David Arruda Toneli

Desenvolvimento de um Tutorial para o Ensino de Computação Quântica

São José dos Campos, SP

David Arruda Toneli

Desenvolvimento de um Tutorial para o Ensino de Computação Quântica

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de São Paulo – UNIFESP

Instituto de Ciência de Tecnologia

Bacharelado em Ciência da Computação

Orientador: Prof. Dr. Álvaro Luiz Fazenda

São José dos Campos, SP

Fevereiro de 2022

David Arruda Toneli

Desenvolvimento de um Tutorial para o Ensino de Computação Quântica

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em 10 de fevereiro de 2022:

Prof. Dr. Álvaro Luiz Fazenda
Orientador

Prof. Dr. Claudio Saburo Shida
Convidado 1

Prof. Dra. Denise Stringhini
Convidado 2

São José dos Campos, SP
Fevereiro de 2022

Este trabalho é dedicado aos meus pais.

Agradecimentos

Primeiramente, agradeço à Deus por ter me capacitado para realizar esse trabalho. Eu não seria nada sem Ele. Agradeço aos meus pais por estarem ao meu lado em todos os momentos difíceis da vida. Agradeço aos professores e funcionários do câmpus de São José dos Campos da Unifesp. Em especial, agradeço ao professor Álvaro Luiz Fazenda, meu orientador nesse trabalho.

“O primeiro avião voou apenas por 12 segundos
e, portanto, não havia aplicação prática disso,
mas mostrou a possibilidade de um avião voar.”
(Sundar Pichai)

Resumo

Os avanços recentes alcançados na área da computação quântica demonstram a relevância desse tema e motivaram a realização desse trabalho que tem como objetivo a elaboração de um tutorial para o ensino de computação quântica. Esse tutorial propõe uma abordagem introdutória à programação de computadores quânticos para estudantes que possuam conhecimentos básicos em programação com a linguagem Python. Espera-se contribuir com a elaboração de uma metodologia de ensino de computação quântica para estudantes que não possuem conhecimentos avançados em física. Nesse trabalho é apresentado uma revisão dos conceitos fundamentais e o estado da arte da computação quântica. As definições de qubit e portas quânticas são apresentadas. Uma breve revisão sobre as linguagens disponíveis para computação quântica é realizada nesse trabalho. Nessa revisão, as linguagens utilizadas nos computadores quânticos da IBM, Rigetti e Google foram apresentadas em maiores detalhes. Algumas implementações e simulações também são realizadas nesse trabalho.

Palavras-chaves: computação quântica, qubit, porta quântica, circuito quântico, linguagem de programação quântica.

Abstract

The recent advances achieved in the field of quantum computing demonstrate the relevance of this theme and motivated this work, which aims to prepare a tutorial for teaching quantum computing. This tutorial proposes an introductory approach to quantum computer programming for students with basic Python programming knowledge. It is expected to contribute to quantum computing teaching methodologies for students with no advanced knowledge in physics. This work presents a review of the fundamental concepts and state of the art of quantum computing. The definition of qubit and quantum gates are presented. A brief review of the available quantum computing languages is carried out in this work. In this review, the languages used in quantum computers from IBM, Rigetti and Google were presented in more detail. Some implementations and simulations were also carried out in this work.

Key-words: quantum computing, qubit, quantum gate, quantum circuit, quantum programming language.

Sumário

1 Introdução

Atribui-se o iníício da computação quântica à publicação do artigo *"Simulating physics with computers"* pelo físico norte-americano Richard Feynman (FEYNMAN, 1982). Nas palavras de Feynman, *"Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy."*. Outras grandes contribuições para o desenvolvimento da computação quântica foram dadas pelo físico israelense David Deutsch, que desenvolveu a máquina de Turing quântica em 1985 (DEUTSCH, 1985), pelo matemático norte-americano Peter Williston Shor, que propôs um algoritmo quântico para fatoração em tempo polinomial em 1994 (SHOR, 1994), e pelo engenheiro elétrico indiano Lov Kumar Gorver, que propôs um algoritmo quântico de busca com complexidade $\mathrm{O}\left(\sqrt{N}\right)$ em 1997 (GROVER, 1997).

Um grande marco na história da computação quântica ocorreu em 23 de outubro de 2019, quando a revista Nature publicou um artigo de um grupo de pesquisadores associados à empresa Google (ARUTE et al., 2019) que afirmava ter utilizado o processador quântico Sycamore de 53 qubits para provar que o resultado de um gerador de números aleatórios era realmente aleatório. Essa operação foi realizada em aproximadamente 200 segundos pelo Sycamore e uma estimativa de 10 mil anos foi feita para o tempo que o Summit pudesse completar tal operação. O Summit é um computador de 200 petaflops produzido pela IBM para o governo dos EUA que está localizado no Laboratório Nacional de Oak Ridge. Ao tomar conhecimento, a IBM corrigiu a declaração dos pesquisadores da Google e afirmou que o Summit realizaria tal operação em dois dias e meio (PEDNAULT et al., 2019).

Grandes esforços têm sido feitos para desenvolver processadores quânticos com maiores quantidades de qubits. Até fevereiro de 2022, o processador quântico com maior quantidade de qubits é o Eagle que foi produzido pela IBM e possui 127 qubits. Destacam-se também outros processadores como o Aspen 10 com 32 qubits desenvolvido pela Rigetti, o Tangle Lake com 49 qubits desenvolvido pela Intel e o Bristlecone com 72 qubits desenvolvido pela Google. A IBM pretende lançar, ainda neste ano, o processador Osprey com 433 qubits e, em 2023, o processador Condor com 1.123 qubits. Processadores quânticos requerem temperaturas extremamente baixas para o seu funcionamento. Esses processadores são acoplados na extremidade inferior de um refrigerador de diluição que permite que o processador quântico opere em temperaturas próximas ao zero absoluto. Processadores quânticos também precisam ser blindados de toda interferência externa. Para isso, são inseridos em cilindros onde é feito vácuo e há sistemas de proteção contra campos eletromagnéticos externos.

O grande objetivo de empresas que investem no desenvolvimento de novas tecnologias é inserir no mercado um produto que seja uma inovação tecnológica e, assim, obter o retorno

financeiro pelo investimento realizado. Há uma expectativa muito grande de que computadores quânticos serão comercializados em breve. Antes de inserir computadores quânticos no mercado, é preciso capacitar profissionais para trabalharem com essa tecnologia. Esse é o motivo que levou as empresas que investem no desenvolvimento de computadores quânticos a investirem também em iniciativas para a divulgação e popularização da computação quântica. Entre essas iniciativas estão o desenvolvimento de cursos e tutoriais de ensino e permitir o acesso a simuladores e computadores quânticos. Tanto a IBM quanto a Rigetti oferecem acesso aos seus computadores quânticos o que possibilita o estudo e desenvolvimento de algoritmos quânticos por pesquisadores e estudantes. Dumitrescu *et al.* utilizaram os computadores quânticos da IBM e da Rigetti para calcular a energia de ligação do deutério (DUMITRESCU et al., 2018). Estudos de algoritmos quânticos para aprendizagem de máquina também foram realizados nesses computadores quânticos (CINCIO et al., 2018; ABHIJITH et al., 2018).

Desde o início da computação quântica com os trabalhos de Feynman, a computação quântica tratava-se de uma área de pesquisa acessível apenas a pesquisadores e estudantes de pós-graduação. A partir do momento em que há uma expectativa da inserção de computadores quânticos no mercado e uma preocupação com a formação de profissionais nessa área, surgem trabalhos que propõem metodologias para o ensino de computação quântica tanto para o ensino superior quanto para o ensino médio (ANGARA et al., 2021; HUGHES et al., 2020; CARRASCAL; BARRIO; BOTELLA, 2021; SEEGERER; MICHAELI; ROMEIKE, 2021; RABELO; COSTA, 2018; JESUS et al., 2021; SUN, 2019; GATTI; SOTELO, 2021).

O objetivo desse trabalho é desenvolver um tutorial para o ensino de conceitos básicos de computação quântica para estudantes de cursos de computação que não possuem conhecimentos avançados em física. Os conhecimentos necessários para esse tutorial são conhecimentos matemáticos básicos, que incluem números complexos e representação vetorial, e conhecimentos básicos de programação em linguagem Python. Logo, esse tutorial também pode ser utilizado por estudantes do nível médio que tenham conhecimentos básicos de programação em linguagem Python. O tutorial é dividido em uma série de algoritmos que possibilitam uma aprendizagem prática e estimulam o interesse do estudante pelo assunto. Novos conceitos de computação quântica são introduzidos em cada algoritmo. O título do tutorial desenvolvido nesse trabalho de conclusão de curso é *Tutorial de Programação de Computadores Quânticos*. Esse tutorial encontra-se no Apêndice A e está disponível em quantumcomputing.orgfree.com.

Nos próximos capítulos desse trabalho, conceitos de computação quântica são apresentados de um modo mais formal o que requer conhecimentos em álgebra linear e mecânica quântica. A forma como os conteúdos são apresentados nos capítulos a seguir difere da forma em que são apresentados no tutorial. O tutorial foi desenvolvido para estudantes e não requer conhecimentos avançados em física e álgebra linear. Esse trabalho e os outros trabalhos que são citados aqui são referências para estudantes que desejam aprofundar seus conhecimentos em computação quântica. O capítulo 2 trata apenas da definição de qubit por ser um tópico

de grande relevância para esse trabalho. As principais portas quânticas utilizadas no desenvolvimento de algoritmos quânticos são apresentadas no capítulo 3. Atualmente há uma grande quantidade de linguagens para programação de computadores quânticos. Um breve revisão dessas linguagens é apresentada no capítulo 4 onde é dado uma maior atenção para as linguagens utilizadas nos computadores da IBM, Rigetti e Google. O capítulo 5 é dedicado ao estudo de um estado emaranhado. Nesse capítulo é apresentado a implementação de um circuito quântico que gera um estado emaranhado. O capítulo 6 apresenta a implementação de um circuito quântico *Full Adder*. Uma analogia é realizada entre o circuito quântico *Full Adder* e o circuito lógico *Full Adder*. O capítulo 7 trata do algoritmo de Grover. Esse capítulo apresenta definições e cálculos bastante complexos, portanto requer um maior tempo de dedicação para a sua compreensão. O algoritmo de Grover é um dos algoritmos básicos da computação quântica sendo apresentado na grande maioria dos tutoriais de ensino (ASFAW et al., 2020). Como um dos algoritmos básicos, pode-se dizer que o algoritmo de Grover é um algoritmo de fácil compreensão quando comparado a outros algoritmos o que demonstra o quão complicado pode ser o processo de ensino e aprendizagem da computação quântica. O capítulo 8 apresenta uma descrição do método utilizado no tutorial desenvolvido nesse trabalho para ensinar conceitos de computação quântica sem a necessidade de conhecimentos avançados em física e matemática. Esse trabalho de conclusão de curso encerra-se com as conclusões finais apresentadas no capítulo 9. O Anexo A contém o tutorial para o ensino de programação de computadores quânticos desenvolvido nesse trabalho.

2 Fundamentação Teórica

2.1 Bit quântico

De acordo com o princípio da complementariedade, os modelos corpuscular e ondulatório são complementares tal que, se uma experiência comprovar o caráter ondulatório da radiação, ou da matéria, então essa experiência é incapaz de comprovar o caráter corpuscular e vice-versa. Medidas da razão entre a carga e a massa de partículas em um tubo de raios catódicos resultaram na descoberta do elétron por em Joseph John Thomson em 1906. Esse experimento levou à conclusão de que o elétron seria uma partícula. Porém, em 1937, George Paget Thomson, através de experimentos de difração de elétrons comprovam o caráter ondulatório do elétron confirmando o postulado de ondas de matéria de Victor Pierre Raymond de Broglie.

Enquanto, na mecânica clássica, à uma partícula é associada uma função que determina sua posição em função do tempo, na mecânica quântica, como resultado do princípio da complementariedade, uma função de onda é associada à partícula. Grandezas cinemáticas e dinâmicas podem ser obtidas a partir dessa função de onda.

Dada uma função de onda $\psi(\vec{r},t)$ associada a uma partícula, a probabilidade de encontrar essa partícula em uma região de volume infinitesimal dado por d^3r, na posição determinada por $\vec{r}$, no tempo t, é dada por $|\psi(\vec{r},t)|^2 d^3r$. Como a partícula deve estar em alguma posição do espaço, a integral de sua probabilidade sobre todo o espaço deve ser igual a 1. As funções de onda que satisfazem $\int |\psi(\vec{r},t)|^2 \mathrm{d}^3r = 1$ pertencem ao conjunto das funções quadrado-integráveis L^2. As funções de onda associadas às partículas também devem ser definidas em todo ponto, contínuas e infinitamente diferenciáveis, portanto pertencem a um conjunto F que é subconjunto de L^2. Pode-se provar que F e L^2 satisfazem aos critérios de um espaço vetorial, portanto as funções de onda desses conjuntos podem ser escritas em função das componentes de uma base para esses espaços vetoriais, ou seja, $\psi(\vec{r}) = \sum_i c_i u_i(\vec{r})$, onde $\{u_i(\vec{r})\}$ é uma base de F.

Na notação de Dirac, o estado quântico de uma partícula é caracterizado por um vetor de estado que pertence a um espaço ε que é o espaço de estados da partícula. Toda função de onda que pertence a F é associada a um vetor de estado que pertence ao espaço de estados ε. O vetor de estado, que também pode ser chamado de ket, é representado pelo símbolo $|x\rangle$, onde x representa qualquer elemento utilizado para distinguir os diferentes vetores de estado. Como o espaço das funções quadrado-integráveis é uma realização de um espaço de Hilbert e o espaço F é um subconjunto de L^2, temos que o espaço de estados é um subconjunto de um espaço de Hilbert (COHEN-TANNOUDJI; DIU; LALOË, 1977).

O qubit é matematicamente definido por um vetor de estado $|\psi\rangle$ representado sobre a base $\{|0\rangle, |1\rangle\}$ de seu espaço de estados (NIELSEN; CHUANG, 2010):

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle , \tag{2.1}$$

onde $\alpha, \beta \in \mathbb{C}$ e $|\alpha|^2 + |\beta|^2 = 1$. A base $\{|0\rangle, |1\rangle\}$ também é chamada de base computacional. Quando $\alpha = 1$ e $\beta = 0$, o qubit estará no estado $|0\rangle$ que é associado ao valor 0 armazenado em um bit de um processador clássico. Ao valor 1 armazenado em um bit é associado o estado $|1\rangle$ de um qubit que ocorre para $\alpha = 0$ e $\beta = 1$. Ao contrário de um bit de um processador clássico que deve armazenar o valor 0 ou o valor 1 em cada operação, o qubit de um processador quântico pode estar nos estados $|0\rangle$ e $|1\rangle$ em um mesma operação devido à superposição de estados. O paralelismo quântico, que é realizado pela superposição de estados, garante a maior eficiência de um processador quântico quando comparado a um processador clássico.

2.2 Portas quânticas

Por definição, algoritmo é um conjunto de regras e procedimentos lógicos perfeitamente definidos que levam à solução de um problema em um número finito de etapas. Um algoritmo quântico é um conjunto de procedimentos que leva à construção de um circuito quântico para a solução de um determinado problema. Em um processador clássico, operações sobre bits são realizadas por portas lógicas agrupadas em circuitos lógicos. Em um processador quântico, operações sobre qubits são realizadas por portas quânticas dispostas em um circuito quântico. Nas próximas seções deste capítulo serão descritas as principais portas quânticas para programação de computadores quânticos.

2.2.1 Porta de Pauli I

A porta de Pauli I é definida pelo operador linear:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} . \tag{2.2}$$

A aplicação da porta de Pauli I não altera o estado do qubit, $I |\psi\rangle = |\psi\rangle$, portanto, essa porta também é chamada de porta identidade. Em circuitos quânticos, a porta de Pauli I é representada por uma caixa com a letra *I* em seu interior como mostrado na Figura 1.

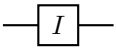

Figura 1 – Representação da porta de Pauli I em circuitos quânticos.

2.2.2 Porta de Pauli X

A porta de Pauli X é matematicamente definida pelo operador linear:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{2.3}$$

A porta de Pauli X, quando aplicada sobre $|\psi\rangle$, realiza a troca dos coeficientes da base $\{|0\rangle, |1\rangle\}$. Costuma-se comparar a porta de Pauli X com a porta clássica NOT, porque $X|0\rangle = |1\rangle$ e $X|1\rangle = |0\rangle$. Há duas formas de representar a porta de Pauli X em circuitos quânticos. Essas formas são mostradas na Figura 2.

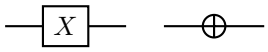

Figura 2 – Representações da porta de Pauli X em circuitos quânticos.

2.2.3 Porta de Pauli Y

A porta de Pauli Y é definida pelo operador linear:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}. \tag{2.4}$$

A ação da porta de Pauli Y sobre $|\psi\rangle$ resulta no estado $Y|\psi\rangle = -i\beta|0\rangle + i\alpha|1\rangle$. A representação da porta de Pauli Y em circuitos quânticos é mostrada na Figura 3.

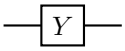

Figura 3 – Representação da porta de Pauli Y em circuitos quânticos.

2.2.4 Porta de Pauli Z

A porta de Pauli Z é matematicamente definida pelo operador linear:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{2.5}$$

Como resultado da aplicação da porta de Pauli Z sobre $|\psi\rangle$, ocorre a inversão de sinal do coeficiente do estado da base $|1\rangle$. A Figura 4 mostra a representação da porta de Pauli Z em circuitos quânticos.

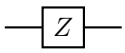

Figura 4 – Representação da porta de Pauli Z em circuitos quânticos.

2.2.5 Porta de Hadamard

A superposição de estados pode ser realizada pela aplicação da porta de Hadamard sobre o estado de um qubit. A porta de Hadamard é matematicamente definida como sendo o operador linear:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.6}$$

Quando aplicada sobre o estado $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$, a porta de Hadamard resultará em $H |\psi\rangle = \left((\alpha + \beta)/\sqrt{2}\right) |0\rangle + \left((\alpha - \beta)/\sqrt{2}\right) |1\rangle$. Para o estado $|1\rangle$, o resultado será $H |1\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$ e, para o estado $|0\rangle$, $H |0\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$. Em circuitos quânticos, a porta de Hadamard é representada por uma caixa com a letra *H* em seu interior como mostrado na Figura 5.

Figura 5 – Representação da porta de Hadamard em circuitos quânticos.

2.2.6 Porta CNOT

A porta CNOT ou NOT-controlada é uma porta que atua em dois qubits. O primeiro qubit é chamado de qubit de controle e o segundo qubit é chamado de qubit alvo. O qubit alvo mudará o seu estado quando o qubit de controle estiver no estado $|1\rangle$. A ação da porta CNOT pode ser representada por: $CNOT |a, b\rangle \rightarrow |a, a \oplus b\rangle$, onde $a, b \in \{0, 1\}$. Assim, $CNOT |00\rangle \rightarrow |00\rangle$, $CNOT |01\rangle \rightarrow |01\rangle$, $CNOT |10\rangle \rightarrow |11\rangle$ e $CNOT |11\rangle \rightarrow |10\rangle$.

A porta CNOT é matematicamente definida pelo operador linear

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.7}$$

As representações da porta CNOT em circuitos quânticos são mostradas na Figura 6. O qubit de controle é representado pelo ponto e o qubit alvo é representado pelo símbolo $\oplus$ ou pelo símbolo que representa uma porta de Pauli X.

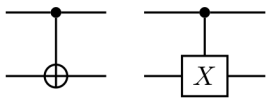

Figura 6 – Representações da porta CNOT em circuitos quânticos.

2.2.7 Porta TOFFOLI

Assim como a porta CNOT, a porta TOFFOLI também é uma porta controlada, porém, há dois qubits de controle. Sendo $|a\rangle$ e $|b\rangle$ os qubits de controle e $|c\rangle$ o qubit alvo, a ação da porta TOFFOLI pode ser representada por: $TOFFOLI\,|a,b,c\rangle \rightarrow |a,b,c \oplus (a \text{ AND } b)\rangle$, onde a, b, $c \in \{0, 1\}$. O operador linear que define a porta TOFFOLI é

$$TOFFOLI = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.8}$$

A Figura 7 mostra as representações da porta TOFFOLI em circuitos quânticos. Os dois qubits de controle são representados por pontos e o qubit alvo é representado pelo símbolo $\oplus$ ou pelo símbolo que representa a porta de Pauli X.

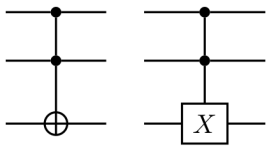

Figura 7 – Representações da porta TOFFOLI em circuitos quânticos.

2.2.8 Porta SWAP

A porta SWAP é matematicamente definida pelo operador linear:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.9}$$

A ação da porta SWAP é descrita por $SWAP\,|a,b\rangle \rightarrow |b,a\rangle$, onde $a, b \in \{0,1\}$. A porta SWAP é equivalente à aplicação de três portas CNOT em sequência sendo que a segunda porta CNOT deve estar invertida, ou seja, o qubit de controle na primeira e terceira porta CNOT é o qubit alvo na segunda porta CNOT e o qubit alvo na primeira e terceira porta CNOT é o qubit de controle na segunda porta CNOT. A Figura 8 mostra a representação da porta Swap em um circuito quântico.

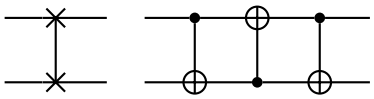

Figura 8 – Representação da porta SWAP em circuitos quânticos.

2.3 Estado emaranhado

Quando duas ou mais partículas interagem e, como resultado dessa interação, um estado emaranhado é obtido, o estado quântico de cada partícula não pode mais ser descrito independentemente. O estado emaranhado, que também é conhecido como estado de Bell, representa o sistema como um todo e o que ocorre em uma partícula afeta as outras partículas do sistema mesmo que essas partículas estejam separadas por uma distância muito grande. Para gerar um estado emaranhado, pode-se utilizar as portas de Hadamard e CNOT e dois qubits no estado $|0\rangle$. Primeiramente, aplica-se a porta de Hadamard sobre o primeiro qubit que servirá como qubit de controle para a porta CNOT. Em seguida, aplica-se a porta CNOT.

$$CNOT.(H \otimes I)\,|00\rangle = CNOT\left(\frac{|00\rangle + |10\rangle}{\sqrt{2}}\right) = \frac{|00\rangle + |11\rangle}{\sqrt{2}}. \tag{2.10}$$

A Figura 9 mostra o circuito quântico equivalente à Equação 2.10. O símbolo semelhante a um visor de voltímetro indica um processo de medição dos qubits.

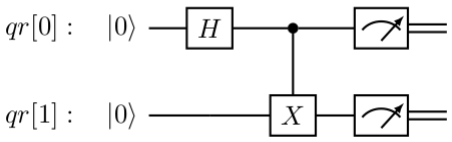

Figura 9 – Circuito quântico para gerar um estado emaranhado.

Comparando o estado obtido na Equação 2.10 com o estado que é resultado da superposição dos estados de dois qubits $|\psi_1\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ e $|\psi_2\rangle = \gamma\,|0\rangle + \delta\,|1\rangle$ dado por $\alpha\gamma\,|00\rangle + \alpha\delta\,|01\rangle + \beta\gamma\,|10\rangle + \beta\delta\,|11\rangle$ temos um sistema de equações que não admite solução. Portanto, o estado emaranhado é um estado que não deveria existir. Experimentos recentes têm

encontrado evidências de estados emaranhados de fótons (MOREAU et al., 2019) e elétrons (QIAO et al., 2020).

2.4 *Full Adder* quântico

O circuito quântico equivalente ao circuito lógico *Full Adder* pode ser construído com o uso de portas CNOT e TOFFOLI. A Figura 10 mostra o circuito quântico *Full Adder*. Os qubits qr[0] e qr[1] armazenam os valores a serem somados. O qubit qr[2] equivale ao bit *carry in* e o qubit qr[3] equivale ao bit *carry out*. O resultado da soma pode ser obtida com a medição dos qubits qr[2] e qr[3]. As portas de Pauli X são utilizadas para modificar os estados iniciais dos qubits para obter os valores a serem somados.

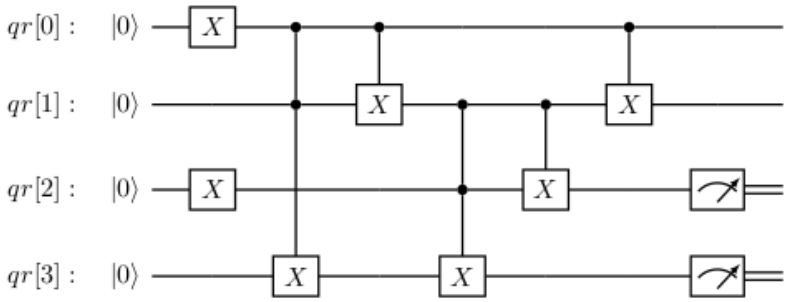

Figura 10 – Circuitos quântico *Full Adder*.

2.5 Algoritmo de Grover

Nesta seção, será apresentado o algoritmo de Grover que propõe uma solução de complexidade $O\left(\sqrt{N}\right)$ para a tarefa de busca em uma lista não ordenada. A solução para essa mesma tarefa em um computador clássico teria complexidade $O\left(N\right)$. O conteúdo desse capítulo baseia-se no livro *"Uma introdução à computação quântica"* (PORTUGAL et al., 2004) e em suas referências.

