pré-processamento que antecedem a construção do circuito quântico. Essas etapas são inteiramente clássicas, escritas exclusivamente com recursos nativos da linguagem *Python* e têm como objetivo otimizar a execução do algoritmo, evitando o processamento quântico desnecessário em casos que possam ser resolvidos por métodos clássicos diretos.

Assim como no caso do *Cirq*, são avaliadas três condições principais: i) se o número N a ser fatorado é par, ii) se N é um número primo, e iii) se N é uma potência perfeita. Essas verificações iniciais correspondem, de forma aproximada, às linhas 3 a 15 do Algoritmo 1 e estão detalhadas no Algoritmo 8.

Primeiramente, testa-se a paridade de N com a operação módulo (linha 2, Alg. 8); se for par, o fator 2 já é retornado como uma solução válida, encerrando o processo. Na sequência, entre as linhas 6 a 8, utiliza-se a função `isprime` da biblioteca `sympy` para determinar se N é primo; sendo N primo, não possui fatores não triviais, e o algoritmo é encerrado. Em seguida, nas linhas 11 a 14, busca-se identificar se N pode ser escrito como p^k, sendo p e k inteiros e $k \geq 2$. Essa verificação é feita calculando a raiz k-ésima de N; em cada iteração arredonda-se o resultado e verifica-se se a potência obtida corresponde

a N. Caso positivo, o valor p é retornado como fator não trivial.

Algoritmo 8 Validações iniciais com *Qiskit*

```
1  # Se N é par, 2 é um fator não trivial de N.
2  if N % 2 == 0:
3      return 2
4
5  # Se N é primo, não tem fatores não triviais.
6  if sympy.isprime(N):
7      print("N primo!")
8      return None
9
10 # Se N é uma potência perfeita, pode-se achar um fator não trivial eficientemente.
11 for k in range(2, int(log2(N)) + 1):
12     p = round(N ** (1/k))
13     if p ** k == N:
14         return p
15
16 if g is None:
17     while True:
18         g = random.randint(2, N - 1)
19         if gcd(g, N) == 1:
20             break
21 else:
22     d = gcd(g, N)
23     if d > 1:
24         return g #Fator encontrado por acaso
```

Por fim, o trecho entre as linhas 16 e 24 (Alg. 8) trata da escolha da base g, que será utilizada posteriormente na etapa quântica do algoritmo. Se g não tiver sido previamente definida, o código seleciona aleatoriamente um valor no intervalo $[2, N-1]$ que seja coprimo a N, ou seja, cujo máximo divisor comum (função `gcd`) com N seja igual a 1. Se, ao contrário, g já tiver sido definida, realiza-se a verificação de `gcd`(g, N); se o resultado for maior que 1, significa que foi encontrado um fator não trivial de N de forma puramente clássica, dispensando a execução da etapa quântica.

3.3.2 Etapa quântica com *Qiskit*

Após a conclusão das verificações clássicas iniciais e a escolha da base g, coprima a N, inicia-se a etapa quântica responsável por estimar o período r da função $f(x) = g^x \bmod N$. Essa fase, análoga às linhas 16 a 22 do Algoritmo 1, é onde ocorre a principal vantagem quântica do algoritmo de Shor. Nela, constrói-se um circuito que prepara uma superposição de estados no registrador de contagem, aplica exponenciação modular controlada sobre um registrador de trabalho e, por meio da Transformada Quântica Inversa de Fourier (QFT^{-1}), converte a informação de fase acumulada em um padrão de interferência que, ao ser medido, fornece múltiplos de s/r com alta, onde s é o valor medido e r o período buscado. A partir disso, tenta-se deduzir r e, se encontrado corretamente, calcular os fatores não triviais de N.

A função `criar_circuito_shor` (Alg. 9) instancia i) um registrador quântico `expoente` para armazenar o expoente, ii) um registrador quântico `alvo` para armazenar

o resultado dos cálculos da exponenciação modular durante a execução do circuito e iii) um registrador clássico `cont_classico` para armazenar os resultados das medições. O circuito e suas operações são construídas conforme o Algoritmo 9.

Algoritmo 9 Construção do circuito quântico principal com *Qiskit*

```
1  def criar_circuito_shor():
2      L = N.bit_length()
3      m = 2*L
4      expoente = QuantumRegister(m, 'expoente')
5      alvo = QuantumRegister(L, 'alvo')
6      cont_classico = ClassicalRegister(L, 'c_cont')
7
8      qc = QuantumCircuit(expoente, alvo, cont_classico)
9      # Passo 1: superposição uniforme em cont
10     qc.h(expoente)
11
12     # Passo 2: inicializa alvo em |1>
13     qc.x(alvo[0])
14
15     # Passo 3: exponenciação modular controlada
16     for i in range(m):
17         potencia = pow(2, i)
18         mult = modular_exponentiation_classical(g, potencia, N)
19         _controlled_modular_multiplication(qc, expoente[i], alvo, mult
20 )
21     # Passo 4: QFT inversa no registrador de contagem
22     circuito_QFT = QFTGate(m).inverse()
23     qc.append(circuito_QFT, expoente)
24
25     # Passo 5: medição do contador
26     qc.measure(expoente, cont_classico)
27     return qc
```

Inicialmente, no Algoritmo 9, calcula-se as variáveis L e m (linhas 2 e 3), onde L é o número de *bits* necessários para representar o inteiro N a ser fatorado, e $m = 2 * L$. Então, são criados os registradores quânticos `expoente`, com m *qubits*; `alvo`, com L *qubits*, e um registrador clássico para auxiliar nas operações, com L *bits* (linhas 4 a 6). Na sequência, cria-se um objeto do tipo `QuantumCircuit` para representar o circuito quântico, contendo os registradores recém-criados (linha 8).

Com o circuito já criado, aplica-se a porta Hadamard (linha 10) a todos os *qubits* do registrador `expoente`, gerando uma superposição uniforme de todos os estados possíveis no intervalo $[0, 2^m - 1]$. Em paralelo, o registrador `alvo` é preparado no estado $|1\rangle$ utilizando a porta X (linha 13), representando o valor inteiro 1. Em seguida, aplica-se a exponenciação modular de forma controlada (linhas 16 a 19). Para cada *qubit* do registrador `expoente`, calcula-se previamente, de maneira clássica, o valor g^{2^i} mod N. Esse valor é usado como multiplicador em uma operação de multiplicação modular controlada, que atua sobre o registrador `alvo` somente quando o *qubit* de controle correspondente está no estado $|1\rangle$. Essa sequência acumula, no registrador clássico de contagem, uma fase proporcional a x/r, onde x é o valor binário presente na superposição.

Ainda no Algoritmo 9, após a etapa de exponenciação modular, aplica-se a QFT

inversa sobre o registrador `expoente` (linhas 22 e 23). Essa transformação converte a fase armazenada nas amplitudes quânticas em um padrão de interferência que concentra alta probabilidade em valores próximos a múltiplos inteiros de s/r. A biblioteca *Qiskit* oferece a classe `QFTGate`, utilizada aqui para compor e inverter a transformada de forma eficiente. Por fim, o registrador `expoente` é medido e seus resultados são armazenados no registrador clássico correspondente (linha 26). O estado quântico do registrador `alvo` não é mais relevante após a medição e, portanto, é descartado.

O circuito gerado pela função `criar_circuito_shor` (Alg. 9) pode ser visualizado em diversos formatos, desde texto e imagens simples até diagramas interativos. A Figura 4 ilustra um exemplo construído para o caso $N = 15$. Nessa representação, o registrador `expoente` contém $m = 2 * L = 8$ *qubits*, indicados como $exp_0, exp_1, ..., exp_{m-1}$, enquanto o registrador `alvo` é composto por $L = 4$ *qubits*, nomeados $alvo_0, alvo_1, ...alvo_{L-1}$. Além disso, um registrador clássico `cont_c` armazena os resultados das medições. A imagem do circuito também evidencia a sequência de operações aplicadas aos *qubits*, em ordem, desde a porta Hadamard (H) e X, passando pela exponenciação modular controlada, QFT inversa e, por fim, a medição do registrador `expoente`.

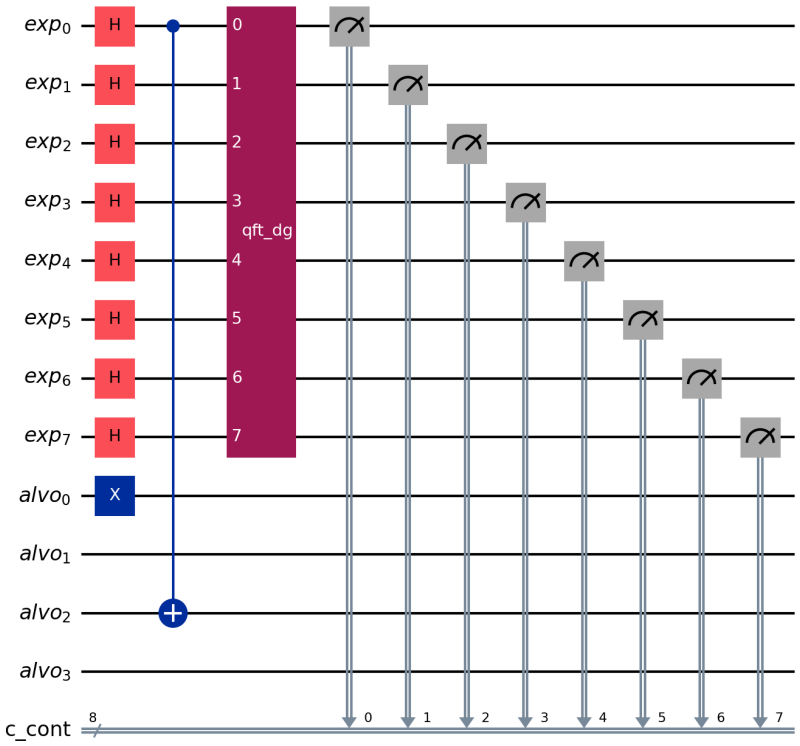

Figura 4 – Circuito do Algoritmo de Shor em *Qiskit* com $N = 15$

Uma vez completada a definição do circuito, incluindo registradores envolvidos e sequência de portas a serem aplicadas, é necessário executá-lo para extrair resultados. A função `executar_circuito_quantico`, descrita no Algoritmo 10, conclui a etapa quântica da implementação e tem como propósito traduzir o circuito quântico criado pela função

`criar_circuito_shor` (Alg. 9) para a base do simulador escolhido - no contexto deste trabalho, um simulador `QiskitAer`, simulador de alta performance com modelos de ruídos - e executar o circuito um número definido de vezes (repetições, `shots`) e, por fim, devolver as contagens de medição no formato de um dicionário, assim como esperado pela etapa subsequente de pós-processamento, que busca os fatores de N.

Algoritmo 10 Simulação e execução do circuito quântico principal com *Qiskit*

```
1  def executar_circuito_quantico(shots=1000):
2      simulador = AerSimulator()
3
4      qc = criar_circuit_shor()
5
6      # Traduz para o simulador
7      pass_manager = generate_preset_pass_manager(optimization_level=1, backend=simulador)
8      qc_traduzido = pass_manager.run(qc)
9
10     # Executa o circuito
11     exec = simulador.run(qc_traduzido, shots=shots)
12     resultado = exec.result()
13     counts = resultado.get_counts()
14
15     return counts
```

A tradução (Alg. 10, linhas 7 e 8) mapeia o circuito de alto nível para a base de portas e para as restrições do simulador, aplicando otimizações leves (por exemplo, cancelamento de portas adjacentes e reescritas equivalentes), sem alterar a semântica do circuito. Mesmo em simulação, esse passo é recomendado porque deixa explícita a forma "executável" do circuito e pode reduzir custo computacional. O número de *shots* determina a resolução probabilística das distribuições medidas: quanto maior, mais nítidos tendem a ser os picos nas *bitstrings* associadas às fases de interesse.

Por fim, o objeto `resultado` (Alg. 10, linha 12) agrega os resultados da execução do circuito, e na linha seguinte a função `get_counts` retorna um dicionário de medições cujas chaves são *bitstrings*, e cujos valores são as respectivas frequências de ocorrência. Para ilustrar, em uma simulação com $N = 15, g = 13$, em que o circuito foi executado 1000 vezes, obteve-se o seguinte dicionário de medições: `counts='11000000': 241, '01000000': 273, '00000000': 268, '10000000': 218`, representando, por exemplo, que a *bitstring* '11000000' foi obtida 241 vezes, enquanto '01000000', 273 vezes, etc.