2.5.1 Descrição matemática do algoritmo de Grover

Para um dado número $n \in \mathbb{N}$, seja {a, b, ...} uma lista contendo $N = 2^n$ elementos. Cada elemento pode armazenar um determinado valor de modo a produzir uma lista não ordenada. O elemento a ser procurado nessa lista encontra-se na posição $i = i_0$ onde $i \in \{0, 1, ..., N\text{-}1\}$. Seja f: $\{0, 1, ..., N\text{-}1\} \rightarrow \{0, 1\}$ uma função que determina a posição do elemento procurado definida por:

$$f(i) = \begin{cases} 1, \text{ se } i = i_0 \\ 0, \text{ se } i \neq i_0 \end{cases} . \tag{2.11}$$

Para implementar o algoritmo de Grover para buscar um determinado valor em uma lista com N elementos, seria preciso de um processador quântico com, pelo menos, n+1 qubits. O algoritmo de Grover utiliza dois registradores, sendo que, um deles deve conter n qubits e outro, apenas 1 qubit. No registrador com n qubits, todos os qubits devem ser inicializados no estado $|0\rangle$ resultando no estado $|0...0\rangle$ que está associado ao elemento $i = 0$ da lista. Esse é um dos estados que formam a base para os estados do registrador de n qubits. Portanto, cada elemento da lista está associado a um dos estados da base do registrador de n qubits.

Como exemplo, considere uma lista com quatro elementos. Nesse caso, $i \in \{0, 1, 2, 3\}$ e os estados da base de estados para o registrador de $n = 2$ qubits são: $|00\rangle$, $|01\rangle$, $|10\rangle$ e $|11\rangle$. Em notação decimal, esses mesmos estados são representados respectivamente por: $|0\rangle$, $|1\rangle$, $|2\rangle$ e $|3\rangle$. As matrizes que representam esses estados são:

$$|00\rangle = |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, |01\rangle = |1\rangle = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, |10\rangle = |2\rangle = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \text{ e } |11\rangle = |3\rangle = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}.$$

O registrador de um qubit faz-se necessário para identificar o elemento da lista em que se encontra o valor buscado. O estado inicial desse registrador deve ser o estado $|1\rangle$. A base de estados desse registrador é a base computacional $\{|0\rangle, |1\rangle\}$. Portanto, a base de estados para o sistema completo, que inclui todos os qubits, será $\{|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle\}$ ou, em notação decimal, $\{|0\rangle, |1\rangle, |2\rangle, |3\rangle, |4\rangle, |5\rangle, |6\rangle, |7\rangle, |8\rangle\}$. As matrizes que representam os elementos dessa base são:

$$|0\rangle = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}, \; |1\rangle = \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix}, \; |2\rangle = \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\end{bmatrix}, \; |3\rangle = \begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix}, \; |4\rangle = \begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix}, \; |5\rangle = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix},$$

$$|6\rangle = \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix} \text{ e } |7\rangle = \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}.$$

A vantagem de um processador quântico é a possibilidade de executar todas as entradas possíveis em uma única execução de um estado obtido pela superposição de todos os estados. Assim, a primeira coisa a se fazer é aplicar a porta de Hadamard sobre todos os estados do dois registradores como mostrado na Figura 11.

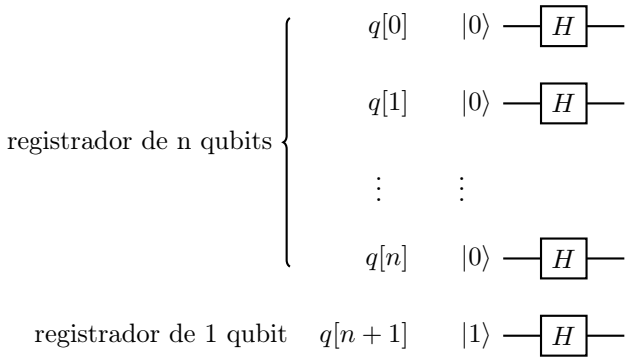

Figura 11 – Registradores de n qubits e 1 qubit utilizados na implementação do algoritmo de Grover.

O resultado será o estado

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle \tag{2.12}$$

para o registrador que contém n qubits e, para o registrador de um qubit, o estado

$$|-\rangle = \frac{1}{\sqrt{2}} \left(|0\rangle - |1\rangle\right). \tag{2.13}$$

A busca pelo elemento da lista é realizada com o auxílio da função f em um algoritmo clássico. Em um algoritmo quântico, a busca é feita com a aplicação de um operador linear sobre o estado quântico do sistema. O operador associado à função f, identificado por U_f, é definido por

$$U_f \left(|i\rangle |0\rangle\right) = \begin{cases} |i\rangle |1\rangle, \text{ se } i = i_0 \\ |i\rangle |0\rangle, \text{ se } i \neq i_0 \end{cases} \tag{2.14}$$

e

$$U_f \left(|i\rangle |1\rangle\right) = \begin{cases} |i\rangle |0\rangle, \text{ se } i = i_0 \\ |i\rangle |1\rangle, \text{ se } i \neq i_0 \end{cases}. \tag{2.15}$$

Essas equações podem ser escritas como $U_f \left(|i\rangle |j\rangle\right) = |i\rangle |j \oplus f(i)\rangle$.

Portanto, deve-se aplicar o operador linear U_f sobre o estado $|\psi\rangle|-\rangle$ obtido após a aplicação das portas de Hadamard sobre os estados iniciais dos qubits do sistema.

$$\begin{aligned}
U_f\left(|\psi\rangle|-\rangle\right) &= U_f\left(\left(\frac{1}{\sqrt{N}}\sum_{i=0}^{N-1}|i\rangle\right)\left(\frac{1}{\sqrt{2}}\left(|0\rangle-|1\rangle\right)\right)\right) = \\
&= \frac{1}{\sqrt{N}}\sum_{i=0}^{N-1}U_f\left(\frac{1}{\sqrt{2}}\left(|i\rangle|0\rangle-|i\rangle|1\rangle\right)\right) = \\
&= \frac{1}{\sqrt{N}}\sum_{i=0}^{N-1}\frac{1}{\sqrt{2}}\left(U_f\left(|i\rangle|0\rangle\right)-U_f\left(|i\rangle|1\rangle\right)\right) = \\
&= \frac{1}{\sqrt{N}}\sum_{i=0}^{N-1}\frac{1}{\sqrt{2}}\left(|i\rangle|0\oplus f(i)\rangle-|i\rangle|1\oplus f(i)\rangle\right) = \\
&= \frac{1}{\sqrt{N}}\left(\left(\sum_{i=0,i\neq i_0}^{N-1}\frac{1}{\sqrt{2}}\left(|i\rangle|0\rangle-|i\rangle|1\rangle\right)\right)+\frac{1}{\sqrt{2}}\left(|i_0\rangle|1\rangle-|i_0\rangle|0\rangle\right)\right) = \\
&= \frac{1}{\sqrt{N}}\left(\left(\sum_{i=0,i\neq i_0}^{N-1}\frac{1}{\sqrt{2}}|i\rangle\left(|0\rangle-|1\rangle\right)\right)+\frac{1}{\sqrt{2}}|i_0\rangle\left(|1\rangle-|0\rangle\right)\right) = \\
&= \frac{1}{\sqrt{N}}\left(\left(\sum_{i=0,i\neq i_0}^{N-1}|i\rangle|-\rangle\right)-|i_0\rangle|-\rangle\right) = \\
&= \left(\frac{1}{\sqrt{N}}\sum_{i=0}^{N-1}(-1)^{f(i)}|i\rangle\right)|-\rangle\,. \qquad (2.16)
\end{aligned}$$

A partir do resultado acima, pode-se extrair duas importantes propriedades da ação do operador linear U_f sobre o estado $|\psi\rangle|-\rangle$:

- o estado $|i_0\rangle$, que está associado ao elemento da lista procurado, tem o sinal da sua amplitude modificado de positivo para negativo;
- o estado $|-\rangle$ do segundo registrador não é alterado pela ação do operador U_f.

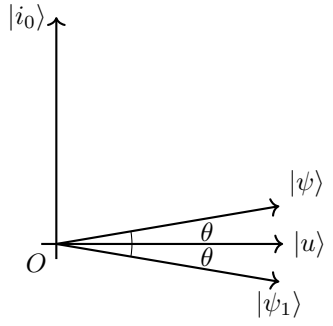

Figura 12 – Representação gráfica dos estados $|\psi\rangle$ e $|\psi_1\rangle$.

A mudança de sinal de um estado da base $\{|0\rangle, |1\rangle, ... |N-1\rangle\}$ implica em uma reflexão em relação ao subespaço ortogonal a esse estado. A Figura 12 mostra a reflexação do estado $|\psi\rangle$ em relação ao subespaço gerado por $|u\rangle$, onde

$$|u\rangle = \sum_{i=0, i\neq i_0}^{N-1} |i\rangle \,. \tag{2.17}$$

Esse subespaço é ortogonal a $|i_0\rangle$. O estado refletido $|\psi_1\rangle$ deve-se à ação de U_f sobre $|\psi\rangle$.

Embora o elemento a ser procurado na lista tenha sido identificado em uma única execução, não é possível extrair essa informação do sistema porque medidas de um sistema quântico retornam o quadrado do módulo da amplitude de cada estado da base na qual o estado quântico do sistema está representado. Portanto, uma medida do estado do sistema após a aplicação do operado U_f representado na Equação 2.16 resulta em probabilidades iguais a $1/N = \left|\pm 1/\sqrt{N}\right|^2$ para todos os estados da base.

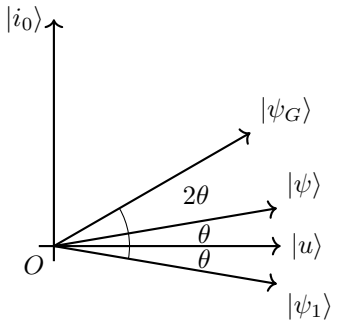

Figura 13 – Representação gráfica da reflexão do estado $|\psi_1\rangle$ sobre o estado $|\psi\rangle$.

Uma solução para obter um estado no qual a amplitude de sua componente em $|i_0\rangle$ fosse superior às componentes dos outros estados seria aplicar um operador linear sobre o estado $|\psi_1\rangle$ que resultasse na reflexão desse estado sobre o estado $|\psi\rangle$. O operador linear $2|\psi\rangle\langle\psi| - I$, quando aplicado sobre o estado $|\psi_1\rangle$, resulta no estado $|\psi_G\rangle$ que é a reflexão de $|\psi_1\rangle$ sobre $|\psi\rangle$ como mostra a Figura 13. Dessa forma, uma medida do estado $|\psi_G\rangle$ permite identificar o elemento procurado da lista.

$$\begin{aligned}
|\psi_G\rangle &= (2|\psi\rangle\langle\psi| - I)|\psi_1\rangle = \\
&= (2|\psi\rangle\langle\psi| - I)\left(\frac{1}{\sqrt{N}}\sum_{i=0}^{N-1}(-1)^{f(i)}|i\rangle\right) = \\
&= (2|\psi\rangle\langle\psi| - I)\left(|\psi\rangle - \frac{2}{\sqrt{N}}|i_0\rangle\right) = \\
&= 2\langle\psi|\psi\rangle|\psi\rangle - I|\psi\rangle - \frac{4}{\sqrt{N}}\langle\psi|i_0\rangle|\psi\rangle + \frac{2}{\sqrt{N}}I|i_0\rangle = \\
&= \frac{N-4}{N}|\psi\rangle + \frac{2}{\sqrt{N}}|i_0\rangle \,.
\end{aligned} \tag{2.18}$$

Ao circuito quântico da Figura 11 acrescentamos os oráculos que representam a aplicação dos operadores U_f e $2\left|\psi\right\rangle\left\langle\psi\right| - I$. O operador de Grover é formado pela composição desses dois operadores, $G = \left(\left(2\left|\psi\right\rangle\left\langle\psi\right| - I\right) \otimes I\right) U_f$.

A componente de $\left|\psi_G\right\rangle$ no estado $\left|i_0\right\rangle$ é dado por:

$$\left\langle i_0\right|\left.\psi_G\right\rangle = \frac{N-4}{N}\left\langle i_0\right|\left.\psi\right\rangle + \frac{2}{\sqrt{N}}\left\langle i_0\right|\left.i_0\right\rangle = \frac{N-4}{N}\frac{1}{\sqrt{N}} + \frac{2}{\sqrt{N}} = \frac{3N-4}{N\sqrt{N}}. \tag{2.19}$$

A medida do estado $\left|\psi\right\rangle$ resulta em uma probabilidade de $\left|1/\sqrt{N}\right|^2$ para obter o estado $\left|i_0\right\rangle$, enquanto uma medida do estado $\left|\psi_G\right\rangle$ resulta em uma probabilidade $\left|\left(3N-4\right) / \left(N\sqrt{N}\right)\right|^2$ sendo possível determinar o elemento procurado da lista. Porém, para valores grandes de N, essa probabilidade ainda é baixa sendo necessária outras aplicações do operador de Grover. Pode-se demonstrar que o número de aplicações do operador de Grover é inferior a $\sqrt{N}$ o que garante que a complexidade do algoritmo é $O\left(\sqrt{N}\right)$ (PORTUGAL et al., 2004).

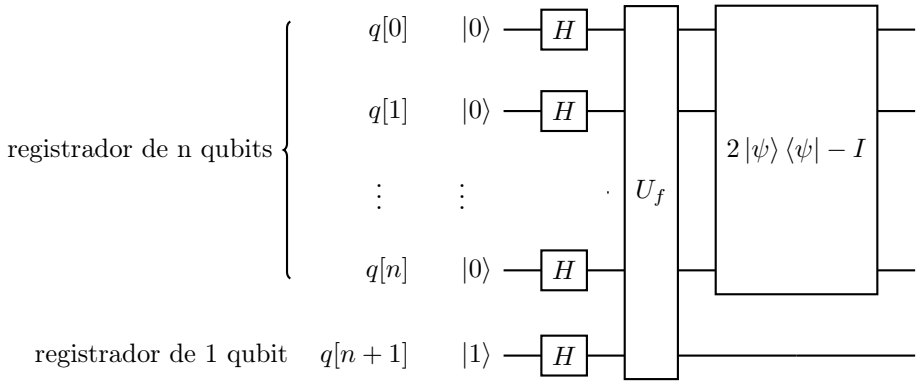

Figura 14 – Representação do operador de Grover em um circuito quântico.

2.5.2 Circuito quântico do algoritmo de Grover

Na Figura 14, há um esboço de um circuito quântico para o algoritmo de Grover. O operador U_f é aplicado sobre todos os qubits dos dois registradores e o operador $2\left|\psi\right\rangle\left\langle\psi\right| - I$ é aplicado apenas sobre os qubits do primeiro registrador. Para cada valor do elemento i_0 que deve ser procurado na lista, a componente do circuito que representa o operador U_f é modificada. A Figura 15 mostra uma possível representação para U_f para $n = 2$ e $i_0 = 1$ em (a) e para $n = 2$ e $i_0 = 2$ em (b). Quando ativada pela entrada correta, a porta Toffoli muda o sinal do estado $\left|-\right\rangle$. A Figura 16 mostra uma das formas em que se pode representar a componente do operador $2\left|\psi\right\rangle\left\langle\psi\right| - I$ em um circuito quântico para o algoritmo de Grover para $n = 2$. A Figura 17 mostra um circuito quântico que corresponde ao algoritmo de Grover para $n = 2$ e $i_0 = 1$.

O algoritmo de Grover e sua representação em circuitos quânticos é apresentada com maiores detalhes por Renato Portugal e Franklin L. Marquezino (PORTUGAL; MARQUEZINO, 2019) que incluem também um circuito quântico que possui um qubit a menos chamado

de circuito econômico com a mesma funcionalidade do circuito quântico apresentado na Figura 17.

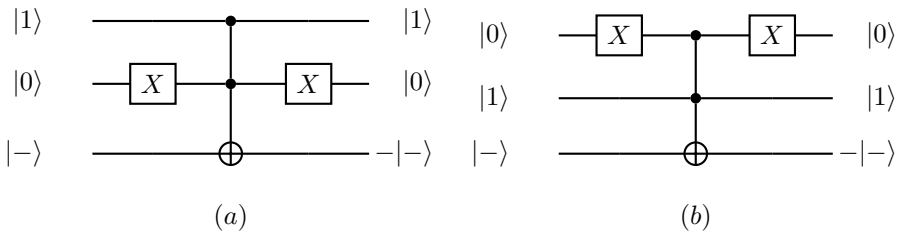

Figura 15 – Representação do operador U_f em um circuito quântico para $n = 2$ e $i_0 = 1$ (a) e $i_0 = 2$ em (b).

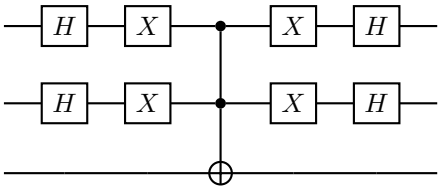

Figura 16 – Representação do operador $2\left|\psi\right\rangle\left\langle\psi\right| - I$ em um circuito quântico.

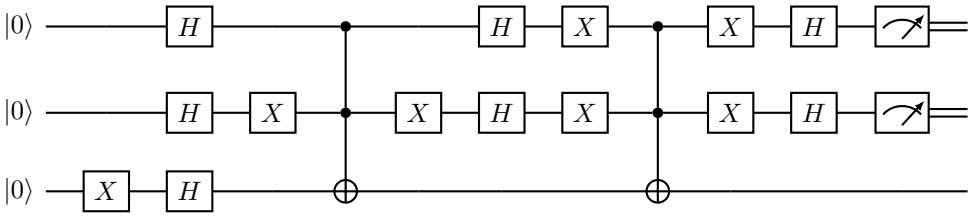

Figura 17 – Circuito quântico para o algoritmo de Grover para $n = 2$ e $i_0 = 1$.

2.6 Linguagens de computação quântica

Para escrever um algoritmo quântico, faz-se necessário uma linguagem de programação para um computador quântico. Porém, cada empresa que está desenvolvendo um computador quântico também desenvolve uma linguagem de programação para operar a sua máquina. Há também linguagens desenvolvidas por pesquisadores e grupos de pesquisa. O lado positivo e o lado negativo do cenário atual da computação quântica foi destacado por LaRose: *"This sheer number of programs, while positively reflecting the growth of the field, makes it difficult for students and researchers to decide on which software platform to use, getting lost in documentation or being too overwhelmed to know where to start"* (LAROSE, 2019). Em alguns dos principais

trabalhos dedicados à revisão de linguagens para a computação quântica (GARHWAL; GHORANI; AHMAD, 2021; HEIM et al., 2020; GAY, 2006; SELINGER, 2004a), encontra-se uma divisão em linguagens que seguem o paradigma imperativo e linguagens que seguem o paradigma funcional.

Knill (KNILL, 1996) propôs convenções para a escrita de pseudocódigos consistentes com o modelo de máquina quântica de acesso aleatório (QRAM – *"Quantum Random Access Machine"*) para computadores quânticos. Esse trabalho teve grande influência no desenvolvimento das linguagens imperativas para a computação quântica. Entre as linguagens imperativas estão as linguagens QCL (*"Quantum Computing Language"*) (ÖMER, 1998), qGCL (*"Quantum Guarded Command Language"*) (ZULIANI, 2001), Q (BETTELLI; CALARCO; SERAFINI, 2003), QASM (*"Quantum Assembly Language"*) (SVORE et al., 2006), Quil (SMITH; CURTIS; ZENG, 2016), LanQ (MLNARIK, 2007) e $Q\,|SI\rangle$ (LIU et al., 2018).

Lisp, Miranda, ML, Haskell são linguagens funcionais baseadas no Lambda Cálculo. Maymin (MAYMIN, 1996) propôs Lambda-Q Cálculo que é uma extensão do Lambda Cálculo para linguagens de programação quântica. Entre as linguagens funcionais para computação quântica estão as linguagens QPL (*Quantum Programming Language*) (SELINGER, 2004b), QFC (*Quantum Flow Charts*) (SELINGER, 2004b), QML (*Quantum Markup Language*) (ALTENKIRCH; GRATTAGE, 2005), Quipper (GREEN et al., 2013), QuaFL (LAPETS et al., 2013) e o LIQ$Ui\,|\rangle$ (WECKER; SVORE, 2014).

Para facilitar a programação de seus computadores quânticos, empresas têm desenvolvido kits de desenvolvimento de software (SDK – *Software Development Kit*) como o Qiskit da IBM, o Cirq da Google e o Forest da Rigetti.

2.6.1 Qiskit e OpenQASM

Os computadores quânticos da IBM executam conjuntos de instruções escritos na linguagem OpenQASM (CROSS et al., 2017) que é uma das variantes da linguagem QASM. Existem várias variantes da linguagem QASM e suas diferenças têm levado a busca por uma linguagem de baixo nível que seja independente do hardware (KHAMMASSI et al., 2018).

A IBM oferece a plataforma IBM Q Experience para programar em OpenQASM e executar programas escritos nessa linguagem em seus simuladores e computadores quânticos. Pode-se ter acesso a simuladores de até 5000 qubits e a computadores quânticos com processadores Hummingbird de 65 qubits. Usuários comuns, aqueles que possuem uma conta simples na plataforma, têm acesso a computadores quânticos com processadores de 5 qubits. Na plataforma IBM Q Experience, encontram-se duas opções para escrever programas em OpenQASM: o IBM Q Composer e o IBM Q Lab. O IBM Q Composer oferece uma facilidade muito grande para criar circuitos quânticos com a opção de selecionar portas quânticas, arrastá-las e aplicá-las sobre os qubits desejados. O programa em OpenQASM associado ao circuito quântico construído

é mostrado em uma janela lateral. A Figura 18 mostra a interface do IBM Q Composer.

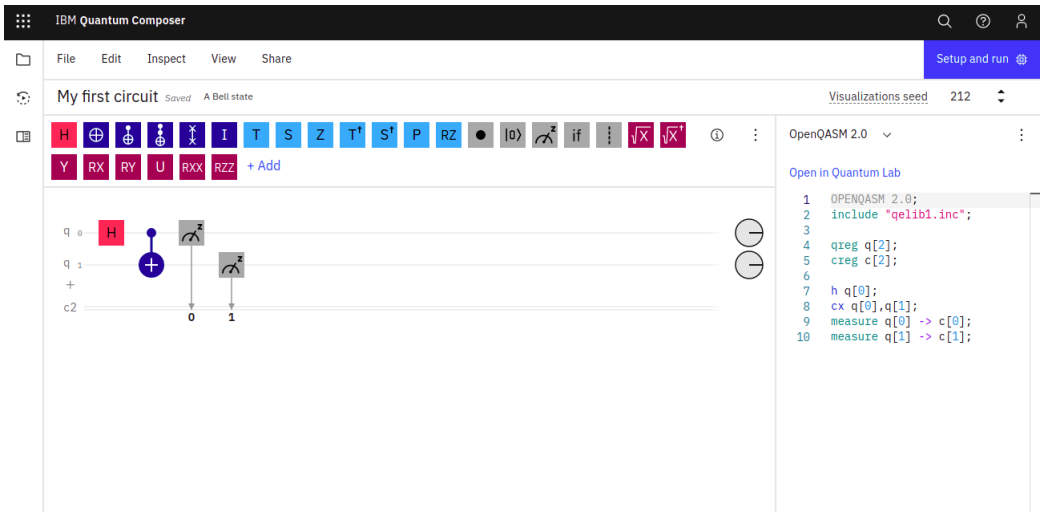

Figura 18 – Interface do IBM Q Composer para o desenvolvimento de circuitos quânticos e programas em OpenQASM (IBM, 2021).

O IBM Q Lab oferece um ambiente de programação em Python através do Jupyter Notebook sem a necessidade de instalação de programas no computador do usuário. A IBM também oferece o Qiskit para ser instalado na máquina do usuário e possui um ambiente de programação em Python através do Jupyter Notebook semelhante ao IBM Q Lab. O Qiskit também está disponível em JavaScript e Swift.

Ecossistemas de software são grandes coleções de pacotes de software que possuem uma relação de dependência e evoluem de forma conjunta. O ecossistema do Qiskit inclui quatro frameworks: Terra, Aer, Ignis e Aqua. O Qiskit Terra é o framework base que fornece as ferramentas para o desenvolvimento e a otimização de programas quânticos. O Qiskit Terra também é responsável pelo gerenciamento da execução de programas em batch através de acesso remoto. No Qiskit Terra estão presentes os módulos: qiskit.circuit, qiskit.pulse, qiskit.transpiler, qiskit.providers, qiskit.quantum_info e qiskit.visualization. O Qiskit Aer inclui três simuladores de alta performance: QasmSimulator, StatevectorSimulator e UnitarySimulator. Esses simuladores são utilizados no estudo dos erros que ocorrem durante a execução em computadores quânticos reais. O Qiskit Ignis também é utilizado no estudo dos erros durante a execução de programas quânticos e possibilita projetar códigos de correção de erros e caracterizar os erros de execução. O Qiskt Ignis contém as bibliotecas qiskit.ignis.characterization, qiskit.ignis.verification e qiskit.ignis.mitigation. O Qiskit Aqua é o framework utilizado no desenvolvimento de programas quânticos para aplicações que requerem o uso de aceleradores quânticos para a realização de tarefas específicas.