3.3.3 Pós-processamento com *Qiskit*

A etapa de pós-processamento no algoritmo de Shor tem como objetivo extrair, a partir dos resultados obtidos nas medições no registrador `expoente`, uma aproximação de fração racional da forma s/r, em que r representa o período da função $f(x) = g^x \bmod N$, e s é um número inteiro tal que $0 \leq s < r$. Essa etapa, correspondente às linhas 23 a 36 do Algoritmo 1, é realizada inteiramente de modo clássico, não requerendo recursos

quânticos adicionais nem o uso direto de funcionalidades do *Qiskit*. Apesar de clássica, desempenha um papel essencial na obtenção dos fatores não triviais de N.

Após a execução do circuito quântico, obtém-se um conjunto de medições associadas ao registrador `expoente`, representadas como cadeias de *bits* (*bitstrings*). Cada *bitstring* é convertida para seu valor inteiro equivalente e, em seguida, normalizada pela base $B = 2^m$, sendo m o número de *qubits* do registrador `expoente`.

Por exemplo, se o valor medido for 128 e $m = 8$, então $B = 2^8 = 256$, resultando na fração $128/256 = 0.5$. Esse valor é interpretado como uma estimativa da fase quântica associada à operação de exponenciação modular no circuito, correspondendo a uma aproximação da razão s/r. A partir dessas estimativas, são selecionados candidatos ao período r, e com base nesses candidatos, busca-se determinar os fatores não triviais de N.

Na função `posprocessamento_classico` (Alg. 11) é implementado o procedimento de extração das fases. Essa função recebe como entrada o dicionário `medicoes`, no qual cada chave é uma *bitstring* obtida na medição, e cada valor indica o número de vezes que essa *bitstring* foi observada. Por exemplo, a entrada `{'00100011':231}` indica que o resultado `'00100011'` foi obtido 231 vezes.

Algoritmo 11 Extração das fases a partir dos resultados de medição

```
1  def posprocessamento_classico(medicoes):
2      fases = []
3      total_shots = sum(medicoes.values())
4
5      for bitstring, contagem in medicoes.items():
6          inteiro_medido = int(bitstring, 2)
7          fase = inteiro_medido / (2**m)
8          probabilidade = contagem / total_shots
9          fases.append((fase, probabilidade, inteiro_medido))
10
11     fases.sort(key=lambda x: x[1], reverse=True)
12
13     return extrair_fatores_desde_fases(fases)
```

Essa função percorre cada elemento do dicionário, realizando três etapas principais: i) conversão da *bitstring* para inteiro decimal (linha 6), ii) cálculo da fase estimada dividindo o valor por 2^m (linha 7), iii) determinação da probabilidade associada, dividindo a contagem daquela *bitstring* pelo total de medições (`total_shots`, linha 8).

Ainda no Algoritmo 11, esses valores são armazenados como tuplas no formato `(fase, probabilidade, inteiro_medido)` (linha 9) e organizados em ordem decrescente de probabilidade (linha 11) de modo que as fases mais frequentes (com maior probabilidade) apareçam primeiro. Ao final, a função `extrair_fatores_desde_fases` (linha 13), que é responsável pelas etapas subsequentes de obtenção dos fatores, recebe essa lista ordenada como argumento.

Na função `extrair_fatores_desde_fases` (Alg. 12), aplica-se o algoritmo de fra-

ções contínuas para aproximar cada fase como uma fração reduzida s/r. Essa conversão utiliza a classe `Fraction` da biblioteca padrão do *Python*, limitando o denominador ao valor de N por meio da função `limit_denominator(N)` (linha 6).

Se o denominador obtido for igual a 1 (linhas 8 e 9), a fase é descartada pois não produziria fatores não triviais de N. Caso contrário, o denominador é considerado um candidato ao período r. Esse valor é então testado pela função `verifica_periodo` (Alg. 13, linhas 1 a 6), que avalia se $g^r \bmod N = 1$. Se aprovado, tenta-se extrair fatores usando `extrair_fatores_desde_periodo`. Caso não haja sucesso, múltiplos de r também são testados, pois o valor inicial pode corresponder a um submúltiplo do período real.

Algoritmo 12 Extração dos fatores a partir das fases estimadas

```
1  def extrair_fatores_desde_fases(fases):
2      for fase, prob, inteiro_medido in fases[:20]:  # Usando top 20 medições
3          if fase == 0:
4              continue
5
6          frac = Fraction(phase).limit_denominator(N)
7
8          if frac.denominator == 1:
9              continue
10
11         periodo_candidato = frac.denominator
12
13         if verifica_periodo(periodo_candidato):
14             fatores = extrair_fatores_desde_periodo(periodo_candidato)
15             if fatores:
16                 return fatores
17
18         for multiplicador in [2, 4, 8]:
19             mult_periodo = periodo_candidato * multiplicador
20             if mult_periodo < N and verifica_periodo(mult_periodo):
21                 fatores = extrair_fatores_desde_periodo(mult_periodo)
22                 if fatores:
23                     return fatores
24
25     return None
```

A validação de r é feita por `verifica_periodo`, confirmando se $g^r \bmod N = 1$. Se aprovada, a função `extrair_fatores_desde_periodo` (Alg. 13) prossegue com o cálculo dos fatores por meio de $MDC(g^{r/2} \pm 1, N)$. Essa etapa exige que r seja par e que $g^{r/2} \not\equiv -1 \bmod N$, evitando resultados triviais. Uma vez obtido um período válido, o cálculo dos fatores é concluído verificando-se se o produto dos fatores encontrados reconstitui N (Alg. 13, linhas 38 a 42). Caso positivo, a fatoração é considerada bem-sucedida e o processo é encerrado, retornando a lista completa de fatores. Por outro lado, se a multiplicação dos fatores não resultar em N, a função segue e testa outros candidatos de período até que uma decomposição correta seja obtida.