A instalação do SDK Qiskit é um procedimento bastante simples. O primeiro passo consiste na instalação do Anaconda Python. Tendo instalado o Anaconda Python, pode-se instalar o Qiskit. Para isso, bastar executar o comando

```
pip install qiskit
```

no terminal dos sistemas operacionais Linux ou MacOS ou no Anaconda Prompt para o sistema operacional Windows.

2.6.2 Forest e Quil

A Rigetti Computing permite o acesso aos seus processadores quânticos através de sua própria plataforma de computação quântica em nuvem chamada de *Quantum Cloud Services* (QCS). A linguagem utilizada para programar os computadores quânticos da Rigetti é o Quil (*Quantum Instruction Language*). Através da biblioteca pyQuil, programas escritos em linguagem Python podem ser traduzidos para Quil e executados nos simuladores ou computadores quânticos da Rigetti. O desenvolvimento de algoritmos quânticos para os computadores e simuladores quânticos da Rigetti é facilitado com o uso do SDK Forest. O Forest inclui o compilador quilc e o simulador QVM (*Quantum Virtual Machine*).

O uso da biblioteca pyQuil e do SDK Forest requer a versão 3.7 ou superior do Python. Portanto, deve-se verificar a versão instalada no computador antes de iniciar os procedimentos de instalação que podem ser encontrados na documentação da biblioteca pyQuil. A instalação da biblioteca pyQuil é feita com a execução da linha de comando

```
pip install pyquil
```

no terminal dos sistemas operacionais Linux ou MacOS ou no prompt de comando do sistema operacional Windows. Após a instalação do pyQuil, deve-se fazer o download do SDK Forest. No Windows, a instalação do SDK Forest resume-se em executar o arquivo forest-sdk.msi e seguir as instruções. No macOS, a instalação requer a execução do arquivo forest-sdk.dmg seguido da execução do arquivo forest-sdk.pkg. No Linux, para a distribuição .deb executa-se:

```
tar -xf forest-sdk-linux-deb.tar.bz2
cd forest-sdk-linux-deb
sudo ./forest-sdk-linux-deb.run
```

Para a distribuição .rmp executa-se:

```
tar -xf forest-sdk-linux-rpm.tar.bz2
cd forest-sdk-linux-rpm
sudo ./forest-sdk-linux-rpm.run
```

Para maiores detalhes sobre os procedimentos de instalação, pode-se consultar o Guia de Instalação da biblioteca pyQuil e do SDK Forest.

2.6.3 Cirq

O Cirq é uma biblioteca para a linguagem Python que permite escrever algoritmos para programar os computadores quânticos da Google. Através do Cirq, pode-se simular um processador quântico no computador do usuário. O acesso aos processadores quânticos da Google através do Cirq é possível, porém o acesso é restrito. A Google disponibiliza um amplo material dedicado a ensino de computação quântica em quantumai.google.

A instalação do Cirq não apresenta dificuldades. Antes de instalar o Cirq, certifique-se de que a versão 3.7 ou superior do Python está instalada em seu computador. Para instalar o Cirq, basta executar os comandos:

```
python -m pip install -upgrade pip
python -m pip install cirq
```

Maiores detalhes sobre o procedimento de instalação pode ser encontrados no Guia de Instalação do SDK Cirq.

2.7 Simuladores de computadores quânticos

Escalabilidade é uma propriedade de um sistema computacional, que pode ser hardware ou software, que determina a variação da potência computacional em função da quantidade de recursos computacionais. Quando a potência computacional e a quantidade de recursos variam de forma proporcional, o sistema é classificado como um sistema escalável. Para softwares, a escalabilidade pode ser chamada de eficiência de paralelização. Uma das formas de medir a escalabilidade de um software é o *speedup* que é a razão entre o tempo de execução de um programa de forma sequencial e o tempo de execução de forma paralela com o uso de N processadores. Se o valor do *speedup* é N, então temos um *speedup* linear. Quando o valor do *speedup* é maior do que N, temos um *speedup* superlinear.

Na computação quântica, um sistema escalável é aquele que é capaz de operar de modo estável (SHIBAGAKI, 2020). Atualmente, os computadores quânticos ainda não são capazes de evitar falhas em seu funcionamento e operar de modo estável com uma quantidade considerável de qubits. Essas falhas podem ocorrer devido ao próprio funcionamento do sistema ou serem causadas por ruídos, que são interferências do ambiente, e resultam em problemas de decoerência. Decoerência é a perda da informação quântica durante a operação do sistema resultando em erros de cálculo. Essa fase atual da computação quântica é chamada de era NISQ (*Noisy Intermediate-Scale Quantum*) onde os processadores quânticos possuem até algumas centenas de qubits e não são tolerantes a falhas. O fim da era NISQ é previsto com o desenvolvimento de processadores quânticos com alguns milhares de qubits e algoritmos de correção de erros eficazes. Então, algoritmos como o algoritmo de Shor poderão ser implementados e criptografias RSA poderão ser quebradas.

Na era NISQ, simuladores de computadores quânticos são de grande importância para o estudo e desenvolvimento de algoritmos quânticos. Simuladores são desenvolvidos desde a década de 90 quando algoritmos quânticos como o algoritmo de Shor e o algoritmo de Grover mostraram o poder computacional da computação quântica. Porém, a simulação de computadores quânticos requer um aumento exponencial de operações e capacidade de armazenamento para cada qubit adicionado ao circuito quântico o que limita a quantidade de qubits que pode ser simulada em um computador clássico. Embora técnicas de programação paralela e GPUs sejam

utilizadas pelos simuladores (AMARIUTEI; CARAIMAN, 2011; AVILA et al., 2020; EFTHYMIOU et al., 2021; HENG; KIM; HAN, 2020), a quantidade de qubits que pode ser simulada ainda é pequena. O limite de 50 qubits anteriormente estabelecido (BOIXO et al., 2018), foi superado pela simulação de circuitos quânticos com 56 qubits (PEDNAULT et al., 2017) e 64 qubits (CHEN et al., 2018).

O paralelismo quântico refere-se a capacidade do qubit estar nos estados $|0\rangle$ e $|1\rangle$ ao mesmo tempo e garante o expressivo poder computacional dos computadores quânticos quando comparados aos computadores clássicos. Um sistema com apenas um qubit contém uma base com 2^1 estados que são so estados $|0\rangle$ e $|1\rangle$. Qualquer operação sobre o estado desse sistema é realizada através do produto de uma matriz de dimensão 2×2, que representa uma porta quântica, pelo vetor que representa o estado do sistema. Para um sistema com dez qubits, há 2^{10} = 1024 estados. Qualquer operação sobre o estado desse sistema envolve o produto de uma matriz de dimensão 1024×1024 por um vetor com 1024 elementos. Esse é o crescimento exponencial mencionado no parágrafo anterior. Uma quantidade muito grande de memória é preciso para armazenar todos esses dados nas operações matemáticas realizadas. Deve-se considerar também que circuitos quânticos complexos contêm uma grande quantidade de portas quânticas que implicam em uma grande quantidade de operações matemáticas que devem ser realizadas.

3 Metodologia de ensino

Neste capítulo, será apresentada a metodologia de desenvolvimento empregada na produção do material didático fruto desse trabalho. Essa metodologia é comparada à metodologia de outros dois cursos de ensino de computação quântica.

No artigo *"Teaching Quantum Computing to High School Students"* (HUGHES et al., 2020), é proposto um curso de computação quântica para estudantes do nível médio com idades entre 15 e 18 anos (PERRY et al., 2019). Esse curso requer conhecimentos básicos de eletricidade, magnetismo e ondas. Conhecimentos sobre efeito fotoelétrico e dualidade onda-partícula, que são abordados em um curso de introdução à física moderna, não são necessários, mas indicados para um melhor desempenho nesse curso. A Figura 19 ilustra os tópicos abordados no curso e a sequência que deve ser seguida.

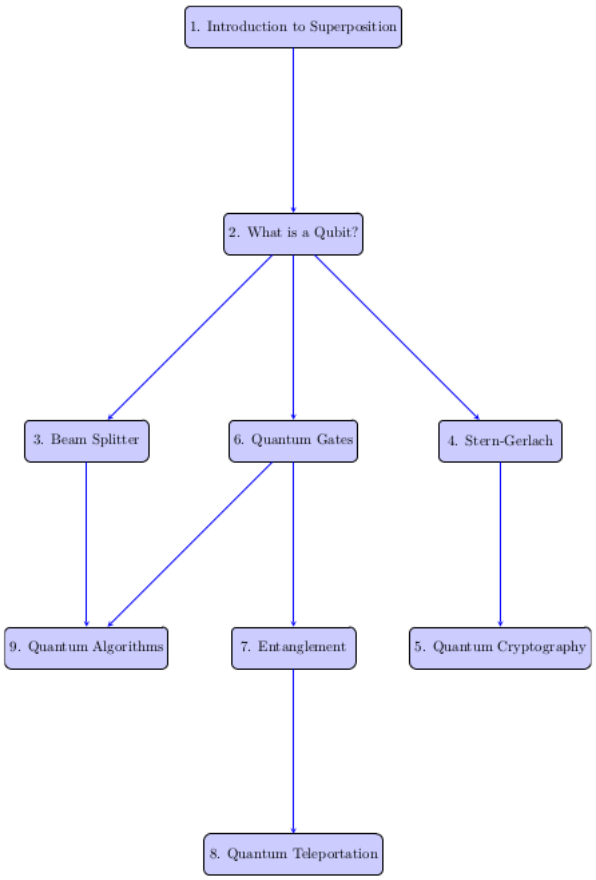

Figura 19 – Tópicos do curso *Quantum Computing as a High School Module* (HUGHES et al., 2020).

No artigo *"First experiences teaching Quantum Computing"* (CARRASCAL; BARRIO; BOTELLA, 2021), os autores destacam a relevância do problema ao citar: *"It has been claimed that it is possible to effectively teach quantum computing to undergraduate students without a physics background by means of computer programming"*. Uma metodologia de en-

sino é proposta. Essa metodologia é dividida em quatro estágios. No primeiro estágio, o conceito de qubit é apresentado aos estudantes. Conhecimentos de álgebra linear são necessários para a compreensão da representação vetorial de qubits. O processo de medição do estado de um qubit também é abordado nesse primeiro estágio. Ao estudante, é proposto desenvolver um programa que simula um qubit e o processo de medição. No segundo estágio, o estudante terá contato com a plataforma IBM Q Experience e utilizará a interface IBM Q Composer para o desenvolvimento de algoritmos quânticos. Nesse estágio, conceitos como superposição e estrelaçamento são apresentados aos estudantes. De fato, a interface IBM Q Composer é reconhecida por outros trabalhos como um eficiente recurso educacional para o ensino de computação quântica (JESUS et al., 2021; RABELO; COSTA, 2018). No terceiro estágio, o estudante utilizará o SDK Qiskit para o desenvolvimento de funções que implementam portas lógicas com o uso de portas quânticas. No quarto e último estágio, alguns dos algoritmos quânticos mais conhecidos são apresentados aos estudantes. Entre esses algoritmos estão o algoritmo de Deutsch-Josza, o algoritmo de Bernstein-Vazirani, o algoritmo de Simon, a tranformada quântica de Fourier, o algoritmo de Grover e o algoritmo de Shor. Esse último estágio está condicionado ao nível de conhecimento matemático do estudante.

O objetivo do tutorial desenvolvido nesse trabalho é introduzir conceitos básicos de computação quântica que possibilitem aos estudantes produzir algoritmos quânticos simples. Embora abordados em muitos tutoriais de ensino (ASFAW et al., 2020), alguns dos algoritmos conhecidos da computação quântica como o algoritmo de Grover e o algoritmo de Shor requerem conhecimentos mais avançados o que torna difiícil o ensino desses algoritmos para estudantes de cursos de computação que não possuem conhecimentos avançados em física e matemática.

Ao comparar a metodologia do tutorial desenvolvido nesse trabalho com a metodologia do curso *"Quantum Computing as a High School Module"* (PERRY et al., 2019), conclui-se que o nosso objetivo corresponde a atingir o item *Quantum Algorithms* na Figura 19. De acordo com essa figura, antes de estudar o tópico *Quantum Algorithms*, é preciso estudar os tópicos *Introduction to Superposition*, *What is Qubit?* e *Quantum Gates* ou *Beam Splitter*. Com exceção do tópico *Beam Splitter*, esses tópicos também são abordados no tutorial desenvolvido nesse trabalho. Embora tenha-se procurado evitar conceitos de mecânica quântica e o uso de linguagem matemática no desenvolvimento do tutorial, não foi possível evitar os conceitos de estado quântico e superposição de estados. Na verdade, a palavra superposição foi evitada no texto, mas o seu conceito está impliícito na definição de qubit, numa tentativa de facilitar o entendimento. A Figura 20 mostra a metodologia de ensino adotada para o desenvolvimento do tutorial proposto nesse trabalho.

O tutorial é dividido em: Introdução, Kits de Desenvolvimento de Software, Algoritmo 1 – O que é qubit?, Algoritmo 2 – Porta de Hadamard, Algoritmo 3 – Porta de Pauli X, Algoritmo 4 – Circuitos com dois ou mais qubits, Algoritmo 5 – Porta CNOT, Algoritmo 6 – Porta

TOFFOLI, Algoritmo 7 – Estado de Bell, Algoritmo 8 – *Full Adder* quântico, Algoritmo 9 – *Full Subtractor* quântico e Referências. Na Introdução pretende-se motivar o estudante para iniciar os estudos em computação quântica. Para isso, o estado da arte da computação quântica é brevemente apresentado. O próximo capítulo trata dos Kits de Desenvolvimento de Software. Nesse capítulo, os SDKs Qiskit, Forest e Cirq, que são utilizados ao longo do tutorial, são descritos.

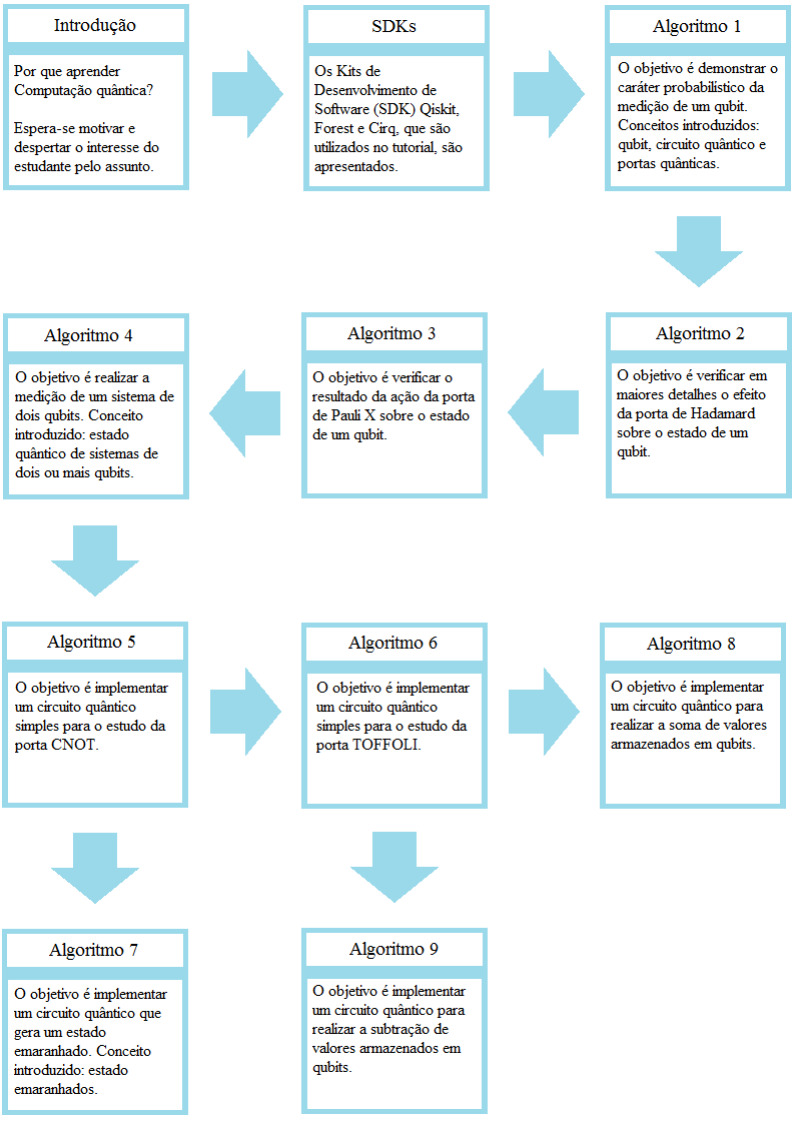

Figura 20 – Metodologia de ensino do Tutorial de Programação de Computadores Quânticos.

O primeiro algoritmo é apresentado no capítulo 3. O objetivo desse capítulo é fazer com que o estudante possa entender o que é um qubit e sua diferença em relação a um bit clássico. A definição de qubit requer o conceito de estado quântico. Esse conceito não é omitido, porém não é aprofundado. Ao estudante, é passada a informação de que um qubit pode estar em dois estados sendo esses estados representados pela notação de Dirac que é usual na mecânica quântica. Para familiarizar o estudante com essa notação é feita uma analogia com a representação de um vetor em um espaço bidimensional com componentes na base $\{\hat{x},\hat{y}\}$. O conceito de

circuito quântico e sua relação com o algoritmo quântico são introduzidos nesse capítulo. Ao executar o primeiro algoritmo, o estudante deve verificar que o resultado pode ser 0 ou 1. Para isso, deve-se implementar uma porta de Hadamard o que faz necessário definir portas quânticas. Para facilitar a compreensão desse conceito, é feita uma analogia entre portas lógicas e portas quânticas. O algoritmo 1 é um dos mais extensos porque são introduzidas as definições de qubit, circuito quântico e portas quânticas. As linhas de comando necessárias para a execução do algoritmo também são explicadas com maiores detalhes por se tratar do primeiro algoritmo a ser executado pelo estudante.

Embora a porta de Hadamard tenha sido utilizada no algoritmo 1, o seu efeito sobre o estado do qubit não pode ser observado no primeiro algoritmo. Assim, o segundo algoritmo dedica-se a permitir que o estudante possa visualizar a ação da porta de Hadamard sobre o estado de um quibit. No terceiro algoritmo é apresentada a porta de Pauli X. Essa porta quântica é associada à porta lógica NOT. Através da execução do algoritmo o estudante poderá verificar a alteração no estado do qubit que é consequência da ação da porta de Pauli X. Nos algoritmos anteriores, o estudante trabalhou com apenas um qubit. No algoritmo 4, o estudante irá executar um algoritmo com dois qubits. Estados quânticos de sistemas de dois e três qubits são apresentados nesse capítulo. O algoritmo 5 tem como objetivo apresentar ao estudante a porta quântica CNOT. Uma comparação entre a porta CNOT e a porta lógica XOR é realizada. Após o algoritmo 5, o estudante terá condições de estudar o algoritmo 7 e poderá estudar o algoritmo 7 e depois retornar para a sequência e estudar o algoritmo 6, seguir a sequência de ensino ou, ainda, deixar de estudar o algoritmo 7. No algoritmo 6, a porta TOFFOLI é apresentada. Após o algoritmo 6, o estudante estará preparado para estudar o algoritmo 8 ou o algoritmo 9.

No algoritmo 7, o estudante terá o seu primeiro contato com um estado emaranhado. Esse estado também é conhecido como estado de Bell. Esse algoritmo é chamado, por alguns tutoriais de ensino, de *Hello Qubits* ou *Hello Quantum World*, ou seja, é o equivalente ao programa *Hello World* que o estudante aprende ao iniciar o estudo de uma linguagem. Porém, esse algoritmo não é tão simples. O estudante precisa de muitos conceitos de computação quântica para a compreensão desse algoritmo. Nesse capítulo, o estudante precisará realizar alguns cálculos simples para verificar que não há solução para a Equação A.5. Essa é uma situação na qual, embora não seja desejado, o uso de linguagem matemática não pode ser evitado. Como mostra a Figura 20, esse algoritmo pode não ser incluído na sequencia de algoritmos a ser ensinado caso não seja desejado entrar em assuntos um pouco mais complexo como estados emaranhados.

O algoritmo 8 apresenta uma implementação de um circuito quântico para um somador. A operação de soma pode ser realizada por um circuito quântico que possui apenas portas CNOT e TOFFOLI. As funções dos qubits desse circuito na operação de soma são comparadas às funções dos bits de um somador clássico. Nesse capítulo discute-se o resultado da ação de cada porta quântica sobre o estado dos qubits do circuito para ajudar o estudante a entender o funcionamento do circuito. No algoritmo 9, o estudante irá aprender a versão quântica de um

subtrator. Procedimento semelhante ao realizado no ensino do somador quântico é realizado para explicar o funcionamento do subtrator quântico.

Ao contrário do curso proposto por Hughes e seus colaboradores, o tutorial desenvolvido nesse trabalho não requer conhecimentos de eletricidade, magnetismo e ondas. Ao estudante, é suficiente conhecimentos básicos de matemática que incluem números complexos e representação vetorial. Porém conhecimentos básicos de programação em linguagem Python são necessários. A fim de minimizar esses conhecimentos, optou-se por não utilizar bibliotecas como Matplotlib que poderiam melhorar a forma como os resultados são apresentados.

4 Resultados de simulações de algoritmos quânticos

Nesse capítulo serão apresentados os resultados das simulações de algoritmos quânticos que implementam os circuitos quânticos das Figuras 9, 10 e 17. Para o circuito quântico que gera um estado emaranhado e para o circuito quântico *Full Adder*, a implementação é realizada para os SDKs Qiskit, Forest e Cirq. Por se tratar de um circuito um pouco mais extenso, a implementação do circuito quântico para o algoritmo de Grover é realizada apenas com o SDK Qiskit.

4.1 Implementação em Python com o SDK Qiskit do circuito quântico que gera um estado emaranhado

Para realizar a medição do estado de um qubit, todo circuito quântico deve conter, pelo menos, dois registradores sendo um quântico e outro clássico que é utilizado para armazenar o resultado da medida realizada. No Qiskit, é necessário importar a biblioteca *QuantumCircuit* para instanciar o circuito quântico e as bibliotecas *QuantumRegister* e *ClassicalRegister* para instanciar os registradores quântico e clássico. Essas bibliotecas são importadas por:

```
from qiskit import QuantumCircuit,
from qiskit import QuantumRegister
```

e

```
from qiskit import ClassicalRegister.
```

Para instanciar o registrador quântico utiliza-se o construtor *QuantumRegister()* ao qual deve ser fornecido a quantidade de qubits. Pode-se também fornecer um nome utilizado na identificação do registrador, entretanto esse nome não é um argumento obrigatório. Quando o nome do registrador é omitido, o registrador passa a ser identificado pela letra *q* seguida de um índice. O registrador clássico é instanciado pelo construtor *ClassicalRegister()* ao qual deve ser fornecido a quantidade de bits e, também como opção, o nome do registrador clássico. Na ausência do nome, os registradores clássico são identificados pela letra *c* seguida de um índice. O construtor *QuantumCircuit()* é utilizado para instanciar um circuito quântico. Ao instanciar o circuito qântico, pode-se fornecer os registradores que devem estar presentes no circuito ou implementar esses registradores posteriormente com o uso do método *add_register()*. Os comandos para instanciar os registradores quânticos e clássicos e o circuito quântico em linguagem Python utilizando o SDK Qiskit são:

```
qr = QuantumRegister(2),
```

`cr = ClassicalRegister(2)`
e
`qc = QuantumCircuit(qr,cr).`

A porta de Hadamard é implementada no algoritmo por:
`qc.h(qr[0]).`
A porta CNOT é implementada por:
`qc.cx(qr[0],qr[1]).`
Os qubits qr[0] e qr[1] são medidos através do método *measure()* implementado por:
`qc.measure(qr,cr).`

Após a etapa de construção do circuito quântico, vem a etapa de execução. Primeiro, deve-se importar o *framework Aer* que possibilitará o uso do método *get_backend()* para escolher entre executar o programa no computador do usuário, em um dos simuladores da IBM ou em um dos computadores quânticos da IBM. Como argumento do método *get_backend()*, existem as opções *qasm_simulator*, *statevector_simulator* e *unitary_simulator*. Após a definição do *backend*, deve-se importar a biblioteca *execute* para utilizar o método *execute()* para o qual devem ser fornecidos, como argumentos, o circuito quântico e o *backend*. O número de vezes que a execução do programa deve ser repetida também pode ser fornecido ao método *execute()* por meio da variável *shots*. O resultado é obtido através do método *result()*. A contagem de cada tipo de resultado é realizada com o método *get_counts()*. O algoritmo completo é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(2)
cr = ClassicalRegister(2)
qc = QuantumCircuit(qr,cr)
qc.h(qr[0])
qc.cx(qr[0],qr[1])
qc.measure(qr,cr)
backend = Aer.get_backend('qasm_simulator')
job = execute(qc,backend,shots=100)
resultado = job.result()
contagem = resultado.get_counts()
print(qc.qasm())
print(contagem)
```

Um dos possíveis resultados da execução do programa acime é {'11': 50, '00': 50}. Uma

medição desses qubits pode retornar o resultado 00 ou 11. O número total de medições é o valor atribuído à variável *shots* que nesse algoritmo foi 100. O resultado 11 foi obtido em 50 medições e o resultado 00 também foi obtido em 50 medições.

Para converter o algoritmo em Python para OpenQASM, utiliza-se o método *qasm()*.A primeira linha em todo programa em OpenQASM deve conter o comando OPENQASM seguido pela sua versão. Na segunda linha do programa, temos a inclusão do arquivo *qelib1.inc*. Nas linhas seguintes, estão os comandos para instanciar os registradores quânticos e clássicos, os comandos para a implementação das portas de Hadamard e CNOT e o comando para medir o estado de um qubit. Cross *et al.* (CROSS et al., 2017) apresentam a versão 2.0 da linguagem OpenQASM com maiores detalhes.

O resultado da conversão do algoritmo para a linguagem OpenQASM é:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q0[2];
creg c0[2];
h q0[0];
cx q0[0],q0[1];
measure q0[0] -> c0[0];
measure q0[1] -> c0[1];
```

O conteúdo apresentado nessa seção, dedicado à implementação do circuito quântico que gera o estado de Bell através do SDK Qiskit, baseou-se no trabalho de Renato Portugal e Franklin L. Marquezino (PORTUGAL; MARQUEZINO, 2019).

4.2 Implementação em Python com o SDK Forest do circuito que gera um estado emaranhado

No Forest, a biblioteca *get_qc* é necessária para instanciar uma máquina quântica virtual que é utilizada na execução do programa. A biblioteca *Program* é necessária para instanciar o circuito quântico. O módulo *gates* contém as bibliotecas necessárias para cada uma das portas quânticas utilizadas no circuito. Os registradores clássicos onde os resultados das medidas são armazenados são instanciados com o método *Declare()* que requer a biblioteca *Declare* que está presenta no módulo *quilbase*. As linhas de comando iniciais do programa, que são utilizadas para importar as bibliotecas necessárias, são:

```
from pyquil import get_qc, Program,
form pyquil.gates import H, CNOT, MEASURE
```

e

```
from pyquil.quilbase import Declare.
```

O circuito quântico é instanciado por:

```
p = Program().
```

A porta de Hadamard e a porta CNOT são implementadas por:

```
p += H(0)
```

e

```
p += CNOT(0,1).
```

Os dois registradores clássicos necessários para armazenar os dados das medições realizadas são instanciados por:

```
ro = p.declare('ro', 'BIT', 2)
```

A medida do estado quântico de um qubit é realizada com o método *MEASURE()* para o qual deve ser passado um primeiro argumento, que especifica o qubit a ser medido, e um segundo argumento, que especifica o registrador onde o resultado será armazenado. Os medidores quânticos são implementados por:

```
p += MEASURE(0, ro[0])
```

e

```
p += MEASURE(1, ro[1]).
```

Um maior número de execuções pode ser determinado pelo método *wrap_in_numshots_loop()*. Para executar o programa em um simulador ou em um processador quântico através da plataforma QCS, faz-se necessário definir uma máquina quântica virtual. Nesse algoritmo, uma QVM de dois qubits é instanciada por *get_qc("2q-qvm")*. O programa é compilado pelo método *compile()* e executado pelo método *run()*. A leitura dos resultados é feita com o método *readout_data_get()*. O programa completo é mostrado a seguir:

```
from pyquil import get_qc, Program
from pyquil.gates import H, CNOT, MEASURE
from pyquil.quilbase import Declare
p = Program()
p += H(0)
p += CNOT(0,1)
ro = p.declare('ro', 'BIT', 2)
p += MEASURE(0, ro[0])
p += MEASURE(1, ro[1])
p.wrap_in_numshots_loop(10)
qc = get_qc("2q-qvm")
executable = qc.compile(p)
result = qc.run(executable)
print(result.readout_data.get('ro'))
print(p)
```

O resultado que deve ser obtido com a execução do algoritmo acima é [[0 0] [1 1] [0 0]

[1 1] [0 0] [1 1] [1 1] [1 1] [1 1] [0 0]]. Cada medida deve resultar em um estado $|00\rangle$ ou $|11\rangle$. Como o número de execuções especificado pelo método *wrap_in_numshots_loop()* é 10, temos 10 resultados diferentes. O comando *print()* imprime o programa, que foi escrito em Python, na linguagem Quil. O resultado é:

```
H 0
CNOT 0 1
DECLARE ro BIT[2]
MEASURE 0 ro[0]
MEASURE 1 ro[1]
```

4.3 Implementação em Python com o SDK Cirq do circuito que gera um estado emaranhado

No Cirq, o circuito quântico é instanciado pelo método *cirq.Circuit()*. Os métodos *circ.NamedQubit()*, *circ.LineQubit()* e *circ.GridQubit()* podem ser usados para instanciar os registradores quânticos. No método *circ.NamedQubit()*, o registrador quântico é identificado através da string passada como argumento para essa função. No método *circ.LineQubit()*, o registrador quântico é identificado pelo valor passado como argumento para essa função e, esse registrador, pertence a um array de qubits. No método *circ.GridQubit()*, o registrador quântico declarado pertence a uma rede de registradores e sua identificação é feita por dois números inteiros. Na implementação em Python com o SDK Cirq do circuito que gera um estado emaranhado, será utilizado o método *circ.NamedQubit()*.

O circuito quântico é instanciado por:

```
qc = cirq.Circuit().
```

Os dois qubits do circuito quântico são implementados no algoritmo por:

```
qr0 = cirq.NamedQubit('q[0]')
```

e

```
qr1 = cirq.NamedQubit('q[1]')
```

. Sobre o qubit qr[0] deve ser aplicada uma porta de Hadamard por:

```
qc.append(cirq.H(qr0)).
```

A porta CNOT é implementada no algoritmo por:

```
qc.append(cirq.CNOT(qr0,qr1)),
```

onde *qr0* é o qubit de controle e *qr1* é o qubit alvo. As medições dos qubits são realizadas por:

```
qc.append(cirq.measure(qr0,key = 'q[0]'))
```

e

```
qc.append(cirq.measure(qr1,key = 'q[1]')).
```

O simulador é instanciado por:

```
simulador = cirq.Simulator().
```

O comando para executar o programa é:
`resultado = simulador.run(qc,repetitions=10).`
O algoritmo completo é mostrado a seguir:

```
import cirq
qc = cirq.Circuit()
qr0 = cirq.NamedQubit('q[0]')
qr1 = cirq.NamedQubit('q[1]')
qc.append(cirq.H(qr0))
qc.append(cirq.CNOT(qr0,qr1))
qc.append(cirq.measure(qr0,key = 'q[0]'))
qc.append(cirq.measure(qr1,key = 'q[1]'))
simulador = cirq.Simulator()
resultado = simulador.run(qc,repetitions=10)
print(resultado)
print(qc.to_qasm())
```

Um dos possíveis resultados para a execução desse algoritmo é:

```
q[0]=0000011010
q[1]=0000011010
```

Nota-se que os resultados são 00 ou 11 como esperado. O método *to_qasm()* é utilizado para imprimir o algoritmo em linguagem OpenQASM. O resultado é:

```
// Generated from Cirq v0.13.1
OPENQASM 2.0;
include "qelib1.inc";
// Qubits: [q[0], q[1]]
qreg q[2];
creg m0[1]; // Measurement: q[0]
creg m1[1]; // Measurement: q[1]
h q[0];
cx q[0],q[1];
measure q[0] -> m0[0];
measure q[1] -> m1[0];
```

4.4 Implementação em Python com o SDK Qiskit do circuito quântico *Full Adder*

As bibliotecas necessária para a implementação do circuito quântico *Full Adder* são as mesmas utilizadas no algoritmo anterior. Essas bibliotecas devem ser importadas por:
`from qiskit import QuantumCircuit,`

```
from qiskit import QuantumRegister,
from qiskit import ClassicalRegister,
from qiskit import Aer
```
e
```
from qiskit import execute.
```
São utilizados um registrador quântico com quatro qubits e um registrador clássico com dois bits. Os registradores e o circuito quântico são instanciados por:
```
qr = QuantumRegister(4),
cr = ClassicalRegister(2)
```
e
```
qc = QuantumCircuit(qr,cr).
```
O estado inicial dos qubits é o estado $|0\rangle$. Para especificar os estados de entrada no circuito quântico *Full Adder*, deve-se aplicar a porta de Pauli X. Na Figura 10, a porta de Pauli X foi aplicada aos qubits qr[0] e qr[2], portanto o circuito quântico corresponde a um circuito *Full Adder* clássico que realiza a soma dos bits 1 e 0 com *carry in* 1 o que resultaria no valor 0 para o bit de saída e 1 para o bit de *carry out*, ou seja, o valor binário 10. As portas de Pauli X são implementadas no circuito através de:
```
qc.x(qr[0])
```
e
```
qc.x(qr[2]).
```
A porta TOFFOLI com dois qubits de controle é implementada pelo método *ccx()*. Os dois primeiros qubits passados como argumento para esse método são os qubits de controle e o terceiro qubit é o qubit alvo. Após as portas de Pauli X que foram utilizadas para modificar o estado inicial dos qubits qr[0] e qr[2], uma porta TOFFOLI é implementada tendo os qubits qr[0] e qr[1] como qubits de controle e qr[3] como qubit alvo. Essa porta quântica é implementada por:
```
qc.ccx(qr[0],qr[1],qr[3]).
```

Após a primeira porta TOFFOLI, uma porta CNOT é implementada no circuito. Essa porta CNOT tem qr[0] como qubit de controle e qr[1] como qubit alvo. Então, outra porta TOFFOLI é implementada sendo qr[1] e qr[2] os qubis de controle e qr[3] o qubit alvo. Duas portas CNOT são implementadas no final do circuito, a primeira tendo qr[1] como qubit de controle e qr[2] como qubit alvo e a segunda tendo qr[0] como qubit de controle e qr[1] como qubit alvo. Essas portas quânticas são implementadas pelas linhas de comando:
```
qc.cx(qr[0],qr[1]),
qc.ccx(qr[1],qr[2],qr[3]),
qc.cx(qr[1],qr[2])
```
e
```
qc.cx(qr[0],qr[1]).
```
As medições são realizadas sobre os qubits qr[2], que corresponde ao bit de saída, e qr[3], que corresponde ao bit *carry out*. O resultado é salvo nos bits cr[0] e cr[1] do registrador clássico.

Os medidores quânticos são implementados por:
`qc.measure(qr[2],cr[0])`
e
`qc.measure(qr[3],cr[1]).`
O método *qasm()* é utilizado para a conversão do código em linguagem Python para a linguagem OpenQASM. Utiliza-se o método *get_backend()* para selecionar o simulador e o método *execute()* para determinar a execução do programa. Os resultado são extraídos pelo método *result()* e separados por resultado obtido através do método *get_counts()*. Para uma melhor visualização, pode utilizar a função *plot_histogram()*. O algoritmo completo é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(4)
cr = ClassicalRegister(2)
qc = QuantumCircuit(qr,cr)
qc.x(qr[0])
qc.x(qr[2])
qc.ccx(qr[0],qr[1],qr[3])
qc.cx(qr[0],qr[1])
qc.ccx(qr[1],qr[2],qr[3])
qc.cx(qr[1],qr[2])
qc.cx(qr[0],qr[1])
qc.measure(qr[2],cr[0])
qc.measure(qr[3],cr[1])
backend = Aer.get_backend('qasm_simulator')
job = execute(qc,backend,shots=100)
resultado = job.result()
contagem = resultado.get_counts()
print(qc.qasm())
print(contagem)
```

O resultado da simulação do *Full Adder* quântico é {'10': 100}. O resultado mostra que o estado $|10\rangle$, que corresponde ao valor binário 10, foi obtido em todas as medições realizadas. O reasultado da conversão de Pyhton para OpenQASM é:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q0[4];
```

```
creg c0[2];
x q0[0];
x q0[2];
ccx q0[0],q0[1],q0[3];
cx q0[0],q0[1];
ccx q0[1],q0[2],q0[3];
cx q0[1],q0[2];
cx q0[0],q0[1];
measure q0[2] -> c0[0];
measure q0[3] -> c0[1];
```

4.5 Implementação em Python com o SDK Forest do circuito quântico *Full Adder*

Para implementar o *Full Adder* quântico em linguagem Python com o SDK Forest, é preciso importar as biblioteca *Program* para instanciar o circuito quântico por:

```
p = Program().
```

As duas portas de Pauli X necessárias para estabelecer os valores de entrada do somador são implementadas por:

```
p += X(0)
```

e

```
p += X(2).
```

O uso de portas quânticas requer que as bibliotecas presentes no módulo *gates* sejam importadas. No Forest, portas TOFFOLI são implementadas pelo método *CCNOT()* para o qual é passado os índices que determinam os qubits de controle e o qubit alvo. A primeira porta TOFFOLI é implementada por:

```
p += CCNOT(0,1,3).
```

Essa porta quântica não altera o estado do seu qubit alvo porque o qubit qr[1] está no estado $|0\rangle$. Em seguida, é implementado no circuito uma porta CNOT através de:

```
p += CNOT(0,1).
```

Essa porta quântica altera o estado de qr[1] para $|1\rangle$. A segunda porta TOFFOLI é implementada por:

```
p += CCNOT(1,2,3).
```

O qubit ar[3] tem o seu estado alterado para $|1\rangle$ pela ação dessa porta TOFFOLI. Temos aqui, todos os qubits no estado $|1\rangle$. A seguir, são implementadas duas portas CNOT em sequência por:

```
p += CNOT(1,2)
```

e

```
p += CNOT(0,1).
```

A primeira faz com que o qubit qr[2] retorne para o estado $|0\rangle$ e a segunda faz com que o qubit qr[1] também retorne para esse estado. Os registradores clássicos são instanciados por:
`ro = p.declare('ro', 'BIT', 2).`
O método *declare()* requer o uso da biblioteca *Declare* que pertence ao módulo *quilbase*. Esses registradores são necessários para armazenar os resultados das medições dos qubits qr[2] e qr[3] que foram medidos pelos medidores implementados por:
`p += MEASURE(2, ro[0])`
e
`p += MEASURE(2, ro[0]).`
As linhas de comando finais para a compilação e execução do algoritmo são as mesmas adicionadas no algoritmo anterior. O algoritmo completo é mostrado abaixo.

```
from pyquil import get_qc, Program
from pyquil.gates import *
from pyquil.quilbase import Declare
p = Program()
p += X(0)
p += X(2)
p += CCNOT(0,1,3)
p += CNOT(0,1)
p += CCNOT(1,2,3)
p += CNOT(1,2)
p += CNOT(0,1)
ro = p.declare('ro', 'BIT', 2)
p += MEASURE(2, ro[0])
p += MEASURE(3, ro[1])
qc = get_qc("4q-qvm")
executable = qc.compile(p)
result = qc.run(executable)
print(result.readout_data.get('ro'))
```

O resultado da execução desse algoritmo é [[01]] que corresponde ao valor 0 para o qubit qr[2] e 1 para o qubit qr[3].

4.6 Implementação em Python com o SDK Cirq do circuito quântico *Full Adder*

No Cirq, a única biblioteca a ser importada é a biblioteca *cirq*. O circuito quântico e os quatro qubits são instanciados por:
`qc = cirq.Circuit(),`

```
qr0 = cirq.NamedQubit('q[0]'),
qr1 = cirq.NamedQubit('q[1]'),
qr2 = cirq.NamedQubit('q[2]')
```

e

```
qr3 = cirq.NamedQubit('q[3]').
```

A porta TOFFOLI é instanciada pelo método *TOFFOLI()* que recebe as variáveis que identificam os qubits de controle e o qubit alvo. No Cirq, as portas quânticas são implementadas pelo método *append()*. A sequência de linhas de comando que implementam as portas quânticas do circuito do *Full Adder* quântico é:

```
qc.append(cirq.X(qr0)),
qc.append(cirq.X(qr2)),
qc.append(cirq.TOFFOLI(qr0,qr1,qr3)),
qc.append(cirq.CNOT(qr0,qr1)),
qc.append(cirq.TOFFOLI(qr1,qr2,qr3)),
qc.append(cirq.CNOT(qr1,qr2)),
qc.append(cirq.CNOT(qr0,qr1)),
qc.append(cirq.measure(qr2,key = 'q[2]'))
```

e

```
qc.append(cirq.measure(qr3,key = 'q[3]')).
```

Os comandos finais do algoritmo são:

```
simulador = cirq.Simulator(),
```

que instancia o simulador, e

```
resultado = simulador.run(qc,repetitions=10),
```

que executa o método *run()*. O algoritmo completo é mostrado abaixo.

```
import cirq
qc = cirq.Circuit()
qr0 = cirq.NamedQubit('q[0]')
qr1 = cirq.NamedQubit('q[1]')
qr2 = cirq.NamedQubit('q[2]')
qr3 = cirq.NamedQubit('q[3]')
qc.append(cirq.X(qr0))
qc.append(cirq.X(qr2))
qc.append(cirq.TOFFOLI(qr0,qr1,qr3))
qc.append(cirq.CNOT(qr0,qr1))
qc.append(cirq.TOFFOLI(qr1,qr2,qr3))
qc.append(cirq.CNOT(qr1,qr2))
qc.append(cirq.CNOT(qr0,qr1))
qc.append(cirq.measure(qr2,key = 'q[2]'))
qc.append(cirq.measure(qr3,key = 'q[3]'))
```

```
simulador = cirq.Simulator()
resultado = simulador.run(qc,repetitions=10)
print(resultado)
```

O resultado da execução do algoritmo acima é:

```
q[2]=0000000000
q[3]=1111111111
```

Nas dez medições realizadas, o mesmo resultado foi obtido. Esse resultado corresponde ao número decimal 2 como esperado.

4.7 Implementação em Python com o SDK Qiskit do algoritmo de Grover

Nesta seção, o circuito quântico para o algoritmo de Grover mostrado na Figura 17 será implementado em linguagem Python com o uso do SDK Qiskit. As bibliotecas utilizadas na implementação do algoritmo de Grover são as mesmas utilizadas nas implementações dos algoritmos anteriores: *QuantumCircuit*, *QuantumRegister*, *ClassicalRegister*, *Aer* e *execute*. Serão utilizados dois qubits no registrador de n qubits e um qubit no registrador de um qubit. Esses registradores são instanciados por:

```
qr_n = QuantumRegister(2)
```

e

```
qr_1 = QuantumRegister(1).
```

O registrador clássico é utilizado para armazenar os resultados das medidas dos qubits do registrador de n qubits. O registrador clássico é instanciado por:

```
cr = ClassicalRegister(2).
```

Como há dois qubits, é preciso um registrador clássico com dois bits. O circuito quântico é instanciado por:

```
qc = QuantumCircuit(qr_n,qr_1,cr).
```

Como todos os qubits iniciam no estado $|0\rangle$, a porta de Pauli X é aplicada sobre o qubit do registrador de um qubit para mudar o seu estado para $|1\rangle$. Essa porta de Pauli X é implementada no algoritmo por:

```
qc.x(qr_1[0]).
```

Portas de Hadamard devem ser aplicadas sobre todos os qubits do registrador de n qubits de modo a gerar uma superposição dos estados associados a esses qubits. Uma porta de Hadamard também deve ser aplicada sobre o qubit do registrador de um qubit para levar esse qubit para o estado $|-\rangle$. As portas de Hadamard são implementadas no algoritmo por:

```
qc.h(qr_n[0]),
qc.h(qr_n[1])
```

e

```
qc.h(qr_1[0]).
```

O operador U_f é implementado no algoritmo por:

```
qc.x(qr_n[1]),
qc.ccx(qr_n[0],qr_n[1],qr_1[0])
```

e

```
qc.x(qr_n[1]).
```

O operador $2|\psi\rangle\langle\psi| - I$ é implementado por:

```
qc.h(qr_n[0]),
qc.h(qr_n[1]),
qc.x(qr_n[0]),
qc.x(qr_n[1]),
qc.ccx(qr_n[0],qr_n[1],qr_1[0]),
qc.x(qr_n[0]),
qc.x(qr_n[1]),
qc.h(qr_n[0])
```

e

```
qc.h(qr_n[1]).
```

Os medidores sobre os qubits do primeiro registrador são implementado no algoritmo. O *backend* selecionado é o *qasm_simulator*. O algoritmo completo é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr_n = QuantumRegister(2)
qr_1 = QuantumRegister(1)
cr = ClassicalRegister(2)
qc = QuantumCircuit(qr_n,qr_1,cr)
qc.x(qr_1[0])
qc.h(qr_n[0])
qc.h(qr_n[1])
qc.h(qr_1[0])
qc.x(qr_n[1])
qc.ccx(qr_n[0],qr_n[1],qr_1[0])
qc.x(qr_n[1])
qc.h(qr_n[0])
qc.h(qr_n[1])
qc.x(qr_n[0])
```