Esse processo de validação garante que só períodos consistentes com a aritmética modular sejam aceitos como solução. Além disso, a checagem final do produto dos fatores, implementada na função `extrair_fatores_desde_periodo` (linhas 42 e 43), atua como

salvaguarda contra falsos positivos que poderiam surgir devido a aproximações numéricas ou submúltiplos de r. Dessa forma, o pós-processamento com *Qiskit* integra de maneira robusta os resultados obtidos da etapa quântica com os cálculos clássicos subsequentes, assegurando que a fatoração somente seja declarada concluída quando todos os critérios matemáticos forem atendidos e a consistência da decomposição de N estiver garantida.

Algoritmo 13 Extração dos fatores a partir de possíveis períodos

```
def verifica_periodo(r):
    if r <= 0:
        return False

    # Checa se g^r  1 (mod N)
    return modular_exponentiation_classical(g, r, N) == 1

def extrair_fatores_desde_periodo(r):
    # Período deve ser par
    if r % 2 != 0:
        return None

    # Calcula g^(r/2) mod N
    potencia_meio_periodo = modular_exponentiation_classical(g, r // 2, self.N)

    # Checa se g^(r/2)  -1 (mod N)
    if potencia_meio_periodo == N - 1:
        return None

    # Calcula possíveis fatores
    fator1 = gcd(potencia_meio_periodo - 1, N)
    fator2 = gcd(potencia_meio_periodo + 1, N)

    # Checa se fatores não triviais foram encontrados
    fatores = []
    for fator in [fator1, fator2]:
        if 1 < fator < N:
            fatores.append(fator)

    if fatores:
        # Termina a fatoração
        resto = N
        todos_fatores = []
        for fator in fatores:
            if resto % fator == 0:
                todos_fatores.append(fator)
                resto //= fator
        if resto > 1:
            todos_fatores.append(resto)

        # Confere fatoração
        if np.prod(todos_fatores) == N:
            return todos_fatores

    return None
```

3.4 Implementação do Algoritmo de Shor com PyQuil

O *PyQuil* também é uma biblioteca de código aberto (licença Apache 2.0) para a linguagem de programação *Python*, voltada para a programação quântica usando o *Quil* (SMITH; CURTIS; ZENG, 2016), acrônimo de ***Qu****antum* ***i****nstruction* ***l****anguage* - uma linguagem de instruções quânticas de baixo nível, análoga ao papel de linguagens

de montagem em Computação Clássica. Publicado em 2017, é mantido pela empresa norte-americana Rigetti Computing, dedicada ao desenvolvimento de *chips* e processadores quânticos, bem como à oferta de serviços de computação em nuvem voltados para máquinas quânticas. O *PyQuil* desempenha papel central nesse ecossistema, fornecendo a interface de programação de alto nível para interação com os processadores quânticos da Rigetti (acessíveis pela plataforma *Forest*).

Do ponto de vista de uso prático, o *PyQuil* pode ser facilmente obtido via `pip`, sendo compatível com sistemas operacionais modernos e integrado a ambientes interativos como o *Jupyter Notebook* (KLUYVER et al., 2016). Essa integração facilita a prototipagem e a visualização de experimentos, ainda que a biblioteca não ofereça tantas funcionalidades auxiliares ou ferramentas gráficas quanto o *Qiskit*. A documentação oficial é razoavelmente detalhada, com tutoriais simples e guias de referência, mas menos extensa quando comparada a alternativas como *Qiskit*. A comunidade de usuários também é relativamente menor, reflexo do fato de que o foco da Rigetti Computing sempre esteve orientado para o uso de sua própria infraestrutura e máquinas. Ainda assim, o *PyQuil* permanece relevante como uma das primeiras e principais iniciativas a fornecer ferramentas abertas para programação em *hardware* quântico real.

Em termos de atualização e suporte, o *PyQuil* apresenta um ritmo de desenvolvimento mais lento que o observado em ferramentas concorrentes como *Cirq* e *Qiskit*. Ainda que novas versões sejam periodicamente lançadas, a evolução da biblioteca está fortemente ligada ao plano de desenvolvimento estratégico da Rigetti Computing, ou seja, melhorias e novas funcionalidades costumam ser direcionadas a integrar de maneira mais eficiente os recursos de seus próprios processadores quânticos. Essa característica reforça a ligação estreita da ferramenta com o ambiente *Forest*, em contraste com o caráter mais amplo e generalista das demais bibliotecas analisadas.

No que diz respeito às limitações, uma das principais refere-se ao número de *qubits* disponíveis. Assim como em outros provedores de hardware, os processadores da Rigetti Computing são acessíveis em configurações com poucas dezenas de *qubits*, sujeitos ainda a taxas de erro relativamente altas e a restrições de conectividade. Essas condições tornam inviável a execução prática de algoritmos mais complexos, como o de Shor para números maiores, restringindo seu uso a experimentos de pequena escala ou estudos conceituais. Além disso, diferentemente do *Qiskit* e do *Cirq*, o *PyQuil* não possui abstrações de alto nível para operações aritméticas modulares ou rotinas compostas, o que exige do programador a construção manual desses blocos a partir de portas quânticas elementares.

Assim como nas implementações com *Cirq* e *Qiskit*, avaliou-se a utilização do *PyQuil* para o algoritmo de Shor neste trabalho. Entretanto, as diferenças significativas no nível de abstração oferecido pela biblioteca levaram à decisão de não implementar a etapa quântica. A ausência de portas prontas para operações aritméticas compostas

implica que seria necessário desenvolver, do zero, circuitos de exponenciação modular e da Transformada de Fourier Quântica, o que extrapola o escopo deste trabalho. Ainda assim, as etapas clássicas de pré e pós-processamento foram implementadas em *Python*, de modo a preservar a consistência estrutural com as demais implementações.

3.4.1 Pré-processamento com *PyQuil*

Na implementação com *PyQuil*, a etapa de pré-processamento segue a mesma lógica descrita anteriormente para *Cirq* e *Qiskit*. São realizadas verificações clássicas, diretamente em *Python*, que antecedem a construção do circuito quântico.

O Algoritmo 14 apresenta a implementação utilizada. Primeiramente, testa-se se N é par (linhas 2 a 3); nesse caso, 2 já constitui um fator não trivial válido. Em seguida, é verificada a primalidade de N utilizando a biblioteca `sympy` (linhas 5 a 7). Caso N seja primo, o algoritmo se encerra, já que não existem fatores não triviais. Posteriormente, é avaliada a possibilidade de N ser uma potência perfeita (linhas 9 a 12), retornando a base p correspondente como fator válido.

Algoritmo 14 Etapa de pré-processamento com *PyQuil*

```
1  def pre_processamento(N, g=None):
2      if N % 2 == 0:
3          return [2, N // 2]
4
5      if sympy.isprime(N):
6          print("N é primo")
7          return None
8
9      for k in range(2, int(log2(N)) + 1):
10         p = round(N ** (1/k))
11         if p ** k == N:
12             return [p, N // p]
13
14     if g is None:
15         while True:
16             g = random.randint(2, N - 1)
17             if gcd(g, N) == 1:
18                 break
19     else:
20         d = gcd(g, N)
21         if d > 1:
22             return [d, N // d]
23
24     return g
```

Além dessas verificações, ao final do algoritmo (linhas 14 a 22), caso nenhum dos testes anteriores seja conclusivo, é realizada a escolha de um palpite inicial g, selecionado aleatoriamente e que seja coprimo a N (`gcd(g, N) = 1`). Caso g já tenha sido definido anteriormente, confere-se se `gcd(g, N) > 1`. Se positivo, um fator não trivial de N foi encontrado de forma clássica. Portanto, essa etapa garante que a construção do circuito quântico somente será tentada em instâncias de N cuja fatoração não seja trivial ou facilmente identificável por meio desses testes clássicos preliminares.

3.4.2 Etapa quântica com *PyQuil*

Diferentemente das implementações com *Cirq* e *Qiskit*, a etapa quântica não foi implementada em *PyQuil*. Isso se deve ao fato da biblioteca não oferecer primitivas aritméticas (operações) de alto nível - como exponenciação modular controlada ou QFT - disponíveis nas demais ferramentas.

Embora o *PyQuil* forneça uma rica coleção de portas básicas (`H`, `X`, `CNOT`, rotações e portas paramétricas), a ausência de blocos prontos para operações aritméticas complexas exige que a exponenciação modular seja construída manualmente a partir de adições e multiplicações quânticas, cada uma delas também definidas porta a porta. A mesma dificuldade aplica-se à implementação da QFT, que precisa ser escrita inteiramente a partir de portas de rotação controladas. Ainda que viável, esse processo resulta em uma implementação extensa e de baixa abstração, desviando o foco do presente trabalho, que busca comparar diferentes bibliotecas a partir de implementações completas do algoritmo, e não o desenho manual de circuitos aritméticos.

Portanto, optou-se por restringir a implementação em *PyQuil* às etapas clássicas de pré e pós-processamento. Essa decisão também se justifica pelo fato de que, mesmo se a etapa quântica fosse construída manualmente, o pós-processamento depende apenas do formato do resultado da medição, que segue a mesma lógica de contagem de *bitstrings* que já foi implementada anteriormente nas seções 3.2.3 e 3.3.3.

Dessa forma, a ausência de implementação prática com *PyQuil* não compromete a análise comparativa deste trabalho. Pelo contrário, reforça a constatação de que, embora o *PyQuil* se mostre uma ferramenta poderosa e fundamental para exploração de baixo nível e execução em *hardware* proprietário da Rigetti Computing, sendo esta a própria orientação da biblioteca, ele carece de recursos auxiliares e abstrações de alto nível, assim como disponíveis em outros *frameworks* como *Cirq* e *Qiskit*, o que o torna menos adequado para estudos comparativos envolvendo algoritmos complexos em um nível aplicado.

3.4.3 Pós-processamento com *PyQuil*

O pós-processamento em *PyQuil* é estruturalmente semelhante ao observado nas implementações anteriores, mas sua apresentação aqui se justifica por manter a completude e coerência entre as bibliotecas. Assim como em *Cirq* e *Qiskit*, trata-se de uma etapa inteiramente clássica, em que os resultados obtidos da medição dos *qubits* (em formato de *bitstrings*) são convertidos para valores decimais, normalizados pela base $B = 2^m$ e, posteriormente, utilizados no cálculo de aproximações racionais via frações contínuas.

O Algoritmo 15 ilustra esse processo. Inicialmente, são calculadas as fases normalizadas para cada saída registrada (linhas 5 a 9). Em seguida, os resultados são ordenados de maneira decrescente por probabilidade de ocorrência, e para cada uma das fases mais

prováveis busca-se uma aproximação racional cujo denominador seja um candidato ao período r (linhas 11 a 21). Caso um período válido seja encontrado, procede-se ao cálculo clássico dos fatores de N (linhas 22 a 28).

Algoritmo 15 Etapa de pós-processamento com *PyQuil*

```
1  def pos_processamento(medicoes, N, g, n_count):
2      total_shots = sum(medicoes.values())
3      fases = []
4
5      for bitstring, contagem in medicoes.items():
6          inteiro_medido = int(bitstring, 2)
7          fase = inteiro_medido / (2 ** n_count)
8          probabilidade = contagem / total_shots
9          fases.append((fase, probabilidade, inteiro_medido))
10
11     fases.sort(key=lambda x: x[1], reverse=True)
12
13     for fase, prob, inteiro_medido in fases[:20]:
14         if fase == 0:
15             continue
16
17         frac = Fraction(fase).limit_denominator(N)
18         if frac.denominator == 1:
19             continue
20
21         periodo_candidato = frac.denominator
22         if pow(g, periodo_candidato, N) == 1:
23             fator1 = gcd(pow(g, periodo_candidato // 2, N) - 1, N)
24             fator2 = gcd(pow(g, periodo_candidato // 2, N) + 1, N)
25             if 1 < fator1 < N:
26                 return [fator1, N // fator1]
27             if 1 < fator2 < N:
28                 return [fator2, N // fator2]
29
30     return None
```

Embora os exemplos práticos de execução não tenham sido incluídos - dado que a etapa quântica não foi implementada -, o pós-processamento em *PyQuil* foi estruturado de modo a ser compatível com o mesmo formato de saída empregado pelas demais bibliotecas (um dicionário de contagens de *bitstrings*). Dessa forma, assegura-se que, caso a etapa quântica fosse construída, a análise dos resultados poderia prosseguir de forma imediata e sem adaptações adicionais, conforme já visto nas seções 3.2.3 e 3.3.3.

3.5 Análise comparativa entre as implementações

A comparação leva em conta critérios técnicos, conforme estabelecidos em Serrano et al. (2022), e aspectos observados no desenvolvimento prático deste trabalho, englobando desde o suporte a operações aritméticas de alto nível até a maturidade das ferramentas.