```
qc.x(qr_n[1])
qc.ccx(qr_n[0],qr_n[1],qr_1[0])
qc.x(qr_n[0])
qc.x(qr_n[1])
qc.h(qr_n[0])
qc.h(qr_n[1])
qc.measure(qr_n[0],cr[0])
qc.measure(qr_n[1],cr[1])
backend = Aer.get_backend('qasm_simulator')
job = execute(qc,backend,shots = 1024)
resultado = job.result()
contagem = resultado.get_counts()
print(contagem)
```

O resultado da execução desse algoritmo é {'01': 1024}. Esse resultado, que foi obtido em todas as 1024 medições, corresponde ao valor $i_0 = 1$.

5 Conclusões

Computação quântica é uma área interdisciplinar que envolve conhecimentos de computação, física e matemática. O grande avanço tecnológico alcançado nessa área al longo dos últimos anos tornaram computadores quânticos uma realidade. Há grande expectativa de que computadores quânticos serão comercializados em breve tornando-se mais uma ferramenta para computação de alto desempenho. Como consequência desse avanço da computação quântica, surgiu uma nova área dedicada ao desenvolvimento de metodologias de ensino de computação quântica tanto para o ensino médio quanto para o ensino superior. Essa é a área em que se encontra esse trabalho de conclusão de curso.

Nesse trabalho é proposto um tutorial para o ensino de programação de computadores quânticos para estudantes que têm conhecimentos em programação, mas não possuem conhecimentos avançados em física e matemática. Esse é o perfil dos estudantes de cursos de ciência da computação e engenharia da computação. O tutorial requer apenas conhecimentos básicos de matemática, que incluem representação vetorial e números complexos, e programação em linguagem Python. Logo, esse tutorial também pode ser utilizado para o ensino de estudantes do nível médio. O tutorial, que foi chamado de *Tutorial de Programação de Computadores Quânticos*, encontra-se no Apêndice A e está disponível em quantumcomputing.orgfree.com. O tutorial é dividido em algoritmos. A cada algoritmo o estudante adquire novos conhecimentos de computação quântica e pode aplicar esses conhecimentos no desenvolvimento do algoritmo. Isso garante um método de ensino bastante prático.

Além do desenvolvimento do tutorial, que é o objetivo desse trabalho, esse trabalho também inclui uma revisão dos conceitos fundamentais de computação quântica. Dessa forma, esse trabalho também é uma referência para estudantes que, após finalizarem todos os algoritmos presentes no tutorial, desejarem rever os conceitos de um modo mais formal. Porém, isso requer conhecimentos mais avançados de física e matemática.

Em uma próxima etapa, pretende-se aplicar esse tutorial para o ensino de computação quântica em sala de aula. Os resultados colhidos com essa experiência de ensino possibilitará aprimorar o tutorial desenvolvido nesse trabalho. Outra sugestão de trabalho futuro é ampliar a quantidade de algoritmos. Cada novo algoritmo deve ser elaborado de forma a respeitar os limites de conhecimentos dos estudantes que são o público alvo desse tutorial. A avaliação também é um ponto importante a ser desenvolvido e aprimorado porque garante um retorno da eficácia desse tutorial em cumprir o seu objetivo que é ensinar conceitos de computação quântica e desenvolver a habilidade de escrever programas para computadores quânticos. Espera-se que esse tutorial seja realmente eficaz e que sua metodologia venha facilitar e acelerar o processo de aprendizagem e aumentar o interesse do estudante pelo assunto.

Referências

ABHIJITH, J. et al. Quantum algorithm implementations for beginners. *arXiv e-prints*, p. arXiv–1804, 2018. Citado na página 16.

ALTENKIRCH, T.; GRATTAGE, J. A functional quantum programming language. In: IEEE. *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. [S.l.], 2005. p. 249–258. Citado na página 32.

AMARIUTEI, A.; CARAIMAN, S. Parallel quantum computer simulation on the gpu. In: IEEE. *15th International Conference on System Theory, Control and Computing*. [S.l.], 2011. p. 1–6. Citado na página 36.

ANGARA, P. P. et al. Teaching quantum computing to high-school-aged youth: A hands-on approach. *IEEE Transactions on Quantum Engineering*, IEEE, v. 3, p. 1–15, 2021. Citado na página 16.

ARUTE, F. et al. Quantum supremacy using a programmable superconducting processor. *Nature*, Nature Publishing Group, v. 574, n. 7779, p. 505–510, 2019. Citado na página 15.

ASFAW, A. et al. Learn quantum computation using qiskit. *URL: https://qiskit. org/textbook/ch-applications/qaoa. html (accessed 06/21/2021)*, 2020. Citado 2 vezes nas páginas 17 e 38.

AVILA, A. B. de et al. State-of-the-art quantum computing simulators: Features, optimizations, and improvements for d-gm. *Neurocomputing*, Elsevier, v. 393, p. 223–233, 2020. Citado na página 36.

BETTELLI, S.; CALARCO, T.; SERAFINI, L. Toward an architecture for quantum programming. *The European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics*, Springer, v. 25, n. 2, p. 181–200, 2003. Citado na página 32.

BOIXO, S. et al. Characterizing quantum supremacy in near-term devices. *Nature Physics*, Nature Publishing Group, v. 14, n. 6, p. 595–600, 2018. Citado na página 36.

CARRASCAL, G.; BARRIO, A. A. D.; BOTELLA, G. First experiences of teaching quantum computing. *The Journal of Supercomputing*, Springer, v. 77, n. 3, p. 2770–2799, 2021. Citado 2 vezes nas páginas 16 e 37.

CHEN, Z.-Y. et al. 64-qubit quantum circuit simulation. *Science Bulletin*, Elsevier, v. 63, n. 15, p. 964–971, 2018. Citado na página 36.

CINCIO, L. et al. Learning the quantum algorithm for state overlap. *New Journal of Physics*, IOP Publishing, v. 20, n. 11, p. 113022, 2018. Citado na página 16.

COHEN-TANNOUDJI, C.; DIU, B.; LALOË, F. *Quantum mechanics; 1st ed.* New York, NY: Wiley, 1977. Citado na página 19.

CROSS, A. W. et al. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017. Citado 2 vezes nas páginas 32 e 45.

DEUTSCH, D. Quantum theory, the church–turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, The Royal Society London, v. 400, n. 1818, p. 97–117, 1985. Citado na página 15.

DUMITRESCU, E. F. et al. Cloud quantum computing of an atomic nucleus. *Physical review letters*, APS, v. 120, n. 21, p. 210501, 2018. Citado na página 16.

EFTHYMIOU, S. et al. Qibo: a framework for quantum simulation with hardware acceleration. *Quantum Science and Technology*, IOP Publishing, v. 7, n. 1, p. 015018, 2021. Citado na página 36.

FEYNMAN, R. Simulating physics with computers. *International Journal of Theoretical Physics*, v. 21, n. 6, 1982. Citado na página 15.

GARHWAL, S.; GHORANI, M.; AHMAD, A. Quantum programming language: A systematic review of research topic and top cited languages. *Archives of Computational Methods in Engineering*, Springer, v. 28, n. 2, p. 289–310, 2021. Citado na página 32.

GATTI, L.; SOTELO, R. Quantum computing for undergraduate engineering students: Report of an experience. In: IEEE. *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. [S.l.], 2021. p. 397–401. Citado na página 16.

GAY, S. J. Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science*, Cambridge University Press, v. 16, n. 4, p. 581–600, 2006. Citado na página 32.

GREEN, A. S. et al. An introduction to quantum programming in quipper. In: SPRINGER. *International Conference on Reversible Computation*. [S.l.], 2013. p. 110–124. Citado na página 32.

GROVER, L. K. Quantum mechanics helps in searching for a needle in a haystack. *Physical review letters*, APS, v. 79, n. 2, p. 325, 1997. Citado na página 15.

HEIM, B. et al. Quantum programming languages. *Nature Reviews Physics*, Nature Publishing Group, p. 1–14, 2020. Citado na página 32.

HENG, S.; KIM, T.; HAN, Y. Exploiting gpu-based parallelism for quantum computer simulation: A survey. *IEIE Transactions on Smart Processing & Computing*, v. 9, n. 6, p. 468–476, 2020. Citado na página 36.

HUGHES, C. et al. Teaching quantum computing to high school students. *arXiv preprint arXiv:2004.07206*, 2020. Citado 2 vezes nas páginas 16 e 37.

IBM. *IBM Quantum*. 2021. https://quantum-computing.ibm.com/. Citado na página 33.

JESUS, G. F. d. et al. Computação quântica: uma abordagem para a graduação usando o qiskit. *Revista Brasileira de Ensino de Física*, SciELO Brasil, v. 43, 2021. Citado 2 vezes nas páginas 16 e 38.

KHAMMASSI, N. et al. cQASM v1. 0: Towards a common quantum assembly language. *arXiv preprint arXiv:1805.09607*, 2018. Citado na página 32.

KNILL, E. *Conventions for quantum pseudocode*. [S.l.], 1996. Citado na página 32.

LAPETS, A. et al. Quafl: A typed dsl for quantum programming. In: *Proceedings of the 1st annual workshop on Functional programming concepts in domain-specific languages*. [S.l.: s.n.], 2013. p. 19–26. Citado na página 32.

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. Citado na página 31.

LIU, S. et al. $Q\,|SI\rangle$: A quantum programming environment. In: SPRINGER. *Symposium on Real-Time and Hybrid Systems*. [S.l.], 2018. p. 133–164. Citado na página 32.

MAYMIN, P. Extending the lambda calculus to express randomized and quantumized algorithms. *arXiv preprint quant-ph/9612052*, 1996. Citado na página 32.

MLNARIK, H. *Quantum programming language LanQ*. Tese (Doutorado) — Masaryk University-Faculty of Informatics, Czech Republic, 2007. Citado na página 32.

MOREAU, P.-A. et al. Imaging bell-type nonlocal behavior. *Science advances*, American Association for the Advancement of Science, v. 5, n. 7, p. eaaw2563, 2019. Citado na página 25.

NIELSEN, M. A.; CHUANG, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. [S.l.]: Cambridge University Press, 2010. Citado na página 20.

ÖMER, B. A procedural formalism for quantum computing. Citeseer, 1998. Citado na página 32.

PEDNAULT, E. et al. Breaking the 49-qubit barrier in the simulation of quantum circuits. *arXiv preprint arXiv:1710.05867*, v. 15, 2017. Citado na página 36.

PEDNAULT, E. et al. Leveraging secondary storage to simulate deep 54-qubit sycamore circuits. *arXiv preprint arXiv:1910.09534*, 2019. Citado na página 15.

PERRY, A. et al. Quantum computing as a high school module. *arXiv preprint arXiv:1905.00282*, 2019. Citado 2 vezes nas páginas 37 e 38.

PORTUGAL, R. et al. *Uma introdução à computação quântica*. [S.l.]: SBMAC, 2004. (Notas em Matemática Aplicada, 8). ISBN 9788586883170. Citado 2 vezes nas páginas 25 e 30.

PORTUGAL, R.; MARQUEZINO, F. Introdução à programação de computadores quânticos. *Sociedade Brasileira de Computação*, 2019. Citado 2 vezes nas páginas 30 e 45.

QIAO, H. et al. Conditional teleportation of quantum-dot spin states. *Nature communications*, Nature Publishing Group, v. 11, n. 1, p. 1–9, 2020. Citado na página 25.

RABELO, W. R.; COSTA, M. L. M. Uma abordagem pedagógica no ensino da computação quântica com um processador quântico de 5-qbits. *Revista Brasileira de Ensino de Física*, SciELO Brasil, v. 40, 2018. Citado 2 vezes nas páginas 16 e 38.

SEEGERER, S.; MICHAELI, T.; ROMEIKE, R. Quantum computing as a topic in computer science education. *The 16th Workshop in Primary and Secondary Computing Education*, p. 1–6, 2021. Citado na página 16.

SELINGER, P. A brief survey of quantum programming languages. In: SPRINGER. *International Symposium on Functional and Logic Programming*. [S.l.], 2004. p. 1–6. Citado na página 32.

SELINGER, P. Towards a quantum programming language. *Mathematical Structures in Computer Science*, Cambridge University Press, v. 14, n. 4, p. 527–586, 2004. Citado na página 32.

SHIBAGAKI, K. *Achieving Scalability: The key to future quantum computers*. 2020. Disponível em: <https://www.mri.co.jp/en/50th/columns/quantum/no02/>. Citado na página 35.

SHOR, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In: IEEE. *Proceedings 35th annual symposium on foundations of computer science*. [S.l.], 1994. p. 124–134. Citado na página 15.

SMITH, R. S.; CURTIS, M. J.; ZENG, W. J. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016. Citado na página 32.

SUN, J. Teaching high school quantum computing scenarios. *Proceedings of Student-Faculty Research Day CSIS*, 2019. Citado na página 16.

SVORE, K. M. et al. A layered software architecture for quantum computing design tools. *Computer*, IEEE, v. 39, n. 1, p. 74–83, 2006. Citado na página 32.

WECKER, D.; SVORE, K. M. LIQ$Ui\,|\rangle$: A software design architecture and domain-specific language for quantum computing. *arXiv preprint arXiv:1402.4467*, 2014. Citado na página 32.

ZULIANI, P. *Quantum programming*. Tese (Doutorado) — University of Oxford, 2001. Citado na página 32.

Apêndices

APÊNDICE A – Tutorial

Tutorial de Programação de Computadores Quânticos

O objetivo desse Tutorial de Computação Quântica é apresentar conteúdos fundamentais de Computação Quântica de forma simples e didática. Todo o rigor do formalismo da mecânica quântica pode ser omitido em uma etapa inicial do ensino de Computação Quântica para estudantes com conhecimentos básicos de linguagem de programação. O processo de aprendizagem ocorre através de uma série de algoritmos. Os conceitos de Computação Quântica são introduzidos de forma gradual atendendo às necessidades de cada algoritmo. Esses algoritmos são escritos em linguagem Python com o uso de um dos kits de desenvolvimento de software (SDK – *Software Development Kit*): Qiskit, Forest ou Cirq. Espera-se que os estudantes sejam capazes de escrever seus primeiros algoritmos quânticos ao final desse tutorial.

A.1 Introdução

Computação Quântica ainda pode parecer algo distante da nossa realidade, mas já é possível a qualquer pessoa executar um programa em um computador quântico. Grande progresso tem sido alcançado nessa área nos último anos e há interesse por parte das empresas que investem nessa área em formar pessoas que serão capazes de programar esses computadores que estarão disponíveis em alguns anos. Por esse motivo, essas empresas também investem na produção de material para o ensino e divulgação de computação quântica e permitem o acesso a simuladores de computadores quânticos. Além disso, tanto a IBM quanto a Rigetti disponibilizam o acesso a alguns de seus computadores quânticos permitindo que qualquer usuário possa executar um programa em um computador quântico.

Um grande marco na história da Computação Quântica ocorreu em 2019, quando a Google anunciou que o processador quântico Sycamore de 53 qubits foi utilizado com sucesso para realizar uma determinada tarefa computacional em um tempo de aproximadamente 200 segundos que levaria cerca de 10 mil anos para ser realizada pelo Summit [1]. O Summit é um computador de 200 petaflops produzido pela IBM para o governo dos EUA que está localizado

no Laboratório Nacional de Oak Ridge. Ao tomar conhecimento, a IBM corrigiu a declaração dos pesquisadores da Google e afirmou que o Summit realizaria tal operação em dois dias e meio [2]. Ainda assim, o feito realizado pela Google foi impressionante o que garantiu à Google e aos EUA o domínio da Supremacia Quântica. Computação Quântica foi uma das áreas prioritárias do recurso destinado à pesquisa e desenvolvimento pelo governo dos EUA em 2021. A disputa pelo domínio da tecnologia quântica tem levado muitos países a destinar quantias cada vez maiores para pesquisas nessa área de conhecimento.

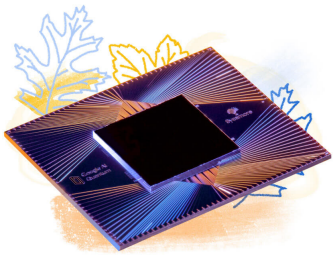

Figura 21 – Processador quântico Sycamore [3].

São destaque entre as empresas que investem no desenvolvimento de processadores quânticos, a Google, a IBM, a Intel e a Rigetti. Até fevereiro de 2022, os processadores de cada uma dessas empresas que possuem a maior quantidade de qubits são: o Aspen 9 com 31 qubits desenvolvido pela Rigetti, o Tangle Lake com 49 qubits desenvolvido pela Intel, o Bristlecone com 72 qubits desenvolvido pela Google e o Eagle com 127 qubits desenvolvido pela IBM. A IBM pretende lançar, em 2022, o processador Osprey com 433 qubits e, em 2023, o processador Condor com 1.123 qubits.

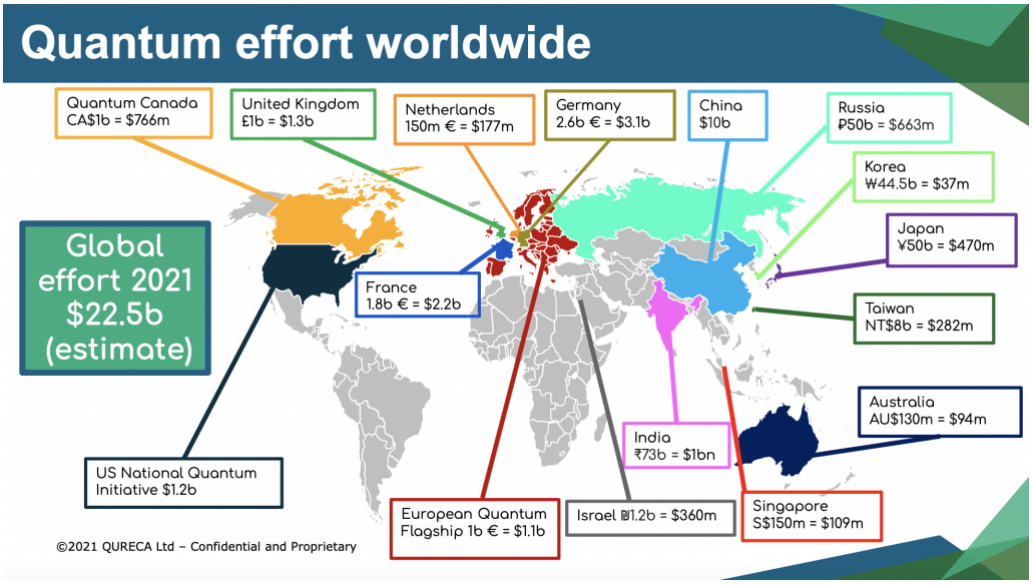

Figura 22 – Recursos destinados à pesquisa e desenvolvimento de tecnologias quânticas [4].

Processadores quânticos requerem temperaturas extremamente baixas para o seu fun-

cionamento. Esses processadores são acoplados na extremidade inferior de um refrigerador de diluição que permite que o processador quântico opere em temperaturas próximas ao zero absoluto. Processadores quânticos também precisam ser blindados de toda interferência externa. Para isso, são inseridos em cilindros onde é feito vácuo e há sistemas de proteção contra campos eletromagnéticos externos.

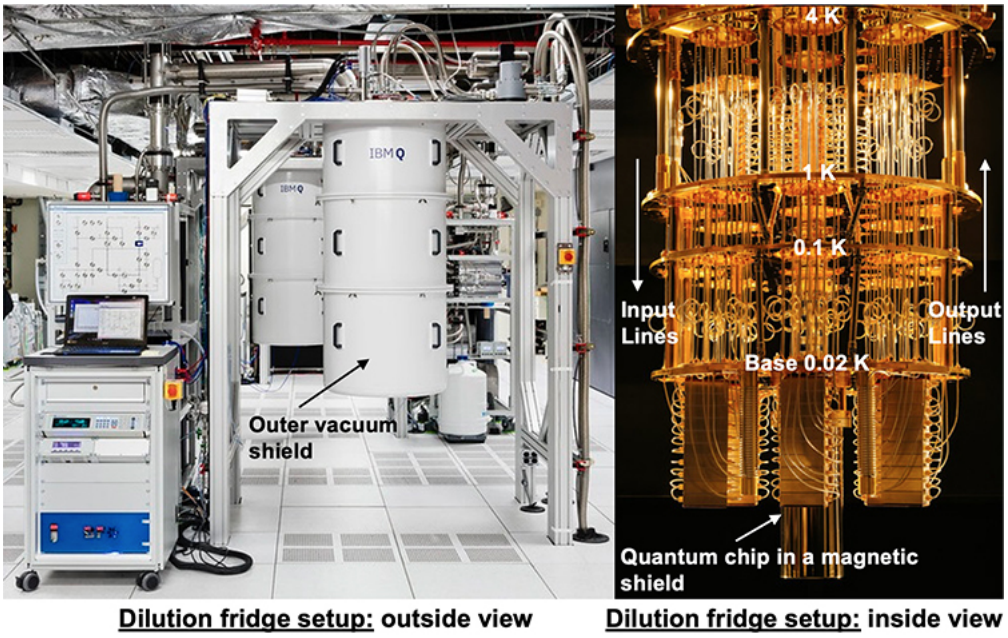

Figura 23 – Refrigerador de diluição [5].

A.2 Kits de Desenvolvimento de Software

A.2.1 Qiskit

Os computadores quânticos da IBM executam conjuntos de instruções escritos na linguagem OpenQASM [6] que é uma das variantes da linguagem QASM [7]. A IBM oferece duas opções para programar em OpenQASM e executar programas escritos nessa linguagem em seus simuladores e computadores quânticos: a plataforma IBM Quantum e o SDK Qiskit.

Na plataforma IBM Quantum, encontram-se duas opções: o IBM Quantum Composer e o IBM Quantum Lab. O IBM Quantum Composer oferece uma facilidade muito grande para o desenvolvimento de algoritmos quânticos, porém permite trabalhar com um número limitado de qubits. O IBM Quantum Lab oferece um ambiente de programação em Python através do Jupyter Notebook sem a necessidade de instalação de programas no computador do usuário.

O SDK Qiskit deve ser instalado no computador do usuário para que este possa desenvolver algoritmos quânticos e submetê-los para execução em um dos simuladores ou computadores quânticos da IBM ou utilizar a sua própria máquina como um simulador quântico. A

programação é realizada em linguagem Python com a opção de conversão para a linguagem OpenQASM.

O ecossistema do Qiskit inclui quatro frameworks: Terra, Aer, Ignis e Aqua. Ecossistemas de software são grandes coleções de pacotes de software que possuem uma relação de dependência e evoluem de forma conjunta. O Qiskit Terra é o framework base que fornece as ferramentas para o desenvolvimento e a otimização de programas quânticos. O Qiskit Terra também é responsável pelo gerenciamento da execução de programas em batch através de acesso remoto. O Qiskit Aer inclui três simuladores de alta performance: QasmSimulator, StatevectorSimulator e UnitarySimulator. Esses simuladores são utilizados no estudo dos erros que ocorrem durante a execução em computadores quânticos reais. O Qiskit Ignis também é utilizado no estudo dos erros durante a execução de programas quânticos e possibilita projetar códigos de correção de erros e caracterizar os erros de execução. O Qiskit Aqua é o framework utilizado no desenvolvimento de programas quânticos para aplicações que requerem o uso de aceleradores quânticos para a realização de tarefas específicas.

A instalação do Qiskit requer a instalação do software Anaconda Python. Para instalar o SDK Qiskit nos sistemas operacionais Linux ou MacOS, basta executar, no terminal, a linha de comando

```
pip install qiskit.
```

No sistema operacional Windows, essa mesma linha de comando deve ser executada no Anaconda Prompt.

A.2.2 Forest

A Rigetti Computing permite o acesso aos seus processadores quânticos através de sua própria plataforma de computação quântica em nuvem chamada de *Quantum Cloud Services* (QCS). A linguagem utilizada para programar os computadores quânticos da Rigetti é o Quil (*Quantum Instruction Language*) [8]. Através da biblioteca pyQuil, programas escritos em linguagem Python podem ser traduzidos para Quil e executados nos simuladores ou computadores quânticos da Rigetti. O desenvolvimento de algoritmos quânticos para os computadores e simuladores quânticos da Rigetti é facilitado com o uso do SDK Forest. O Forest inclui o compilador quilc e o simulador QVM (*Quantum Virtual Machine*).

O uso da biblioteca pyQuil e do SDK Forest requer a versão 3.7 ou superior do Python. Portanto, deve-se verificar a versão instalada no computador antes de iniciar os procedimentos de instalação que podem ser encontrados na documentação da biblioteca pyQuil. A instalação da biblioteca pyQuil é feita com a execução da linha de comando

```
pip install pyquil
```

no terminal dos sistemas operacionais Linux ou MacOS ou no prompt de comando do sistema operacional Windows. Após a instalação do pyQuil, deve-se fazer o download do SDK Forest. No Windows, a instalação do SDK Forest resume-se em executar o arquivo forest-sdk.msi e

seguir as instruções. No macOS, a instalação requer a execução do arquivo forest-sdk.dmg seguido da execução do arquivo forest-sdk.pkg. No Linux, para a distribuição .deb executa-se:

```
tar -xf forest-sdk-linux-deb.tar.bz2
cd forest-sdk-linux-deb
sudo ./forest-sdk-linux-deb.run
```

Para a distribuição .rmp executa-se:

```
tar -xf forest-sdk-linux-rpm.tar.bz2
cd forest-sdk-linux-rpm
sudo ./forest-sdk-linux-rpm.run
```

Para maiores detalhes sobre os procedimentos de instalação, pode-se consultar o Guia de Instalação da biblioteca pyQuil e do SDK Forest.

A.2.3 Cirq

O Cirq é uma biblioteca para a linguagem Python que permite escrever algoritmos para programar os computadores quânticos da Google. Através do Cirq, pode-se simular um processador quântico no computador do usuário. O acesso aos processadores quânticos da Google através do Cirq é possível, porém o acesso é restrito. A Google disponibiliza um amplo material dedicado a ensino de computação quântica em quantumai.google.

A instalação do Cirq não apresenta dificuldades. Antes de instalar o Cirq, certifique-se de que a versão 3.7 ou superior do Python está instalada em seu computador. Para instalar o Cirq, basta executar os comandos:

```
python -m pip install -upgrade pip
python -m pip install cirq
```

Maiores detalhes sobre o procedimento de instalação pode ser encontrados no Guia de Instalação do SDK Cirq.

A.3 Algoritmo 1 – O que é qubit?

Na página da Intel, os processadores i7 são especificados pela sua memória cache de 25 MB e pela sua frequência de 5.00 GHz. Essas são duas propriedades importantes para indicar o desempenho de um processador. Diferente de um processador clássico, um processador quântico tem o seu desempenho determinado pela quantidade de qubits que podem ser manipulados em uma operação. Então, fica a pergunta: o que seria um qubit?

Em um curso de programação de alguma linguagem de alto nível, é suficiente ao estudante saber que um bit é uma unidade que armazena informação, sendo que essa informação pode ser o valor 0 ou o valor 1. Pode-se também definir qubit como uma unidade que armazena informação, porém essa informação é um par de números complexos α e β que satisfazem a condição $|\alpha|^2 + |\beta|^2 = 1$. Para obter o valor armazenado em um qubit realiza-se um processo

chamado de medição. O resultado da medição de um qubit tem probabilidade $|\alpha|^2$ de ser 0 e probabilidade $|\beta|^2$ de ser 1. O qubit pode estar em dois estados diferentes ao mesmo tempo sendo que esses estados são identificados por $|0\rangle$ e $|1\rangle$. Essa notação pode parecer um pouco estranha ao estudante, mas preferiu-se manter essa notação por se tratar da notação utilizada para representar estados quânticos na física. O estudante está familiarizado com a notação $\{\hat{x},\hat{y}\}$ para representar a base de um espaço bidimensional. A base para representar o estado de um qubit é $\{|0\rangle,|1\rangle\}$. Essa base é conhecida como base computacional da computação quântica. O estado de um qubit é matematicamente descrito por [9]:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,. \tag{A.1}$$

O processo de medição interfere no estado do qubit de tal forma que, quando o resultado da medição é 0, o novo estado do qubit é $|0\rangle$ e, quando o resultado é 1, o novo estado do qubit é $|1\rangle$. Essa alteração no estado do qubit provocada pelo processo de medição é chamada de colapso de função de onda.

O primeiro algoritmo desse tutorial é dedicado a verificar essa importante propriedade de um qubit. Antes de iniciar nosso primeiro algoritmo, dois novos conceitos precisam ser introduzidos: circuito quântico e portas quânticas. Escrever um algoritmo quântico equivale a criar um circuito quântico. Portanto, um circuito quântico é uma forma de representar um algoritmo quântico. Um circuito quântico é composto por um conjunto de linhas horizontais sendo que cada linha está associada a um qubit que foi declarado no algoritmo. Algumas formas de representar circuitos quânticos pode incluir uma última linha horizontal dupla que corresponde ao conjunto de registradores clássicos onde os resultados das medições realizadas são armazenados.

As informações armazenadas em bits são manipuladas através de portas lógicas em um circuito digital. De modo análogo, as informações armazenadas em qubits são manipuladas através de portas quânticas em um circuito quântico. Cada porta quântica utilizada no circuito quântico é adicionada sobre a linha associada ao qubit em que a porta foi aplicada. A sequência das portas ao longo de cada linha obedece à ordem em que as portas são aplicadas. A Figura 24 mostra o circuito quântico associado ao primeiro algoritmo. Nesse circuito, utilizamos a porta de Hadamard que é representada por uma caixa com a letra H em seu interior. Essa porta quântica altera o estado do qubit sobre o qual ela é aplicada de modo que os valores $(\alpha+\beta)+\sqrt{2}$ e $(\alpha-\beta)+\sqrt{2}$ são atribuídos às variáveis α e β do novo estado. Para realizar a medição do qubit, foi adicionado ao circuito um medidor quântico. Esse medidor é representado por uma caixa contendo o desenho de um indicador analógico.

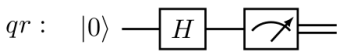

Figura 24 – Circuito quântico associado ao primeiro algoritmo desse tutorial.

Qiskit

As bibliotecas necessárias para implementar o primeiro algoritmo desse tutorial são: *QuantumCircuit*, *QuantumRegister* e *ClassicalRegister*. Essas bibliotecas devem ser importadas através das linhas de comando:

```
from qiskit import QuantumCircuit,
from qiskit import QuantumRegister
```

e

```
from qiskit import ClassicalRegister.
```

A biblioteca *QuantumCircuit* é necessária para criar o circuito quântico através do construtor *QuantumCircuit()*. A biblioteca *QuantumRegister* é necessária para instanciar o qubit utilizado no circuito quântico. Para isso, utiliza-se o construtor *QuantumRegister()*. Ao argumento, atribui-se o valor 1 porque será declarado apenas um qubit no circuito quântico. O valor obtido com a medição desse qubit deve ser armazenado em um registrado clássico que é instanciado pelo construtor *ClassicalRegister()* que também tem o valor 1 como argumento porque é necessário apenas 1 registrador para armazenar o resultado. Os argumentos do construtor *QuantumCircuit()* são os registradores quântico e clássico. Portanto, as linhas de comando que devem ser adicionadas ao programa são:

```
qr = QuantumRegister(1),
cr = ClassicalRegister(1)
```

e

```
qc = QuantumCircuit(qr,cr).
```

Os valores de α e β para o qubit em seu estado inicial são 1 e 0, respectivamente. Portanto, a medição de um qubit em seu estado inicial sempre resultaria no valor 0. A ação da porta de Hadamard leva o qubit para um novo estado onde $\alpha = (1+0)/\sqrt{2} = 0,707$ e $\beta = (1-0)/\sqrt{2} = 0,707$. A medição desse qubit terá probabilidade 0,5 de resultar em 0 e, também, probabilidade 0,5 de resultar em 1. A porta de Hadamard é implementada por:

```
qc.h(qr).
```

A medição do qubit é feita com o método *measure()* para o qual passamos como argumento os registradores quântico e clássico. Logo, deve ser incluída no programa a linha de comando:

```
qc.measure(qr,cr).
```

A expressão *backend* refere-se ao sistema que executará o algoritmo. Esse sistema que pode ser um simulador ou um computador quântico. O método *get_backend()* é utilizado para selecionar esse sistema. Esse método requer o *framework* Aer, portanto deve-se incluir entre as bibliotecas a serem importadas a linha de comando:

```
from qiskit import Aer.
```

O simulador é selecionado por:

```
backend = Aer.get_backend('qasm_simulator').
```

O algoritmo é executado pelo método *execute()* que requer, como argumentos, o circuito quân-

tico que deve ser simulado e o sistema escolhido para executar o algoritmo. Pode-se também incluir, entre os argumentos, o número de execuções através do valor atribuído à variável *shots*. Se essa variável é omitida, então são realizadas 1000 execuções. Para realizar 100 execuções, a linha de comando que deve ser adicionada ao programa é:

```
job = execute(qc,backend,shots=100).
```

O método execute() requer a biblioteca execute. Logo, deve-se incluir entre as bibliotecas:

```
from qiskit import execute.
```

O resultado da execução é obtido com o uso do método *result()*. O método *get_counts()* é necessário para extrair do resultado a quantidade de medições que resultaram em 0 e a quantidade de medições que resultaram em 1. As linhas de comando que implementam esses métodos são:

```
resultado = job.result()
```

e

```
contagem = resultado.get_counts().
```

O resultado é impresso na tela pelo comando:

```
print(contagem).
```

O algoritmo completo é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(1)
cr = ClassicalRegister(1)
qc = QuantumCircuit(qr,cr)
qc.h(qr)
qc.measure(qr,cr)
backend = Aer.get_backend('qasm_simulator')
job = execute(qc,backend,shots=100)
resultado = job.result()
contagem = resultado.get_counts()
print(contagem)
```

Ao executar o algoritmo acima, pode-se obter, por exemplo, {'0': 60, '1': 40} que indica que o valor 0 foi obtido em 60 medições e o valor 1 foi obtido em 40 medições. Outro resultado possível seria {'0': 45, '1': 55}. Em cada medição há a probabilidade $\left|1/\sqrt{2}\right|^2 = 0,5$ de que o resultado seja 0 e, também, a probabilidade $\left|1/\sqrt{2}\right|^2 = 0,5$ de que o resultado seja 1. Se o valor 1 é atribuído à variável *shots*, os resultados possíveis seriam {'0': 1} ou {'1': 1} porque uma única medição é realizada.

Forest

Para escrever algoritmos quânticos com o SDK Forest, é preciso instanciar o objeto *Program* que requer a biblioteca *Program*. Portanto, pode-se começar o algoritmo com os comandos:

```
from pyquil import Program
```

e

```
p = Program()
```

. Para implementar a porta de Hadamard, adiciona-se H(0) à variável p através de:

```
p += H(0).
```

O uso da porta de Hadamard requer que a biblioteca referente às portas quânticas seja adicionada ao programa por:

```
from pyquil.gates import *.
```

Para o processo de medição, é preciso declarar um bit com o método *declare()* por meio de:

```
ro = p.declare('ro', 'BIT', 1).
```

Esse método requer a biblioteca *Declare* que é acrescentada ao programa por:

```
from pyquil.quilbase import Declare.
```

Ao bit declarado, atribui-se um nome que, em nosso exemplo, foi *ro*. O medidor quântico pode ser adicionado ao circuito por:

```
p += MEASURE(0, ro[0]).
```

A quantidade de qubits que será manipulada pela máquina virtual quântica é definida pela função *get_qc()*. No algoritmo foi incluído o comando:

```
qc = get_qc("1q-qvm").
```

Esse comando define uma máquina virtual quântica de 1 qubit. Para uma máquina virtual quântica de 2 qubits, o argumento da função *get_qc()* seria "2q-qvm". O programa é compilado pelo método *compile()* e executado pelo método *run()*. Logo, para executar o algoritmo é preciso incluir as linhas de comando:

```
executable = qc.compile(p)
```

e

```
result = qc.run(executable).
```

Para imprimir no terminal somente os valores do resultado, pode-se adicionar o método *readout_data.get()*. Assim, acrescenta-se ao algoritmo:

```
print(result.readout_data.get( 'ro')).
```

O algoritmo completo é:

```
from pyquil import get_qc, Program
from pyquil.gates import *
from pyquil.quilbase import Declare
p = Program()
p += H(0)
```

```
ro = p.declare('ro', 'BIT', 1)
p += MEASURE(0, ro[0])
qc = get_qc("1q-qvm")
executable = qc.compile(p)
result = qc.run(executable)
print(result.readout_data.get( 'ro'))
```

Os resultados possíveis para esse algoritmo são: [[0]] e [[1]]. Para aumentar o número de vezes que o algoritmo é executado, pode-se adicionar a linha de comando
`p.wrap_in_numshots_loop(shots=10)`
antes do comando para compilar o algoritmo. Nesse caso, um dos possíveis resultados seria:
[[1]
[1]
[1]
[0]
[1]
[0]
[1]
[0]
[1]
[1]]
O resultado 0 foi obtido em três medições e o resultado 1 foi obtido em sete medições. O método *wrap_in_numshots_loop()* retorna uma matriz. Cada elemento dessa matriz é o resultado de uma medição. Funções da linguagem Python podem ser utilizadas para modificar a forma como o resultado é impresso.

Cirq

Com o SDK Cirq, uma única biblioteca é necessária para o primeiro algoritmo que é a biblioteca *cirq*. Assim, inicia-se o programa com:
`import cirq`.
No Cirq, qubits e outros objetos quânticos como circuitos quânticos são identificados por instâncias de subclasses. Logo, um circuito quântico é um objeto criado a partir de:
`qc = cirq.Circuit()`.

Há três subclasses para instanciar qubits no SDK Cirq. Para identificar um qubit por uma *string*, usa-se *cirq.NamedQubit()*. A subclasse *cirq.LineQubit()* é utilizada para instanciar qubits em um vetor sendo que cada qubit é identificado por um índice. Para uma matriz de qubits, onde cada qubit é identificado por um par de índices, usa-se a subclasse *cirq.GridQubit()*. Assim, para instanciar o qubit a ser utilizado no algoritmo, pode-se adicionar a linha de comando:
`qr = cirq.NamedQubit('q[0]')`.

A porta de Hadamard e o medidor quântico podem ser implementados no circuito quântico pelo método *append()*. As linhas de comando a serem adicionadas ao algoritmo são:

```
qc.append(cirq.H(qr))
```

e

```
qc.append(cirq.measure(qr)).
```

No Cirq, é suficiente instanciar um simulador com a subclasse *cirq.Simulator()* e executar o programa com o método *run()* que recebe a variável que identifica o circuito como argumento. Pode-se também incluir entre os argumentos do método *run()* a variável *repetitions*, que especifica o número de execuções do algoritmo, e atribuir um valor a essa variável. Assim, adiciona-se ao algoritmo:

```
simulador = cirq.Simulator(),
resultado = simulador.run(qc,repetitions=10)
```

e

```
print(resultado).
```

O algoritmo completo fica:

```
import cirq
qc = cirq.Circuit()
qr = cirq.NamedQubit('q[0]')
qc.append(cirq.H(qr))
qc.append(cirq.measure(qr))
simulador = cirq.Simulator()
resultado = simulador.run(qc,repetitions=10)
print(resultado)
```

Em cada medição, é possível obter o resultado 0 ou o resultado 1. A probabilidade de obter o resultado 0 é 0,5 e a probabilidade de obter 1 é 0,5 conforme discutido anteriormente. Assim, um possível resultado para esse algoritmo é a sequência q[0]=1000100011 que indica que obteve-se seis vezes o valor 0 e quatro vezes o valor 1 nas dez medições realizadas. Outro possível resultado é q[0]=1101100001.

A.4 Algoritmo 2 – Porta de Hadamard

No primeiro algoritmo quântico, observou-se que a medição de um qubit pode retornar 0 ou 1 demonstrando o caráter probabilístico da leitura de um qubit. A porta de Hadamard foi utilizada para alterar o estado do qubit de $|0\rangle$ para $1/\sqrt{2}\,|0\rangle + 1/\sqrt{2}\,|1\rangle$. Nesse algoritmo serão feitas algumas alterações no algoritmo anterior para visualizar o resultado da ação da porta de Hadamard sobre um qubit.

Qiskit

O simulador utilizado na execução do algoritmo 1 foi o *QASM Simulator*. Esse simulador permite múltiplas execuções, mas para visualizar o vetor de estado associado ao qubit deve-se utilizar o método *get_statevector()* que pertence ao simulador *Statevector Simulator*. Assim, para visualizar a ação da porta de Hadamard sobre o estado do qubit, o simulador do algoritmo 1, que foi selecionado pelo método *get_backend()*, foi modificado para:

```
backend = Aer.get_backend ('statevector_simulator').
```

O medidor quântico também deve ser retirado do circuito quântico porque o processo de medição interfere no estado do qubit. Logo, não será preciso um registrador clássico para armazenar o resultado. O algoritmo 2 completo é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(1)
qc = QuantumCircuit(qr)
qc.h(qr)
backend = Aer.get_backend ('statevector_simulator')
job = execute(qc,backend)
resultado = job.result()
estado = resultado.get_statevector()
print(estado)
```

O resultado da execução desse algoritmo é [0.70710678 + 0.j 0.70710678 + 0.j], onde j é a unidade imaginária. Os valores de α e β encontram-se separados por um espaço dentro de colchetes [α β] de modo que α = 0.70710678 + 0.j e β = 0.70710678 + 0.j. Embora a unidade imaginária seja usualmente representada pela letra i, no SDK Qiskit, a letra j é usada para identificar a unidade imaginária. O valor 0,70710678 é o valor $1/\sqrt{2}$ que as variáveis α e β assumem após a ação da porta de Hadamard.

Forest

No SDK Forest, pode-se utilizar o simulador *Wavefunction* para visualizar o efeito da porta de Hadamard. Para isso, inclui-se entre as bibliotecas importadas no início do programa:

```
from pyquil.api import WavefunctionSimulator.
```

É preciso instanciar um objeto do tipo *WavefunctionSimulator* através de:

```
wf_sim = WavefunctionSimulator().
```

O método *wavefunction()* é utilizado para obter o estado do qubit. Logo, adicona-se ao programa:

```
wavefunction = wf_sim.wavefunction(p).
```

O estado do qubit pode ser impresso na tela do terminal pelo comando:
`print(wavefunction).`
O método *get_outcome_probs()* possibilita obter as probabilidades associadas a cada resultado. Esse método pode ser implementado por:
`print(wavefunction.get_outcome_probs( )).`

O algoritmo 2 completo, escrito para o SDK Forest, é:

```
from pyquil import Program
from pyquil.gates import *
from pyquil.api import WavefunctionSimulator
p = Program()
p += H(0)
wf_sim = WavefunctionSimulator()
wavefunction = wf_sim.wavefunction(p)
print(wavefunction)
print(wavefunction.get_outcome_probs ())
```

O resultado da execução do algoritmo acima é:
(0.7071067812 + 0j)$|0\rangle$ + (0.7071067812 + 0j)$|1\rangle$
{'0': 0.4999999999999999, '1': 0.4999999999999999}

O valor 0,7071067812 equivale a $1/\sqrt{2}$. A ação da porta de Hadamard sobre o estado do qubit resulta em α = 0.7071067812 + 0j e β = 0.7071067812 + 0j, onde j é a unidade imaginária. O comando print(wavefunction.get_outcome_probs()) resulta em {'0': 0.4999999999999999, '1': 0.4999999999999999} onde o valor entre aspas simples é o resultado possível e o valor após os dois pontos é a probabilidade desse resultado. Essa probabilidade é $\left|1/\sqrt{2}\right|^2$ = 0,5.

Cirq

No SDK Cirq, o resultado da ação da porta de Hadamard pode ser visualizado através do algoritmo:

```
import cirq
qc = cirq.Circuit()
qr = cirq.NamedQubit('q[0]')
qc.append(cirq.H(qr))
simulador = cirq.Simulator()
resultado = simulador.simulate(qc)
print(resultado)
```

As alterações em relação ao algoritmo 1 são: a exclusão do medidor quântico e o uso do método *simulate()* ao invés do método *run()*. O resultado da execução desse algoritmo é:
measurements: (no measurements)

output vector: 0.707$|0\rangle$ + 0.707$|1\rangle$

Não foi realizada a medição do estado do qubit nesse algoritmo assim como nos algoritmos para os SDKs Qiskit e Forest. Não há medidores quânticos implementados. A simulação realizada fornece o estado do qubit instanciado no algoritmo. O valor 0,707 é o valor de α e β que equivale a $1/\sqrt{2}$.

A.5 Algoritmo 3 – Porta de Pauli X

Algumas portas quânticas podem ser comparadas com portas lógicas como é o caso da porta de Pauli X que é comparada com a porta lógica NOT porque, quando a porta de Pauli X é aplicada sobre um qubit que se encontra no estado $|0\rangle$, esse qubit passará para o estado $|1\rangle$ e, quando a porta de Pauli X é aplicada sobre um qubit que se encontra no estado $|1\rangle$, esse qubit passará para o estado $|0\rangle$. A porta de Pauli X modifica o estado do qubit de modo que o novo estado terá $\alpha = \beta_0$ e $\beta = \alpha_0$, onde α_0 e β_0 são as constantes complexas do estado anterior. A Figura 25 mostra as duas formas de representar a porta de Pauli X em circuitos quânticos.

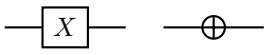

Figura 25 – Representações em circuitos quânticos da porta de Pauli X.

O objetivo desse terceiro algoritmo é demonstrar o resultado da ação da porta de Pauli X. O algoritmo anterior foi desenvolvido para visualizar a ação da porta de Hadamard. Portanto, a única diferença do terceiro para o segundo algoritmo é a troca da porta de Hadamard pela porta de Pauli X. As verões do terceiro algoritmo para os SDKs Qiskit, Forest e Cirq são mostradas abaixo.

Qiskit

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(1)
qc = QuantumCircuit(qr)
qc.x(qr)
backend = Aer.get_backend ('statevector_simulator')
job = execute(qc,backend)
resultado = job.result()
estado = resultado.get_statevector()
print(estado)
```

O estado inicial do qubit é $|0\rangle$ que corresponde a α = 1 e β = 0. A ação da porta de Pauli X altera o estado do qubit para $|1\rangle$ que corresponde a α = 0 e β = 1. O resultado da execução do algoritmo acima é [0. + 0.j 1. + 0.j], onde j é a unidade imaginária. Esse resultado indica que α = 0 e β = 1, portanto o estado do qubit é $|1\rangle$ como esperado.

Forest

```
from pyquil import Program
from pyquil.gates import *
from pyquil.api import WavefunctionSimulator
p = Program()
p += X(0)
wf_sim = WavefunctionSimulator()
wavefunction = wf_sim.wavefunction(p)
print(wavefunction)
print(wavefunction.get_outcome_probs ())
```

O resultado desse algoritmo é:
(1 + 0j)$|1\rangle$
{'0': 0.0, '1': 1.0}

O coeficiente do estado $|1\rangle$ é β, portanto β = 1 + 0j. Como o estado $|0\rangle$ foi omitido, o valor de α é 0. O método *get_outcome_probs()* retorna {'0': 0.0, '1': 1.0} que indica que a probabilidade de obter o resultado 0 é 0 e a probabilidade de obter o resultado 1 é 1.

Cirq

```
import cirq
qc = cirq.Circuit()
qr = cirq.NamedQubit('q[0]')
qc.append(cirq.X(qr))
simulador = cirq.Simulator()
resultado = simulador.simulate(qc)
print(resultado)
```

O resultado desse algoritmo é:
measurements: (no measurements)
output vector: $|1\rangle$

Não foi implementado medidor quântico no circuito para que uma medição fosse realizada. O circuito quântico foi simulado e, como resultado, o estado do qubit é fornecido. Esse estado é $|1\rangle$ devido à porta de Pauli X que alterou o estado do qubit de $|0\rangle$ para $|1\rangle$.

A.6 Algoritmo 4 – Circuitos com dois ou mais qubits

Os algoritmos anteriores continham apenas um qubit e o estado desse qubit era descrito por:

$$|\psi_0\rangle = \alpha_0 |0\rangle + \beta_0 |1\rangle . \tag{A.2}$$

Ao método utilizado para implementar o medidor quântico foi dado como argumento o objeto que representava o circuito quântico. Logo, a medição do estado do qubit é, na verdade, a medição do estado do sistema que era formado por apenas um qubit. Quando o circuito quântico tem dois qubits, os resultados possíveis do processo de medição devem ser 00, 01, 10 e 11. Os estados associados a esses resultados são $|00\rangle$, $|01\rangle$, $|10\rangle$ e $|11\rangle$. Portanto, o estado do sistema é descrito por:

$$|\psi_1\rangle = \alpha_0\alpha_1 |00\rangle + \alpha_0\beta_1 |01\rangle + \beta_0\alpha_1 |10\rangle + \beta_0\beta_1 |11\rangle , \tag{A.3}$$

onde α_0 e β_0 são as constantes complexas associadas ao primeiro qubit e α_1 e β_1 são as constantes complexas associadas ao segundo qubit. Para um sistema de 3 qubits, o estado do sistema é:

$$\begin{aligned} |\psi_2\rangle = \alpha_0\alpha_1\alpha_2 |000\rangle + \alpha_0\alpha_1\beta_2 |001\rangle + \alpha_0\beta_1\alpha_2 |010\rangle + \alpha_0\beta_1\beta_2 |011\rangle + \\ \beta_0\alpha_1\alpha_2 |100\rangle + \beta_0\alpha_1\beta_2 |101\rangle + \beta_0\beta_1\alpha_2 |110\rangle + \beta_0\beta_1\beta_2 |111\rangle , \end{aligned} \tag{A.4}$$

onde α_0 e β_0 são as constantes complexas associadas ao primeiro qubit, α_1 e $\beta 1$ são as constantes complexas associadas ao segundo qubit e $\alpha 2$ e $\beta 2$ são as constantes complexas associadas ao terceiro qubit.

O quarto algoritmo, que é apresentado abaixo em suas versões para Qiskit, Forest e Cirq, tem como objetivo mostrar o estado de um sistema de dois qubits. Para isso são instanciados dois qubits no circuito quântico e uma porta de Hadamard é aplicada em cada qubit do circuito.

Qiskit

No Qiskit, a quantidade de qubits que precisa ser instanciada no circuito deve ser fornecida como argumento para o construtor *QuantumRegister()*. Nesse algoritmo, dois qubits são instanciados por:

```
qr = QuantumRegister(2).
```

Agora, a variável *qr* identifica um vetor de qubits e cada qubit é acessado através do índice que o identifica nesse vetor. Portas de Hadarmad devem ser aplicadas aos qubits qr[0] e qr[1] por:

```
qc.h(qr[0])
```

e

```
qc.h(qr[1]).
```

Essas foram as alterações em relação ao algoritmo anterior. O terceiro algoritmo para o SDK

Qiskit é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(2)
qc = QuantumCircuit(qr)
qc.h(qr[0])
qc.h(qr[1])
backend = Aer.get_backend('statevector_simulator')
job = execute(qc,backend)
resultado = job.result()
estado = resultado.get_statevector()
print(estado)
```

O resultado do algoritmo acima é [0.5 + 0.j 0.5 + 0.j 0.5 + 0.j 0.5 + 0.j], onde j é a unidade imaginária. Portanto, temos quatro constantes complexas que são iguais a 0,5. Essas quatro constantes correspondem aos produtos $\alpha_0\alpha_1$, $\alpha_0\beta_1$, $\beta_0\alpha_1$ e $\beta_0\beta_1$ que aparecem na Equação A.3, onde α_0 e β_0 são as constantes complexas associadas ao primeiro qubit e α_1 e β_1 são as constantes complexas associadas ao segundo qubit.

Forest

Para o SDK Forest, a única mudança nesse quarto algoritmo em relação ao segundo, é o acréscimo da linha de comando:

```
p += H(1).
```

Essa linha de comando implementa a porta de Hadamard sobre o segundo qubit. O quarto algoritmo é mostrado a seguir:

```
from pyquil import Program
from pyquil.gates import *
from pyquil.api import WavefunctionSimulator
p = Program()
p += H(0)
p += H(1)
wf_sim = WavefunctionSimulator()
wavefunction = wf_sim.wavefunction(p)
print(wavefunction)
```

O resultado é (0.5 + 0j) $|00\rangle$ + (0.5 + 0j) $|01\rangle$ + (0.5 + 0j) $|10\rangle$ + (0.5 + 0j) $|11\rangle$, onde j é a unidade imaginária. Como no resultado obtido com o uso do SDK Qiskit, temos: $\alpha_0\alpha_1 = 0.5 + 0j$, $\alpha_0\beta_1 = 0.5 + 0j$, $\beta_0\alpha_1 = 0.5 + 0j$ e $\beta_0\beta_1 = 0.5 + 0j$. A probabilidade de que

o resultado de uma medição, caso fosse realizada, seja 00 é $|0,5|^2 = 0,25$. Essa é a mesma probabilidade para os resultados 01, 10 e 11.

Cirq

No Cirq, é preciso instanciar dois qubits o que é feito por meio de:

```
qr0 = cirq.NamedQubit('q[0]')
```

e

```
qr1 = cirq.NamedQubit('q[1]').
```

As duas portas de Hadamard que devem ser adicionadas ao circuito são implementadas por:

```
qc.append(cirq.H(qr0))
```

e

```
qc.append(cirq.H(qr1)).
```

Os comandos para instanciar o simulador e simular o circuito são os mesmos do algoritmo anterior. O algoritmo 4 completo para o SDK Cirq é mostrado abaixo.