A Tabela 1 resume de maneira organizada os aspectos mais relevantes abordados nas subseções de implementação. Nela, são ressaltados elementos como tipo de tecnologia, formato de licenciamento, número de *qubits* disponíveis, documentação, suporte e recursos complementares. Essa síntese possibilita uma comparação direta, tornando mais fácil

identificar as principais diferenças entre as ferramentas e servindo como ponto de partida para a análise crítica apresentada ao longo deste capítulo.

Tabela 1 – Comparação entre *Cirq*, *Qiskit* e *PyQuil*

Critério	**Cirq**	**Qiskit**	**PyQuil**
Tipo	Biblioteca	Biblioteca e *toolkit*	Biblioteca
Licenciamento	Código aberto (Apache 2.0)	Código aberto (Apache 2.0)	Código aberto (Apache 2.0)
Linguagem	Python	Python	Python
Interface	Programas *Python*; Integração com *notebooks*; visualização textual e diagramas básicos de circuitos	Programas *Python*; Integração com *notebooks*; visualização gráfica detalhada de circuitos	Voltada ao ambiente Forest; integração direta a QPUs da Rigetti Computing
N.º de qubits	Simuladores com milhares; acesso restrito a QPUs Google	Simuladores com centenas; acesso a QPUs IBM (dezenas)	Acesso a QPUs Rigetti (dezenas), simuladores limitados
Documentação	Moderada/alta: bem estruturada, mas comunidade menor	Alta: extensa, exemplos práticos e comunidade ativa	Moderada: documentação oficial suficiente, comunidade reduzida
Atualização e suporte	Alto: atualizações regulares, projeto em desenvolvimento ativo	Muito alto: lançamentos frequentes, *roadmap* público e detalhado	Moderado: evolução mais lenta, mas estável
Facilidade de implementar Shor	Moderada: requer construção manual de portas aritméticas compostas	Alta: abstrações prontas para QFT e exponenciação modular	Baixa: seria necessário implementar operações aritméticas de baixo nível manualmente
Suporte a portas aritméticas	Sim, via definição manual de portas compostas	Sim, com suporte interno e extensões	Não diretamente; exige implementação manual
Compatibilidade com hardware real	Sim (QPU Sycamore via Google Cloud, acesso restrito)	Sim (IBM Quantum Experience)	Sim (QPU Rigetti via nuvem)
Desempenho de simulação	Alto: simuladores escaláveis para milhares de qubits	Alto: simuladores otimizados, integração com Aer	Moderado: simuladores funcionais, mas menos otimizados que IBM/Google
Outros recursos	Portas customizadas, abstrações claras, suporte a experimentos	Visualização avançada, tradutores (*transpilers*), otimizadores, tutoriais, e grande ecossistema	Foco em instruções de baixo nível, integração nativa com hardware Rigetti Computing
Adequação ao Algoritmo de Shor	Boa: implementação viável, ainda que mais trabalhosa	Excelente: implementação direta, com suporte pronto a aritmética quântica	Limitada: inviável sem construção manual extensa de circuitos

De maneira geral, o *Qiskit* se destacou como a biblioteca mais completa, disponibilizando um conjunto de recursos, uma documentação extensa e ferramentas extras para a visualização e análise de circuitos. Ademais, a combinação com simuladores eficazes e a possibilidade de acesso a *hardware* quântico real da *IBM Quantum* fortalecem seu aspecto prático para aplicações em maior escala. Já o *Cirq*, se destacou pela flexibilidade na criação de circuitos e pela transparência de suas abstrações, mesmo que demandando um maior esforço manual em determinados trechos, especialmente na implementação de portas aritméticas personalizadas.

Já o *PyQuil*, embora fundamental no ecossistema da Rigetti Computing, apresenta limitações para o propósito deste trabalho. A ausência de portas primitivas de alto nível para operações aritméticas impõe a necessidade de construção manual de circuitos complexos, deslocando o foco da análise. Essa característica, aliada à menor frequência de

atualizações e à documentação mais restrita em comparação ao *Cirq* e *Qiskit*, motivou a decisão de não implementá-lo neste trabalho.

4 Conclusão

O presente trabalho teve como objetivo central avaliar diferentes plataformas de programação em Computação Quântica, com ênfase na implementação e análise do algoritmo de Shor. Partindo de um conjunto de objetivos específicos, buscou-se não apenas sistematizar informações sobre as tecnologias escolhidas, mas também realizar experimentações práticas que permitissem identificar os recursos disponíveis, as limitações inerentes a cada abordagem e as perspectivas futuras de desenvolvimento dessas ferramentas. Nesse sentido, o estudo contribui para o mapeamento atualizado do cenário atual das bibliotecas mais relevantes para programação quântica.

O *Cirq*, desenvolvido pelo Google, apresenta-se como uma biblioteca flexível e transparente, proporcionando ampla liberdade para a criação de circuitos e portas customizadas. Essa abordagem facilita a exploração detalhada dos componentes do algoritmo, proporcionando um controle mais rigoroso sobre cada etapa. Em contrapartida, essa mesma característica exige mais esforço do usuário, especialmente ao lidar com portas aritméticas mais avançadas, como as requeridas na fase de exponenciação modular do algoritmo de Shor. Dessa forma, o *Cirq* equilibra a clareza conceitual e a carga de implementação, posicionando-se como uma ferramenta apropriada para experimentação e pesquisa, embora exija um maior conhecimento dos fundamentos do problema.

Por outro lado, o *Qiskit* se estabelece como a solução mais abrangente entre as analisadas. Desempenhando simultaneamente o papel de biblioteca e *toolkit*, abrange todo o processo de desenvolvimento de algoritmos quânticos, desde a criação e visualização de circuitos até sua compilação, otimização e execução em simuladores ou dispositivos quânticos reais. A ampla documentação, a comunidade engajada e a existência de um plano de desenvolvimento público destacam sua maturidade e confiabilidade, proporcionando um ambiente propício para ensino, pesquisa e aplicações práticas. Junto a isso, o acesso a processadores quânticos reais da IBM expande as oportunidades de experimentação, mesmo com limitações em relação ao número do *qubits*, filas de execução e ruído físico, restringindo o uso em cenários mais complexos.