```
import cirq
qc = cirq.Circuit()
qr0 = cirq.NamedQubit('q[0]')
qr1 = cirq.NamedQubit('q[1]')
qc.append(cirq.H(qr0))
qc.append(cirq.H(qr1))
simulador = cirq.Simulator()
resultado = simulador.simulate(qc)
print(resultado)
```

O resultado obtido com a execução desse algoritmo é:

measurements: (no measurements)

output vector: 0.5$|00\rangle$ + 0.5$|01\rangle$ + 0.5$|10\rangle$ + 0.5$|11\rangle$

Não há medição do estado do qubit nesse algoritmo. Não temos um medidor quântico implementado para realizar essa operação. O resultado da simulação retorna o estado do qubit que nos diz que: $\alpha_0\alpha_1 = 0,5$, $\alpha_0\beta_1 = 0,5$, $\beta_0\alpha_1 = 0,5$ e $\beta_0\beta_1 = 0,5$. Logo, temos a probabilidade de 0,25 para cada um dos possíveis resultados.

A.7 Algoritmo 5 – Porta CNOT

A porta de Hadamard e a porta de Pauli X são portas quânticas de um qubit. Nesse algoritmo, será implementado a porta CNOT que também é chamada de NOT-controlada. A Figura 26 mostra as duas formas de representação da porta CNOT em circuitos quânticos.

A porta CNOT atua em dois qubits onde o primeiro é chamado de qubit de controle e

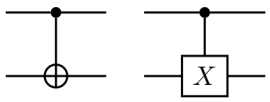

Figura 26 – Representações em circuitos quânticos da porta CNOT.

o segundo é chamado de qubit alvo. Se o qubit de controle encontra-se no estado $|1\rangle$, então o estado do qubit alvo é alterado de $|0\rangle$ para $|1\rangle$ ou de $|1\rangle$ para $|0\rangle$. Se o qubit de controle estiver no estado $|0\rangle$, o qubit alvo não sofre qualquer modificação. A porta CNOT pode ser comparada à porta lógica XOR porque o seu resultado pode ser representado por: CNOT$|a, b\rangle \rightarrow |a, a \oplus b\rangle$ onde a, b $\in$ {0,1}. A Figura 27 mostra o circuito quântico a ser implementado no algoritmo 5 para permitir a visualização da ação da porta CNOT sobre o qubit alvo. Encontram-se abaixo as versões do algoritmo 5 para os SDKs Qiskit, Forest e Cirq.

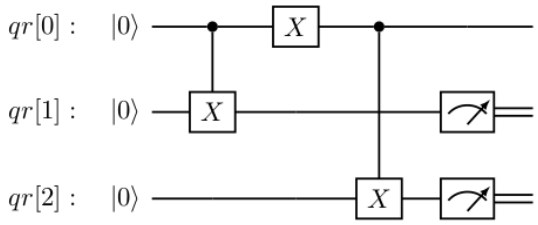

Figura 27 – Circuito quântico para o estudo da porta CNOT.

Qiskit

Nas primeiras linhas do algoritmo, deve-se importar as bibliotecas necessárias que são as mesmas do primeiro algoritmo. O motivo do uso dessas bibliotecas é apresentado na discussão do algoritmo 1. Então, devem ser instanciados três qubits, dois registradores clássicos e o circuito quântico. As linhas de comando necessárias para instanciar esses objetos são:

```
qr = QuantumRegister(3),
cr = ClassicalRegister(2)
```

e

```
qc = QuantumCircuit(qr,cr).
```

No Qiskit, a porta CNOT é implementada pelo método *cx()* que recebe dois argumentos: o qubit de controle e o qubit alvo. A primeira porta CNOT é implementada por:

```
qc.cx(qr[0],qr[1]).
```

Essa porta é seguida por uma porta de Pauli X sobre o primeiro qubit que é implementada através de:

```
qc.x(qr[0]).
```

A segunda porta CNOT, que aparece após a porta de Pauli X, é implementada por:

```
qc.cx(qr[0],qr[2]).
```

Por fim, medidores quânticos sobre o segundo e o terceiro qubit são adicionados ao circuito. Nesse algoritmo será utilizado o simulador *QASM Simulator* sendo atribuído o valor de 100 para a variável *shots* para realizar 100 medições. Os comandos finais para a seleção do *backend*, a execução e a impressão dos resultados são explicadas no algoritmo 1. O algoritmo 5 completo para o SDK Qiskit é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(3)
cr = ClassicalRegister(3)
qc = QuantumCircuit(qr,cr)
qc.cx(qr[0],qr[1])
qc.x(qr[0])
qc.cx(qr[0],qr[2])
qc.measure(qr[1],cr[0])
qc.measure(qr[2],cr[1])
backend = Aer.get_backend('qasm_simulator')
job = execute(qc,backend,shots=100)
resultado = job.result()
contagem = resultado.get_counts()
print(contagem)
```

O resultado da execução do algoritmo acima é {'10': 100}. Em todas as medições realizadas, o resultado para o qubit qr[2] será sempre 1 e, para o qubit qr[1], sempre 0. A primeira porta CNOT não altera o estado do qubit qr[1] porque o estado do qubit de controle, que é o qubit qr[0], é $|0\rangle$. A segunda porta CNOT altera o estado de qr[2] para $|1\rangle$ porque o qubit de controle qr[0] teve o seu estado alterado para $|1\rangle$ pela porta de Pauli X anterior à segunda porta CNOT.

Forest

As bibliotecas utilizadas são aquelas utilizadas no algoritmo 1. O circuito quântico é instanciado por:

```
p = Program().
```

Os dois registradores clássicos necessários para armazenar os resultados das medições são instanciados por:

```
ro = p.declare('ro', 'BIT', 2).
```

A porta CNOT é implementada no circuito pelo método *CNOT()* que recebe como primeiro

argumento o índice do qubit de controle e, como segundo argumento, o índice do qubit alvo. A primeira porta CNOT é implementada por:

```
p += CNOT(0,1).
```

Há a porta de Pauli X e, então, a segunda porta CNOT. Essas portas quânticas são implementadas por:

```
p += X(0)
```

e

```
p += CNOT(0,2).
```

Por fim, os medidores quânticos são implementados por:

```
p += MEASURE(1, ro[0])
```

e

```
p += MEASURE(2, ro[1]).
```

Uma máquina quântica virtual para simulação de sistemas com três qubits deve ser instanciada por:

```
qc = get_qc("3q-qvm").
```

O algoritmo completo é mostrado abaixo. As linhas de comando finais para a execução do algoritmo são discutidas no algoritmo 1.

```
from pyquil import get_qc, Program
from pyquil.gates import *
from pyquil.quilbase import Declare
p = Program()
ro = p.declare('ro', 'BIT', 2)
p += CNOT(0,1)
p += X(0)
p += CNOT(0,2)
p += MEASURE(1, ro[0])
p += MEASURE(2, ro[1])
qc = get_qc("3q-qvm")
executable = qc.compile(p)
result = qc.run(executable)
print(result.readout_data.get('ro'))
```

A execução desse algoritmo resulta em [[0 1]]. O primeiro valor refere-se ao resultado da medição de qr[1] e o segundo, resultado da medição de qr[2]. A ordem em que esses valores são exibidos na tela do terminal é oposta à ordem em que os valores são exibidos com o uso do SDK Qiskit.

Cirq

O algoritmo inicia-se com o comando para importar a biblioteca cirq:

```
import cirq.
```

Então, o circuito quântico e os três qubits são instanciados por:

```
qc = cirq.Circuit(),
qr0 = cirq.NamedQubit('qr[0]'),
qr1 = cirq.NamedQubit('qr[1]')
```

e

```
qr2 = cirq.NamedQubit('qr[2]').
```

A primeira porta CNOT é adicionada ao circuito por:

```
qc.append(cirq.CNOT(qr0,qr1)).
```

Essa porta não altera o estado de qr[1] porque o estado de qr[0] é $|0\rangle$. A porta de Pauli X é adicionada ao circuito por:

```
qc.append(cirq.X(qr0)).
```

O qubit qr[0] tem o seu estado alterado para $|1\rangle$. A segunda porta CNOT é implementada por:

```
qc.append(cirq.CNOT(qr0,qr2)).
```

Essa porta altera o estado de qr[3] para $|1\rangle$ porque, agora, o estado de qr[0] é $|1\rangle$. Os medidores quânticos foram adicionados por:

```
qc.append(cirq.measure(qr1,key = 'qr[1]'))
```

e

```
qc.append(cirq.measure(qr2,key = 'qr[2]')).
```

O simulador foi instanciado e o programa executado com o método *run()*. O algoritmo 5 para o SDK Cirq fica:

```
import cirq
qc = cirq.Circuit()
qr0 = cirq.NamedQubit('qr[0]')
qr1 = cirq.NamedQubit('qr[1]')
qr2 = cirq.NamedQubit('qr[2]')
qc.append(cirq.CNOT(qr0,qr1))
qc.append(cirq.X(qr0))
qc.append(cirq.CNOT(qr0,qr2))
qc.append(cirq.measure(qr1,key = 'qr[1]'))
qc.append(cirq.measure(qr2,key = 'qr[2]'))
simulador = cirq.Simulator()
resultado = simulador.run(qc,repetitions=10)
print(resultado)
```

O resultado da execução é:

q[1]=0000000000

q[2]=1111111111

Como nos algoritmos anteriores, o resultado da medição do qubit qr[1] é 0 e do qubit qr[2] é 1 em todas as medições realizadas. No resultado acima, temos dez valores 0 para o qubit qr[1] e dez valores 1 para o qubit qr[2] que correspondem às dez medições realizadas que é resultado da atribuição do valor 10 à variável *repetitions*.

A.8 Algoritmo 6 – Porta TOFFOLI

A porta quântica TOFFOLI atua em três qubits. O primeiro e o segundo qubit são qubits de controle, enquanto o terceiro qubit é o qubit alvo. Se os qubits de controle encontram-se no estado $|1\rangle$, então o estado do qubit alvo é alterado de $|0\rangle$ para $|1\rangle$ ou de $|1\rangle$ para $|0\rangle$. Se algum dos qubits de controle estiver no estado $|0\rangle$, o qubit alvo não sofre qualquer modificação. A porta TOFFOLI é representada por dois pontos, que marcam os qubits de controle, e uma porta de Pauli X no qubit alvo sendo que os pontos e a porta de Pauli X são ligados por uma linha vertical como mostrado na Figura 28.

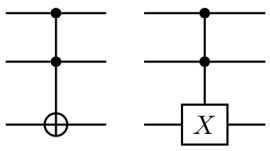

Figura 28 – Representações da porta TOFFOLI em circuitos quânticos.

Para demonstrar a atuação da porta TOFFOLI, será implementado o circuito quântico da Figura 29. Nesse circuito, a primeira porta TOFFOLI tem como qubits de controle os qubits qr[0] e qr[1] e, como qubit alvo, o qubit qr[2]. Os qubits qr[0] e qr[1] estão no estado inicial $|0\rangle$, portanto não há alteração no estado do qubit alvo qr[2]. Portas de Pauli X são aplicadas sobre os qubits qr[0] e qr[1] mudando os estados desses qubits para $|1\rangle$. Então, uma segunda porta TOFFOLI é implementada no circuito sendo qr[0] e qr[1] os qubits de controle e qr[3] o qubit alvo. Como qr[0] e qr[1] estão no estado $|1\rangle$, o estado de qr[3] é alterado de $|0\rangle$ para $|1\rangle$. As medições dos qubits qr[2] e qr[3] devem resultar em 0 e 1, respectivamente.

Qiskit

O sexto algoritmo para o SDK Qiskit inicia-se com as mesmas linhas de comando do algoritmo anterior para importar as bibliotecas necessárias ao algoritmo. Então, são instanciados quatro registradores quânticos e dois registradores clássicos. O circuito quântico também é instanciado. No Qiskit, a porta TOFFOLI é implementada pelo método *ccx()* que recebe como argumentos os qubits de controle e o qubit alvo. A primeira porta TOFFOLI é implementada por:

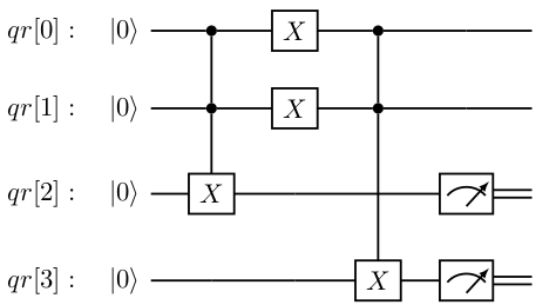

Figura 29 – Circuito quântico correspondente ao algoritmo 6 desse Tutorial para o estudo da porta quântica TOFFOLI.

`qc.ccx(qr[0],qr[1],qr[2]).`

A seguir, há duas portas de Pauli X, que são implementadas por:

`qc.x(qr[0])`

e

`qc.x(qr[1]).`

A segunda porta TOFFOLI é implementada por:

`qc.ccx(qr[0],qr[1],qr[3]).`

Por fim, são implementados os medidores quânticos. Os comandos utilizados para a seleção do *backend* e para a execução do algoritmo foram discutidos no primeiro algoritmo. O algoritmo 6 completo é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(4)
cr = ClassicalRegister(2)
qc = QuantumCircuit(qr,cr)
qc.ccx(qr[0],qr[1],qr[2])
qc.x(qr[0])
qc.x(qr[1])
qc.ccx(qr[0],qr[1],qr[3])
qc.measure(qr[2],cr[0])
qc.measure(qr[3],cr[1])
backend = Aer.get_backend ('qasm_simulator')
job = execute(qc,backend,shots=100)
resultado = job.result()
contagem = resultado.get_counts()
print(contagem)
```

O algoritmo foi executado pelo *QASM Simulator*. O número de execuções foi 100. O resultado que deve ser obtido é {'10': 100} que indica que o resultado da medição do qubit qr[3] é 1 e o resultado da medição do qubit qr[2] é 0 e que esses resultados foram obtidos em todas as 100 medições realizadas.

Forest

As bibliotecas utilizadas no algoritmo 6 para o SDK Forest são as mesmas utilizadas no algoritmo anterior. O circuito quântico é instanciado por:

```
p = Program().
```

No Forest, portas TOFFOLI são implementadas pelo método *CCNOT()* que recebe, como argumentos, os índices dos qubits de controle e do qubit alvo. A primeira porta TOFFOLI é implementada por:

```
p += CCNOT(0,1,2).
```

Antes da segunda porta TOFFOLI, são implementadas as duas portas de Pauli X para alterar os estados dos qubits de controle da segunda porta TOFFOLI. Essas portas quânticas são implementadas por:

```
p += X(0)
```

e

```
p += X(1).
```

A segunda porta TOFFOLI é implementada por:

```
p += CCNOT(0,1,3).
```

Os registradores clássicos são instanciados por:

```
ro = p.declare('ro', 'BIT', 2).
```

Por fim, os medidores quânticos são implementados através de:

```
p += MEASURE(2, ro[0])
```

e

```
p += MEASURE(3, ro[1]).
```

O algoritmo 6 requer uma máquina quântica virtual para quatro qubits que é instanciada por:

```
qc = get_qc("4q-qvm").
```

As linhas de comando finais do algoritmo 6 são explicadas no algoritmo 1. O algoritmo completo é mostrado a seguir:

```
from pyquil import get_qc, Program
from pyquil.gates import *
from pyquil.quilbase import Declare
p = Program()
p += CCNOT(0,1,2)
p += X(0)
p += X(1)
```

```
p += CCNOT(0,1,3)
ro = p.declare('ro', 'BIT', 2)
p += MEASURE(2, ro[0])
p += MEASURE(3, ro[1])
qc = get_qc("4q-qvm")
executable = qc.compile(p)
result = qc.run(executable)
print(result.readout_data.get('ro'))
```

O resultado da execução desse algoritmo é [[0 1]]. No resultado, o primeiro valor, que é 0, refere-se à medição do qubit com índice 2 e o segundo valor, que é 1, refere-se à medição do qubit com índice 3. A ordem em que os resultados das medições são impressos pelo SDK Forest é contrário à ordem de impressão dos resultados pelo SDK Qiskit.

Cirq

O algoritmo 6 para o SDK Cirq inicia-se com a importação da biblioteca *cirq*. Então, o circuito quântico deve ser instanciado por:

```
qc = cirq.Circuit().
```

Os quatro qubits necessários no circuito são instanciados por:

```
qr0 = cirq.NamedQubit('qr[0]'),
qr1 = cirq.NamedQubit('qr[1]'),
qr2 = cirq.NamedQubit('qr[2]')
```

e

```
qr3 = cirq.NamedQubit('qr[3]').
```

No Cirq, a porta TOFFOLI é implementada pelo método *TOFFOLI()* que recebe, como argumentos, as variáveis que identificam os qubits de controle e o qubit alvo. A primeira porta TOFFOLI é implementada no circuito seguida pelas duas portas de Pauli X e pela segunda porta TOFFOLI de acordo com o circuito da Figura 29. Essas portas quânticas são implementadas por:

```
qc.append(cirq.TOFFOLI(qr0,qr1,qr2)),
qc.append(cirq.X(qr0)),
qc.append(cirq.X(qr1))
```

e

```
qc.append(cirq.TOFFOLI(qr0,qr1,qr3)).
```

Então, os medidores quânticos são também implementados, o simulador é instanciado e o método *run()* é utilizado para a execução do algoritmo. O algoritmo 6 completo para o SDK Cirq é mostrado abaixo.

```
import cirq
qc = cirq.Circuit()
```

```
qr0 = cirq.NamedQubit('qr[0]')
qr1 = cirq.NamedQubit('qr[1]')
qr2 = cirq.NamedQubit('qr[2]')
qr3 = cirq.NamedQubit('qr[3]')
qc.append(cirq.TOFFOLI(qr0,qr1,qr2))
qc.append(cirq.X(qr0))
qc.append(cirq.X(qr1))
qc.append(cirq.TOFFOLI(qr0,qr1,qr3))
qc.append(cirq.measure(qr2,key = 'qr[2]'))
qc.append(cirq.measure(qr3,key = 'qr[3]'))
simulador = cirq.Simulator()
resultado = simulador.run(qc,repetitions=10)
print(resultado)
```

O resultado da execução é:
qr[2]=0000000000
qr[3]=1111111111

Nesse resultado, pode-se observar que a medição do qubit qr[2] sempre resulta no valor 0 e a medição do qubit qr[3] sempre resulta no valor 1 como observado na execução das versões do algoritmo 6 para os SDKs Qiskit e Forest.

A.9 Algoritmo 7 – Estado de Bell

A interação entre os qubits de um circuito quântico pode resultar em um estado quântico emaranhado conhecido como estado de Bell [9,10]. Nesse estado, o estado quântico de cada qubit não pode mais ser descrito independentemente. O estado emaranhado representa o sistema como um todo e o que ocorre em um qubit afeta os outros qubits do circuito. Para gerar um estado emaranhado, pode-se utilizar um circuito quântico com dois qubits, uma porta de Hadamard e uma porta CNOT. A porta de Hadamard é aplicada sobre o primeiro qubit que servirá como qubit de controle para a porta CNOT, que terá o segundo qubit como qubit alvo. A Figura 30 mostra o circuito quântico associado ao algoritmo 7 desse tutorial.

O estado do primeiro qubit é descrito por $|\psi_1\rangle = \alpha|0\rangle + \beta|1\rangle$ e o estado do segundo qubit por $|\psi_2\rangle = \gamma|0\rangle + \delta|1\rangle$. A superposição desses dois estados resulta no estado $|\psi\rangle = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$. Esse estado deve ser igual ao estado gerado pela ação da porta de Hadamard seguida pela porta CNOT que é o estado $|\psi\rangle = (|00\rangle + |11\rangle)/\sqrt{2}$. Porém, não é possível obter uma solução para a equação:

$$\alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle = (|00\rangle + |11\rangle)/\sqrt{2}. \quad \text{(A.5)}$$

Portanto, o estado emaranhado é um estado que não deveria existir. Experimentos recentes têm encontrado evidências de estados emaranhados de fótons [11] e elétrons [12]. Temos a seguir,

os algoritmos que implementam o circuito quântico da Figura 30 para os SDKs Qiskit, Forest e Cirq.

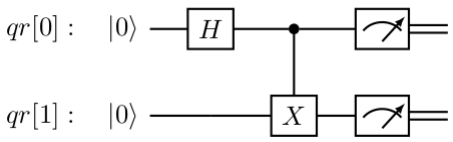

Figura 30 – Circuito quântico correspondente ao algoritmo 7 desse Tutorial para o estudo do estado de Bell.

Qiskit

Para implementar o circuito quântico da Figura 30 utilizando o SDK Qiskit, é necessário as bibliotecas *QuantumCircuit*, *QuantumRegister*, *ClassicalRegister*, *Aer* e *execute*. Essas bibliotecas são importadas pelos mesmos comandos utilizados no algoritmo anterior. Devem ser instanciados dois registradores quânticos e dois clássicos. Logo, as linhas de comando que devem ser inseridas no programa são:

```
qr = QuantumRegister(2)
```

e

```
cr = ClassicalRegister(2).
```

O circuito quântico é instanciado por:

```
qc = QuantumCircuit(qr,cr).
```

A porta de Hadamard e a porta CNOT são implementadas no circuito por:

```
qc.h(qr[0])
```

e

```
qc.cx(qr[0],qr[1]).
```

O circuito é finalizado com os medidores que são implementados no algoritmo por:

```
qc.measure(qr[0],cr[0])
```

e

```
qc.measure(qr[1],cr[1]).
```

O simulador a ser utilizado é o *QASM Simulator* que é selecionado por:

```
backend = Aer.get_backend('qasm_simulator').
```

Os comandos para a execução do algoritmo são os mesmos discutidos no algoritmo 1. O algoritmo 7 completo para o SDK Qiskit é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(2)
```

```
cr = ClassicalRegister(2)
qc = QuantumCircuit(qr,cr)
qc.h(qr[0])
qc.cx(qr[0],qr[1])
qc.measure(qr[0],cr[0])
qc.measure(qr[1],cr[1])
backend = Aer.get_backend('qasm_simulator')
job = execute(qc,backend,shots=100)
resultado = job.result()
contagem = resultado.get_counts()
print(contagem)
```

Como foi atribuído o valor 100 à variável *shots*, são realizadas cem medições que resultam em 00 ou 11. A probabilidade de obter o resultado 00 em um processo de medição é $\left|1/\sqrt{2}\right|^2$ = 0,5. Essa é a mesma probabilidade de obter o resultado 11. Um dos possíveis resultados obtidos na execução do algoritmo acima é {'11': 52, '00': 48}. Outro resultado possível é {'11': 44, '00': 56}

Forest

No Forest, as bibliotecas necessárias para a execução do algoritmo 7 são importadas pelos comandos:

```
from pyquil import get_qc, Program,
from pyquil.gates import *
```

e

```
from pyquil.quilbase import Declare.
```

O circuito quântico é instanciado por:

```
p = Program().
```

A porta de Hadamard e a porta CNOT são implementadas no circuito por:

```
p += H(0)
```

e

```
p += CNOT(0,1).
```

Os dois registradores clássicos necessários para armazenar os dados das medições realizadas são instanciados por:

```
ro = p.declare('ro', 'BIT', 2).
```

Os medidores quânticos são implementados por:

```
p += MEASURE(0, ro[0])
```

e

```
p += MEASURE(1, ro[1]).
```

O maior número de execuções é determinado pelo método *wrap_in_numshots_loop()*. A má-

quina quântica virtual para dois qubits é instanciada por:

```
qc = get_qc("2q-qvm").
```

O programa é compilado pelo método *compile()* e executado pelo método *run()*. O algoritmo 7 completo é mostrado abaixo.

```
from pyquil import get_qc, Program
from pyquil.gates import *
from pyquil.quilbase import Declare
p = Program()
p += H(0)
p += CNOT(0,1)
ro = p.declare('ro', 'BIT', 2)
p += MEASURE(0, ro[0])
p += MEASURE(1, ro[1])
p.wrap_in_numshots_loop(10)
qc = get_qc("2q-qvm")
executable = qc.compile(p)
result = qc.run(executable)
print(result.readout_data.get('ro'))
```

O resultado que deve ser obtido com a execução desse algoritmo é:

```
[ [0 0]
  [1 1]
  [0 0]
  [1 1]
  [0 0]
  [1 1]
  [1 1]
  [1 1]
  [1 1]
  [0 0] ]
```

Como explicado anteriormente, no Forest, obtemos os resultados das medições realizadas em uma matriz. Cada linha dessa matriz contém um elemento que armazena os resultados das medições realizadas em uma execução do algoritmo. Como foram executadas duas medições em cada execução, temos dois resultados em cada elemento dessa matriz. O primeiro valor corresponde ao resultado da medição do qubit qr[0] e o segundo valor, ao resultado da medição do qubit qr[1]. O resultado [1 1] corresponde ao resultado qr[0] = 1 e qr[1] = 1, que foi obtido seis vezes, e o resultado [0 0] corresponde ao resultado qr[0] = 0 e qr[1] = 0, que foi obtido quatro vezes. Em qualquer medição, a probabilidade de obter o resultado [1 1] é 0,5 e a probabilidade de obter o resultado [1 1] também é 0,5.

Cirq

Para implementar o circuito quântico da Figura 30 usando o SDK Cirq, deve-se importar a biblioteca cirq através de:

`import cirq.`

Então, instancia-se o circuito quântico por:

`qc = cirq.Circuit().`

Os dois qubits necessários no circuito são implementados no circuito por:

`qr0 = cirq.NamedQubit('qr[0]')`

e

`qr1 = cirq.NamedQubit('qr[1]').`

Sobre o qubit qr[0] deve ser aplicada uma porta de Hadamard por:

`qc.append(cirq.H(qr0)).`

A porta CNOT é adicionada ao circuito por:

`qc.append(cirq.CNOT(qr0,qr1)).`

As medições dos qubits são realizadas por:

`qc.append(cirq.measure(qr0,key = 'qr[0]'))`

e

`qc.append(cirq.measure(qr1,key = 'qr[1]')).`

O simulador é instanciado por:

`simulador = cirq.Simulator()`

e o comando para executar o programa é:

`resultado = simulador.run(qc,repetitions=10).`

O algoritmo completo é mostrado a seguir:

```
import cirq
qc = cirq.Circuit()
qr0 = cirq.NamedQubit('qr[0]')
qr1 = cirq.NamedQubit('qr[1]')
qc.append(cirq.H(qr0))
qc.append(cirq.CNOT(qr0,qr1))
qc.append(cirq.measure(qr0,key = 'qr[0]'))
qc.append(cirq.measure(qr1,key = 'qr[1]'))
simulador = cirq.Simulator()
resultado = simulador.run(qc,repetitions=10)
print(resultado)
```

Um exemplo de resultado para a execução desse algoritmo é:

qr[0]=0000011010

qr[1]=0000011010

No Cirq, os resultados de cada medição são apresentados em sequência após o sinal de

igualdade que segue a variável que identifica o qubit medido. No resultado apresentado acima, a primeira medição resulta em 0 para qr[0] e 0 para qr[1]. A segunda, a terceira, a quarta, a quinta, a oitava e a décima medição também resultam nos mesmos valores para qr[0] e qr[1]. A sexta, a sétima e a nona medição resultam em 1 para qr[0] e 1 para qr[1]. A probabilidade para o resultado qr[0] = 0 e qr[1] = 0 é 0,5. Essa é a mesma probabilidade de se obter o resultado qr[0] = 1 e qr[1] = 1. Logo, qualquer combinação desses resultados é possível. Por exemplo, outro resultado possível seria:
qr[0]=1110001010
qr[1]=1110001010

A.10 Algoritmo 8 – *Full Adder* quântico

O circuito quântico equivalente ao circuito lógico *Full Adder* pode ser construído com o uso de portas CNOT e TOFFOLI. As portas de Pauli X são utilizadas para modificar os estados iniciais dos qubits para obter os valores a serem somados. A figura abaixo mostra o circuito quântico *Full Adder*. Os qubits qr[0] e qr[1] armazenam os valores a serem somados. Na figura, uma porta de Pauli X foi aplicada ao qubit qr[0] para modificar o seu estado de $|0\rangle$ para $|1\rangle$ para que seja realizada a soma de 1 e 0. O qubit qr[2] equivale ao bit *carry in* e o qubit qr[3] equivale ao bit *carry out*. Outra porta de Pauli X foi aplicada ao qubit qr[2] alterando o seu estado inicial para $|1\rangle$. Assim, esse circuito realizará a soma de 1 e 0 com *carry in* igual a 1. O resultado dessa soma deve ser 2. Para obter o resultado, os qubits a serem medidos são qr[2] e qr[3] e os valores medidos devem ser 0 e 1, respectivamente.