Já o *PyQuil*, criado pela Rigetti Computing, ocupa uma posição peculiar. Apesar de ser um componente essencial da empresa mantenedora e proporcionar acesso direto a QPUs através da plataforma *Forest*, existem restrições consideráveis para a execução do algoritmo de Shor no escopo deste trabalho. A falta de operações básicas de alto nível para portas aritméticas significa que a construção da etapa quântica exige a criação manual de circuitos complexos, desviando o foco da análise proposta. Por essa razão, decidiu-se limitar a implementação às fases clássicas de pré e pós-processamento, que são

semelhantes às outras plataformas e não demandam recursos específicos da biblioteca. Essa decisão evidencia como a aplicabilidade de cada tecnologia em estudos comparativos é influenciada pelo nível de abstração que ela oferece.

Em termos de execução do algoritmo de Shor, as três abordagens deixam evidente sua natureza híbrida. O pré-processamento, encarregado de eliminar casos triviais como números pares, primos ou potências perfeitas, é executado completamente com recursos da própria linguagem *Python*, sem diferenças significativas entre as plataformas. O pós-processamento, fundamentado em frações contínuas para reconstruir o período da função modular e extrair fatores de N, permanece essencialmente o mesmo também, apresentando variações nos aspectos de implementação relacionados ao formato dos resultados obtidos nas medições. É na etapa quântica que aparecem as principais divergências. Por um lado, o *Cirq* prioriza a criação manual e clara dos circuitos; por outro, o *Qiskit* simplifica o procedimento com abstrações já prontas, e o *PyQuil* não oferece suporte imediato a esse nível de operação, limitando sua aplicação neste trabalho.

Assim, os resultados apresentados permitem entender tanto as vantagens quanto as limitações das tecnologias de programação quântica disponíveis. Cada uma das ferramentas analisadas se sobressai em diferentes características: o *Cirq* se destaca pela flexibilidade e transparência nas implementações; o *Qiskit*, pela abrangência e maturidade; e o *PyQuil*, pela integração direta com *hardware* quântico da Rigetti Computing. Essa variedade de características destaca a relevância de análises comparativas, que auxiliam na localização do estágio atual de desenvolvimento de ferramentas quânticas e direcionam as escolhas futuras conforme as demandas específicas de cada aplicação.

4.1 Trabalhos futuros

Como sugestão para pesquisas futuras, propõe-se expandir os experimentos para incluir outros algoritmos quânticos significativos, como Grover e QFT isolada, investigando tanto a implementação quanto as métricas de desempenho em diversas plataformas. Neste contexto, sugere-se a inclusão de ferramentas emergentes, como o *Braket* da Amazon e o *Ocean* da D-Wave, para ampliar o escopo comparativo e acompanhar o rápido avanço da área. Outra opção é a execução de testes práticos em dispositivos quânticos reais, mesmo com limitações, o que possibilita a avaliação da resistência dos algoritmos diante do ruído e a comparação dos resultados com os obtidos em simulações clássicas.

Referências

ARECCHI, F. T.; COURTENS, E.; GILMORE, R.; THOMAS, H. Atomic Coherent States in Quantum Optics. **Physical Review A**, American Physical Society, v. 6, p. 2211–2237, 1972. Disponível em: <https://doi.org/10.1103/PhysRevA.6.2211>. Citado na página 20.

ARUTE, F.; ARYA, K.; BABBUSH, R.; BACON, D.; BARDIN, J. C.; BARENDS, R.; BISWAS, R.; BOIXO, S.; BRANDAO, F. G.; BUELL, D. A. et al. Quantum Supremacy Using a Programmable Superconducting Processor. **Nature**, Nature Publishing Group, v. 574, n. 7779, p. 505–510, 2019. Disponível em: <https://doi.org/10.1038/s41586-019-1666-5>. Citado na página 34.

BENIOFF, P. The Computer as a Physical System: A Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines. **Journal of Statistical Physics**, Springer, v. 22, n. 5, p. 563–591, 1980. Disponível em: <https://doi.org/10.1007/BF01011339>. Citado na página 12.

BOHR, N. I. On the constitution of atoms and molecules. **The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science**, Taylor & Francis, v. 26, n. 151, p. 1–25, 1913. Disponível em: <https://doi.org/10.1080/14786441308634955>. Citado na página 16.

BORN, M.; BORN, H.; BORN, I.; EINSTEIN, A. **The Born-Einstein Letters: Correspondence Between Albert Einstein and Max and Hedwig Born From 1916 to 1955**. New York: Walker, 1971. ISBN 978-0-333-12560-5. Disponível em: <https://doi.org/10.1007/978-1-349-02076-1>. Citado na página 18.

CHUANG, I. L.; GERSHENFELD, N.; KUBINEC, M. Experimental Implementation of Fast Quantum Searching. **Physical Review Letters**, APS, v. 80, n. 15, p. 3408, 1998. Disponível em: <https://doi.org/10.1103/PhysRevLett.80.3408>. Citado na página 13.

CIRQ. **Announcing Cirq: An Open Source Framework for NISQ Algorithms**. 2018. Disponível em: <https://research.google/blog/announcing-cirq-an-open-source-framework-for-nisq-algorithms/>. Citado 2 vezes nas páginas 28 e 33.

_____. **Install | Cirq | Google Quantum AI**. 2024. Disponível em: <https://quantumai.google/cirq/start/install/>. Citado na página 33.

_____. **Shor's Algorithm | Cirq | Google Quantum AI**. 2024. Disponível em: <https://quantumai.google/cirq/experiments/shor/>. Citado 2 vezes nas páginas 34 e 35.

CROSS, A. The IBM Q Experience and QISKit Open-Source Quantum Computing Software. In: **APS March Meeting Abstracts**. American Physical Society, 2018. v. 2018, p. L58.003. Apresentado no APS March Meeting, 2018. Disponível em: <https://ui.adsabs.harvard.edu/abs/2018APS..MARL58003C>. Citado 2 vezes nas páginas 28 e 43.

DIRAC, P. A. M. A New Notation for Quantum Mechanics. **Mathematical Proceedings of the Cambridge Philosophical Society**, Cambridge University Press, v. 35, n. 3, p. 416–418, 1939. Disponível em: <https://doi.org/10.1017/S0305004100021162>. Citado na página 19.