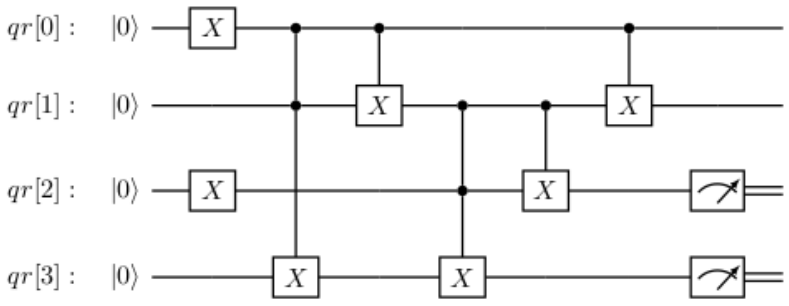

Figura 31 – Circuito quântico correspondente ao algoritmo 8 desse Tutorial para o estudo do *Full Adder* quântico.

Qiskit

Inicia-se o algoritmo com a importação das bibliotecas: *QuantumCircuit*, *QuantumRegister*, *ClassicalRegister*, *Aer* e *execute*. São necessários quatro qubits que devem ser instanciados por:

```
qr = QuantumRegister(4).
```

Os resultados das medições dos qubits qr[2] e qr[3] devem ser armazenados em dois registradores clássicos instanciados por:

```
cr = ClassicalRegister(2).
```

O circuito quântico é instanciado por:

```
qc = QuantumCircuit(qr,cr).
```

Os qubits qr[0] e qr[2] devem ter seu estado alterado de $|0\rangle$ para $|1\rangle$. Para isso, portas de Pauli X são aplicadas sobre esses qubits através de:

```
qc = qc.x(qr[0])
```

e

```
qc = qc.x(qr[2]).
```

Aqui, os estados dos qubits são: qr[0] no estado $|1\rangle$, qr[1] no estado $|0\rangle$, qr[2] no estado $|1\rangle$ e qr[3] no estado $|0\rangle$. Então, adiciona-se ao circuito uma porta TOFFOLI com os qubits qr[0] e qr[1] como qubits de controle e qr[3] como qubit alvo por meio do método:

```
qc.ccx(qr[0],qr[1],qr[3]).
```

O qubit qr[3] não tem o seu estado alterado. Uma porta CNOT é adicionada ao circuito por:

```
qc.cx(qr[0],qr[1]).
```

Essa porta tem qr[0] como qubit de controle e qr[1] como qubit alvo. Como o estado de qr[0] é $|1\rangle$, o estado de qr[1] é alterado para $|1\rangle$. Em seguida, adiciona-se ao circuito a segunda porta TOFFOLI por:

```
qc.ccx(qr[1],qr[2],qr[3]).
```

Essa porta altera o estado de qr[3] para $|1\rangle$ porque tanto qr[1] quanto qr[2] estão no estado $|1\rangle$. Duas portas quânticas CNOT são implementadas no circuito por:

```
qc.cx(qr[1],qr[2])
```

e

```
qc.cx(qr[0],qr[1]).
```

A primeira altera o estado de qr[2] de $|1\rangle$ para $|0\rangle$ e a segunda altera o estado de qr[1] para $|0\rangle$. Então, adicionam-se os operadores que realizam as medições nos qubits qr[2] e qr[3] através de:

```
qc.measure(qr[2],cr[0])
```

e

```
qc.measure(qr[3],cr[1]).
```

O algoritmo é finalizado com os comandos:

```
backend = Aer.get_backend('qasm_simulator'),
job = execute(qc,backend,shots=100),
resultado = job.result(),
contagem = resultado.get_counts()
```

e

```
print(contagem).
```

O algoritmo 8 completo para o SDK Qiskit é mostrado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(4)
cr = ClassicalRegister(2)
qc = QuantumCircuit(qr,cr)
qc.x(qr[0])
qc.x(qr[2])
qc.ccx(qr[0],qr[1],qr[3])
qc.cx(qr[0],qr[1])
qc.ccx(qr[1],qr[2],qr[3])
qc.cx(qr[1],qr[2])
qc.cx(qr[0],qr[1])
qc.measure(qr[2],cr[0])
qc.measure(qr[3],cr[1])
backend = Aer.get_backend('qasm_simulator')
job = execute(qc,backend,shots=100)
resultado = job.result()
contagem = resultado.get_counts()
print(contagem)
```

O resultado desse algoritmo será: {'10': 100}. Nas cem execuções foi obtido o resultado 10, ou seja, qr[3] = 1 e qr[2] = 0. O grande número de execuções mostra que esse é o único resultado que pode ser obtido, portanto a probabilidade desse resultado é 1. O resultado 10, que deve-se a 1 para o bit *carry out* e 0 para o bit de soma, equivale ao número decimal 2 em notação binária.

Forest

O algoritmo 8 para o SDK Forest inicia-se com a importação das bibliotecas necessárias para a execução do programa. Essas são as mesmas utilizadas no algoritmo anterior. Então, o circuito quântico é instanciado por:

```
p = Program().
```

Os qubits qr[0] e qr[2] têm os seus estados alterados pelas portas de Pauli X implementadas por:

```
p += X(0)
```

e

```
p += X(2).
```

A primeira porta TOFFOLI é implementada por:

```
p += CCNOT(0,1,3).
```

Essa porta quântica não altera o estado do seu qubit alvo. Em seguida, é adicionado ao circuito uma porta CNOT pela linha de comando:

```
p += CNOT(0,1).
```

A segunda porta TOFFOLI é adicionada por:

```
p += CCNOT(1,2,3).
```

As duas portas CNOT finais são implementadas por:

```
p += CNOT(1,2),
```

e

```
p += CNOT(0,1).
```

Os registradores clássicos são instanciados por:

```
ro = p.declare('ro', 'BIT', 2).
```

Esses registradores são necessários para armazenar os resultados das medições dos qubits qr[2] e qr[3] que foram medidos pelos medidores implementados por:

```
p += MEASURE(2, ro[0])
```

e

```
p += MEASURE(2, ro[0]).
```

As linhas de comando finais para a compilação e execução do algoritmo são as mesmas adicionadas no algoritmo anterior. O algoritmo completo é mostrado abaixo.

```
from pyquil import get_qc, Program
from pyquil.gates import *
from pyquil.quilbase import Declare
p = Program()
p += X(0)
p += X(2)
p += CCNOT(0,1,3)
p += CNOT(0,1)
p += CCNOT(1,2,3)
p += CNOT(1,2)
p += CNOT(0,1)
ro = p.declare('ro', 'BIT', 2)
p += MEASURE(2, ro[0])
p += MEASURE(3, ro[1])
qc = get_qc("4q-qvm")
executable = qc.compile(p)
result = qc.run(executable)
print(result.readout_data.get('ro'))
```

O resultado da execução do algoritmo acima é [[0 1]]. Esse resultado é expresso em como uma matriz que contém um único elemento que é o par de valores 0 e 1. O valor 0 é o resultado da medição do qubit qr[2] e o valor 1 é o resultado da medição do qubit qr[3]. O qubit qr[2] equivale ao bit que contém o resultado da soma e o qubit qr[3] equivale ao bit de *carry out*. Logo, o resultado [[0 1]] equivale ao número binário 10 que é o número decimal 2.

Cirq

No Cirq, o algoritmo 8 inicia-se com o comando para importar a biblioteca *cirq*:
`import cirq.`
Então, o circuito quântico e os quatro qubits são instanciados por:
`qc = cirq.Circuit(),`
`qr0 = cirq.NamedQubit('qr[0]'),`
`qr1 = cirq.NamedQubit('qr[1]'),`
`qr2 = cirq.NamedQubit('qr[2]')`
e
`qr3 = cirq.NamedQubit('qr[3]').`
As portas quânticas são implementadas no circuito pelo método *append()*. A sequência de comandos que implementam as portas quânticas do circuito são:
`qc.append(cirq.X(qr0)),`
`qc.append(cirq.X(qr2)),`
`qc.append(cirq.TOFFOLI(qr0,qr1,qr3)),`
`qc.append(cirq.CNOT(qr0,qr1)),`
`qc.append(cirq.TOFFOLI(qr1,qr2,qr3)),`
`qc.append(cirq.CNOT(qr1,qr2))`
e
`qc.append(cirq.CNOT(qr0,qr1)).`
Os medidores quânticos são implementados por:
`qc.append(cirq.measure(qr2,key = 'qr[2]'))`
e
`qc.append(cirq.measure(qr3,key = 'qr[3]')).`
Os comandos finais do algoritmo são:
`simulador = cirq.Simulator(),`
que instancia o simulador, e
`resultado = simulador.run(qc,repetitions=10),`
que executa o método *run()*. O algoritmo completo é mostrado abaixo.

```
import cirq
qc = cirq.Circuit()
qr0 = cirq.NamedQubit('qr[0]')
```

```
qr1 = cirq.NamedQubit('qr[1]')
qr2 = cirq.NamedQubit('qr[2]')
qr3 = cirq.NamedQubit('qr[3]')
qc.append(cirq.X(qr0))
qc.append(cirq.X(qr2))
qc.append(cirq.TOFFOLI(qr0,qr1,qr3))
qc.append(cirq.CNOT(qr0,qr1))
qc.append(cirq.TOFFOLI(qr1,qr2,qr3))
qc.append(cirq.CNOT(qr1,qr2))
qc.append(cirq.CNOT(qr0,qr1))
qc.append(cirq.measure(qr2,key = 'qr[2]'))
qc.append(cirq.measure(qr3,key = 'qr[3]'))
simulador = cirq.Simulator()
resultado = simulador.run(qc,repetitions=10)
print(resultado)
```

O resultado da execução do algoritmo acima é:
qr[2]=0000000000
qr[3]=1111111111

Temos o mesmo resultado em todas as medições realizadas. Para o qubit qr[3], que equivale ao bit *carry out*, o valor é 1 e para o qubit qr[2], que equivale ao resultado da soma, o valor é 0. Esse resultado equivale ao número binário 10 que é o número decimal 2.

A.11 Algoritmo 9 – *Full Subtractor* quântico

A Figura 32 mostra o circuito quântico que realiza a operação de subtração. Esse circuito quântico é chamado de *Full Subtractor* Quântico em analogia ao circuito lógico *Full Subtractor*. Considerando as entradas, o qubit qr[0] equivale ao bit *borrow in*, o qubit qr[1] equivale ao bit que representa minuendo e o qubit qr[2] equivale ao bit que representa o subtraendo. Nas saída, o qubit qr[2] equivale à diferença e o qubit qr[3] equivale ao bit *borrow out*. Portas de Pauli X são utilizadas para alterar os estados iniciais dos qubits de $|0\rangle$ para $|1\rangle$. Na Figura 32, os qubits qr[0] e qr[1] têm os seus estados iniciais alterados para $|1\rangle$ o que equivale a realizar a operação de subtração em um *Full Subtractor* clássico com *borrow in* igual a 1, minuendo também igual a 1 e subtraendo igual a 0. O resultado esperado para essa operação é 0 para a diferença e 0 para o *borrow out*.

Qiskit

No Qiskit, as bibliotecas necessárias implementar o circuito quântico da Figura 32 são:

```
from qiskit import QuantumCircuit,
```

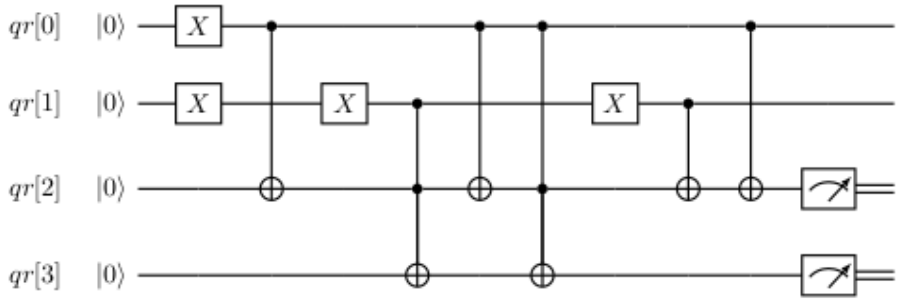

Figura 32 – Circuito quântico correspondente ao algoritmo 9 desse Tutorial para o estudo do *Full Subtractor* quântico.

```
from qiskit import QuantumRegister,
from qiskit import ClassicalRegister,
from qiskit import Aer
```

e

```
from qiskit import execute.
```

São necessários quatro qubits e dois bits clássicos. Os registradores quântico e clássico são instanciados pelos comandos:

```
qr = QuantumRegister(4)
```

e

```
cr = ClassicalRegister(2).
```

Os registradores quântico e clássico são dados como argumento ao construtor *QuantumCircuit()* quando o circuito quântico é instanciado através de:

```
qc = QuantumCircuit(qr,cr).
```

As portas de Pauli X utilizadas para definir os estados dos qubits de entrada são implementadas por:

```
qc.x(qr[0])
```

e

```
qc.x(qr[1]).
```

A primeira porta CNOT tem qr[0] como qubit de controle e qr[2] como qubit alvo. Essa porta quântica é implementada por:

```
qc.cx(qr[0],qr[2]).
```

Como o estado inicial de qr[0] foi alterado para $|1\rangle$ pela porta de Pauli X, o estado de qr[2] será alterado para $|1\rangle$. O qubit qr[1] volta ao estado $|0\rangle$ pela ação da porta de Pauli X que segue a porta CNOT. A primeira porta TOFFOLI é implementada por:

```
qc.ccx(qr[1],qr[2],qr[3]).
```

Essa porta não altera o estado de qr[3] porque qr[1] está no estado $|0\rangle$.

O qubit qr[2] volta ao estado $|0\rangle$ devido à ação da porta CNOT

`qc.cx(qr[0],qr[2]).`
A porta TOFFOLI,
`qc.ccx(qr[0],qr[2],qr[3]),`
não altera o estado de qr[3] porque qr[2] está no estado $|0\rangle$. A porta de Pauli X,
`qc.x(qr[1]),`
altera o estado de qr[1] para $|1\rangle$ que atua como qubit de controle da porta CNOT
`qc.cx(qr[1],qr[2]).`
Essa porta altera o estado de qr[2] para $|1\rangle$, mas a última porta CNOT,
`qc.cx(qr[0],qr[2]),`
faz com que o estado desse qubit volte para $|0\rangle$.

Ao final do algoritmo temos a implementação dos medidores quânticos por:
`qc.measure(qr[2],cr[0])`
e
`qc.measure(qr[3],cr[1]).`
As linhas de comando para a execução do algoritmo são as mesmas implementadas nos algoritmos anteriores com o uso do simulador *QASM Simulator*. O algoritmo completo é apresentado a seguir:

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from qiskit import Aer
from qiskit import execute
qr = QuantumRegister(4)
cr = ClassicalRegister(2)
qc = QuantumCircuit(qr,cr)
qc.x(qr[0])
qc.x(qr[1])
qc.cx(qr[0],qr[2])
qc.x(qr[1])
qc.ccx(qr[1],qr[2],qr[3])
qc.cx(qr[0],qr[2])
qc.ccx(qr[0],qr[2],qr[3])
qc.x(qr[1])
qc.cx(qr[1],qr[2])
qc.cx(qr[0],qr[2])
qc.measure(qr[2],cr[0])
qc.measure(qr[3],cr[1])
backend = Aer.get_backend('qasm_simulator')
job = execute(qc,backend,shots=100)
```

```
resultado = job.result()
contagem = resultado.get_counts()
print(contagem)
```

O resultado desse algoritmo será: {'00': 100}. As cem execuções realizadas obtiveram o mesmo resultado. Logo, a probabilidade para o resultado 00 é 1. Esse é o único resultado que pode ser obtido com a execução desse algoritmo. O resultado 00, que indica que qr[3] = 0 e qr[2] = 0, equivale ao número binário 00 que é o número decimal 0 como esperado.

Forest

No Forest, as bibliotecas necessárias para a execução do algoritmo que implementa o circuito para o *Full Subtractor* são importadas por:

```
from pyquil import get_qc, Program,
from pyquil.gates import *
```

e

```
from pyquil.quilbase import Declare.
```

O circuito quântico é instanciado por:

```
p = Program().
```

Os qubits qr[0] e qr[1] têm os seus estados alterados pela ação das portas de Pauli X que são implementadas por:

```
p += X(0)
```

e

```
p += X(1).
```

A porta CNOT

```
p += CNOT(0,2)
```

altera o estado de qr[2] para $|1\rangle$. O qubit qr[1] volta ao estado $|0\rangle$ devido a

```
p += X(1),
```

portanto a porta TOFFOLI

```
p += CCNOT(1,2,3)
```

não altera o estado de qr[3]. A porta CNOT

```
p += CNOT(0,2)
```

altera o estado de qr[2] para $|0\rangle$. Logo, a porta TOFFOLI

```
p += CCNOT(0,2,3)
```

também não altera o estado de qr[3]. O qubit qr[1] volta ao estado $|1\rangle$ pela ação da porta quântica

```
p += X(1)
```

de modo que

```
p += CNOT(1,2)
```

altera o estado de qr[2] para $|1\rangle$ e

```
p += CNOT(0,2)
```

altera o estado de qr[2] para $|0\rangle$. Um registrador clássico contendo dois bits para armazenar o resultado das medições dos dois qubits é instanciado pelo método *declare()*. Uma máquina quântica virtual de quatro qubits é instanciada e os comandos para compilar e executar são implementado no algoritmo. O algoritmo completo é mostrado abaixo.

```
from pyquil import get_qc, Program
from pyquil.gates import *
from pyquil.quilbase import Declare
p = Program()
p += X(0)
p += X(1)
p += CNOT(0,2)
p += X(1)
p += CCNOT(1,2,3)
p += CNOT(0,2)
p += CCNOT(0,2,3)
p += X(1)
p += CNOT(1,2)
p += CNOT(0,2)
ro = p.declare('ro', 'BIT', 2)
p += MEASURE(2, ro[0])
p += MEASURE(3, ro[1])
qc = get_qc("4q-qvm")
executable = qc.compile(p)
result = qc.run(executable)
print(result.readout_data.get('ro'))
```

O resultado da execução do algoritmo acima é [[0 0]] que é uma matriz com um único elemento. O par de valores 0 e 0 é o elemento dessa matriz. O primeiro 0 é o resultado da medição do qubit qr[2] e o segundo 0, o resultado da medição do qubit qr[3]. Esses resultados indicam que o resultado da operação de subtração é o número binário 00 que é o número decimal 0.

Cirq

No Cirq, o algoritmo inicia-se com a importação da biblioteca *cirq*. Então, o circuito quântico e os qubits são instanciados por:

```
qc = cirq.Circuit(),qr0 = cirq.NamedQubit('q[0]'),
qr1 = cirq.NamedQubit('q[1]'),
qr2 = cirq.NamedQubit('q[2]'),
```

e

```
qr3 = cirq.NamedQubit('q[3]').
```

As portas de Pauli X, que são utilizadas para estabelecer os estados iniciais dos qubits, são implementadas por:

```
qc.append(cirq.X(qr0))
```

e

```
qc.append(cirq.X(qr1)).
```

A primeira porta CNOT é implementada por:

```
qc.append(cirq.CNOT(qr0,qr2)).
```

O qubit qr[1] volta ao estado $|0\rangle$ pela ação da porta de Pauli X implementada por:

```
qc.append(cirq.X(qr1)).
```

Logo, a porta TOFFOLI

```
qc.append(cirq.TOFFOLI(qr1,qr2,qr3))
```

não altera o estado de qr[3]. A porta CNOT

```
qc.append(cirq.CNOT(qr0,qr2))
```

altera o estado de qr[2] para $|0\rangle$. Assim, a porta TOFFOLI

```
qc.append(cirq.TOFFOLI(qr0,qr2,qr3))
```

não altera o estado de qr[3]. O qubit qr[1] tem o seu estado alterado para $|1\rangle$ por:

```
qc.append(cirq.X(qr1)).
```

Assim, a porta CNOT

```
qc.append(cirq.CNOT(qr1,qr2))
```

altera o estado de qr[2] para $|'\rangle$ e a última porta CNOT

```
qc.append(cirq.CNOT(qr0,qr2))
```

altera o estado de qr[2] para $|0\rangle$.

Os qubits qr[2] e qr[3] são medidos por:

```
qc.append(cirq.measure(qr2,key = 'q[2]'))
```

e

```
qc.append(cirq.measure(qr3,key = 'q[3]')).
```

O simulador é instanciado por:

```
simulador = cirq.Simulator().
```

O método *run()*, que é utilizado na execução do algoritmo, é implementado por:

```
resultado = simulador.run(qc,repetitions=10).
```

O algoritmo completo é mostrado a seguir:

```
import cirq
qc = cirq.Circuit()
qr0 = cirq.NamedQubit('q[0]')
qr1 = cirq.NamedQubit('q[1]')
qr2 = cirq.NamedQubit('q[2]')
qr3 = cirq.NamedQubit('q[3]')
```

```
qc.append(cirq.X(qr0))
qc.append(cirq.X(qr1))
qc.append(cirq.CNOT(qr0,qr2))
qc.append(cirq.X(qr1))
qc.append(cirq.TOFFOLI(qr1,qr2,qr3))
qc.append(cirq.CNOT(qr0,qr2))
qc.append(cirq.TOFFOLI(qr0,qr2,qr3))
qc.append(cirq.X(qr1))
qc.append(cirq.CNOT(qr1,qr2))
qc.append(cirq.CNOT(qr0,qr2))
qc.append(cirq.measure(qr2,key = 'q[2]'))
qc.append(cirq.measure(qr3,key = 'q[3]'))
simulador = cirq.Simulator()
resultado = simulador.run(qc,repetitions=10)
print(resultado)
```

O resultado da execução do algoritmo acima é:
q[2]=0000000000
q[3]=0000000000

Em todas as execuções, o valor 0 foi o resultado das medições dos qubits qr[2] e qr[3]. Foram realizadas dez execuções que se deve ao valor 10 atribuído à variável *repetitions*. O resultado 0 para qr[2] e 0 para qr[3] equivale ao número binário 00 que é o número decimal 0.

A.12 Referências

[1] ARUTE, F. et al. Quantum supremacy using a programmable superconducting processor. Nature, Nature Publishing Group, v. 574, n. 7779, p. 505–510, 2019.

[2] PEDNAULT, E. et al. Leveraging secondary storage to simulate deep 54-qubit sycamore circuits. arXiv preprint arXiv:1910.09534, 2019.

[3] Disponível em: https://quantumai.google/hardware.

[4] Disponível em: https://www.qureca.com/overview-on-quantum-initiatives-worldwide.

[5] Disponível em: https://www.ibm.com/blogs/research/2020/01/quantum-limited-amplifiers/.

[6] CROSS, A. W. et al. Open quantum assembly language. arXiv preprint arXiv:1707.03429, 2017.

[7] KHAMMASSI, N. et al. cQASM v1. 0: Towards a common quantum assembly language. arXiv preprint arXiv:1805.09607, 2018.

[8] SMITH, R. S.; CURTIS, M. J.; ZENG, W. J. A practical quantum instruction set architecture.

arXiv preprint arXiv:1608.03355, 2016.

[9] NIELSEN, M. A.; CHUANG, I. L. Quantum Computation and Quantum Information: 10th Anniversary Edition. [S.l.]: Cambridge University Press, 2010.

[10] PORTUGAL, R.; MARQUEZINO, F. Introdução à programação de computadores quânticos. Sociedade Brasileira de Computação, 2019.

[11] MOREAU, P.-A. et al. Imaging bell-type nonlocal behavior. Science advances, American Association for the Advancement of Science, v. 5, n. 7, p. eaaw2563, 2019.

[12] QIAO, H. et al. Conditional teleportation of quantum-dot spin states. Nature communications, Nature Publishing Group, v. 11, n. 1, p. 1–9, 2020.