FEYNMAN, R. P. Simulating Physics with Computers. **International Journal of Theoretical Physics**, v. 21, n. 6/7, 1981. Disponível em: <https://doi.org/10.1007/BF02650179>. Citado na página 12.

GAY, S. J.; NAGARAJAN, R. Communicating Quantum Processes. In: **Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. Long Beach, CA, USA: ACM, 2005. p. 145–157. Disponível em: <https://doi.org/10.48550/arXiv.quant-ph/0409052>. Citado na página 25.

HARRISON, D. M. **Schrödinger's Cat**. 1999. <https://faraday.physics.utoronto.ca/GeneralInterest/Harrison/SchrodCat/cat.gif>. Autor: David M. Harrison. Acesso em: 27 out. 2025. Citado 2 vezes nas páginas 6 e 17.

HEISENBERG, W. On the Perceptual Content of Quantum Theoretical Kinematics and Mechanics. **Zeitschrift für Physik**, Springer, v. 43, n. 3, p. 172–198, 1927. Citado na página 18.

HERTZ, H. On Very Rapid Electric Oscillations. **Annalen der Physik**, v. 267, n. 7, p. 421–448, 1887. Citado na página 16.

IBM Quantum. **Installing Qiskit**. 2025. <https://docs.quantum.ibm.com/start/install>. IBM Quantum. Acesso em: 29 jul. 2025. Citado na página 43.

KLUYVER, T.; RAGAN-KELLEY, B.; PéREZ, F.; GRANGER, B.; BUSSONNIER, M.; FREDERIC, J.; KELLEY, K.; HAMRICK, J.; GROUT, J.; CORLAY, S. et al. Jupyter Notebooks: A Publishing Format for Reproducible Computational Workflows. In: **Positioning and Power in Academic Publishing: Players, Agents and Agendas**. Amsterdam: IOS Press, 2016. p. 87–90. Disponível em: <https://ui.adsabs.harvard.edu/link_gateway/2016ppap.book...87K/doi:10.3233/978-1-61499-649-1-87>. Citado 3 vezes nas páginas 28, 34 e 52.

KOCH, D.; WESSING, L.; ALSING, P. M. Introduction to Coding Quantum Algorithms: A Tutorial Series Using Pyquil. **arXiv preprint arXiv:1903.05195**, 2019. Disponível em: <https://doi.org/10.48550/arXiv.1903.05195>. Citado na página 12.

LAROSE, R. Overview and Comparison of Gate Level Quantum Software Platforms. **Quantum**, Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, v. 3, p. 130, 2019. Disponível em: <https://doi.org/10.22331/q-2019-03-25-130>. Citado na página 24.

PIATTINI, M.; PETERSSEN, G.; PéREZ-CASTILLO, R.; HEVIA, J. L.; SERRANO, M. A.; HERNáNDEZ, G.; GUZMáN, I. G. R. de; PARADELA, C. A.; POLO, M.; MURINA, E. et al. The Talavera Manifesto for Quantum Software Engineering and Programming. In: **QANSWER 2020: Quantum Software Engineering and Programming Workshop**. Talavera de la Reina, Spain: University of Castilla-La Mancha, 2020. p. 1–5. Citado na página 25.

PLANCK, M. On the theory of the energy distribution law of the normal spectrum. **Verh. Deut. Phys. Ges**, v. 2, n. 237, p. 237–245, 1900. Citado na página 16.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. **Communications of the ACM**, ACM New York, NY, USA, v. 21, n. 2, p. 120–126, 1978. Disponível em: <https://doi.org/10.1145/359340.359342>. Citado na página 24.

SCHRÖDINGER, E. The Present Situation in Quantum Mechanics. **Naturwissenschaften**, Springer, v. 23, n. 50, p. 807–812, 823–828, 844–849, 1935. Citado na página 17.

SERRANO, M. A.; CRUZ-LEMUS, J. A.; PEREZ-CASTILLO, R.; PIATTINI, M. Quantum Software Components and Platforms: Overview and Quality Assessment. **ACM Computing Surveys**, ACM New York, NY, v. 55, n. 8, p. 1–31, 2022. Disponível em: <https://doi.org/10.1145/3548679>. Citado 2 vezes nas páginas 25 e 55.

SHOR, P. W. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: **Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)**. Santa Fe, NM, USA: IEEE, 1994. p. 124–134. Disponível em: <https://doi.org/10.1109/SFCS.1994.365700>. Citado 4 vezes nas páginas 14, 23, 28 e 33.

SMITH, R. S.; CURTIS, M. J.; ZENG, W. J. A Practical Quantum Instruction Set Architecture. **arXiv preprint arXiv:1608.03355**, 2016. Disponível em: <https://doi.org/10.48550/arXiv.1608.03355>. Citado 3 vezes nas páginas 25, 28 e 51.

SVORE, K.; GELLER, A.; TROYER, M.; AZARIAH, J.; GRANADE, C.; HEIM, B.; KLIUCHNIKOV, V.; MYKHAILOVA, M.; PAZ, A.; ROETTELER, M. Q# Enabling Scalable Quantum Computing and Development with a High-Level DSL. In: **Proceedings of the Real World Domain Specific Languages Workshop 2018**. Los Angeles, CA, USA: ACM, 2018. p. 1–10. Disponível em: <https://doi.org/10.48550/arXiv.1803.00652>. Citado na página 25.

WANG, J.; GUO, G.; SHAN, Z. SOK: Benchmarking the Performance of a Quantum Computer. **Entropy**, MDPI, v. 24, n. 10, p. 1467, Oct 2022. ISSN 1099-4300. Disponível em: <http://dx.doi.org/10.3390/e24101467>. Citado 2 vezes nas páginas 14 e 24.

Wikimedia Foundation. **Bloch sphere – Wikipedia, The Free Encyclopedia**. 2025. <https://en.wikipedia.org/wiki/Bloch_sphere>. Disponível em: <https://en.wikipedia.org/wiki/Bloch$_sphere$ > .*Acessoem* : 27*out*.2025.*Citado*2*vezesnaspáginas*6*e* 20.

YANOFSKY, N. S.; MANNUCCI, M. A. **Quantum Computing for Computer Scientists**. Cambridge, UK: Cambridge University Press, 2008. ISBN 978-0-521-87996-5. Citado 4 vezes nas páginas 18, 19, 21 e 30.