Revista Brasileira de Ensino de Física, vol. 43, e20210033 (2021)
www.scielo.br/rbef
DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

Artigos Gerais

Computação quântica: uma abordagem para a graduação usando o Qiskit

Quantum computing: an undergraduate approach using Qiskit

Gleydson Fernandes de Jesus[1], Maria Heloísa Fraga da Silva[1], Teonas Gonçalves Dourado Netto[1], Lucas Queiroz Galvão[1], Frankle Gabriel de Oliveira Souza[1], Clebson Cruz[*1]

[1]Universidade Federal do Oeste da Bahia, Centro de Ciências Exatas e das Tecnologias, Grupo de Informação Quântica, CEP 47810-059 Barreiras, BH, Brasil.

Recebido em 26 de janeiro de 2021. Revisado em 06 de abril de 2021. Aceito em 18 de maio de 2021.

Neste artigo, apresentamos a ferramenta Quantum Information Software Development Kit – *Qiskit* para o ensino de computação quântica para estudantes de graduação com conhecimento básico dos postulados da mecânica quântica. Nos concentramos na apresentação da construção dos programas em qualquer laptop ou desktop comum e a sua execução em processadores quânticos reais através do acesso remoto aos *hardwares* disponibilizados na plataforma *IBM Quantum Experience*. Os códigos são disponibilizados ao longo do texto para que os leitores, mesmo com pouca experiência em computação científica, possam reproduzi-los e adotar os métodos discutidos neste artigo para abordar seus próprios projetos de computação quântica. Os resultados apresentados estão de acordo com as previsões teóricas, mostrando a eficácia do pacote *Qiskit* como uma ferramenta de trabalho em sala de aula para a introdução de conceitos aplicados de computação e informação quântica.
Palavras-Chave: Python, IBM, Qiskit, Quantum Experience.

In this paper, we present the Quantum Information Software Development Kit – *Qiskit* for teaching quantum computing to undergraduate students, with basic knowledge of quantum mechanics postulates. We focus on presenting the construction of the programs on any common laptop or desktop computer and their execution on real quantum processors through the remote access to the *quantum hardware* available on the *IBM Quantum Experience* platform. The codes are made available throughout the text so that readers, even with little experience in scientific computing, can reproduce them and adopt the methods discussed in this paper to address their own quantum computing projects. The results presented are in agreement with theoretical predictions and show the effectiveness of the *Qiskit* package as a robust classroom working tool for the introduction of applied concepts of quantum computing and quantum information theory.
Keywords: Python, IBM, Qiskit, Quantum Experience.

1. Introdução

Com o advento do *IBM Quantum Experience* (IBM QE) [1–4], houve uma facilitação ao acesso a plataformas de computação quântica [5–8] por qualquer pessoa com acesso à internet através de um computador doméstico [3, 4]. Entretanto, a maioria dos estudantes dos cursos de ciências exatas e tecnológicas não são apresentados aos conceitos fundamentais da computação quântica até a pós-graduação.

Nos últimos anos, os avanços apresentados pela computação quântica têm mostrado o seu o potencial de revolução tecnológica [9, 10]. Nesse cenário, a computação científica dos próximos anos será liderada por aqueles que têm o conhecimento acerca da utilização de dispositivos quânticos. Portanto, se torna crucial facilitar o acesso à educação científica, garantindo que os estudantes, independentemente de planejarem trabalhar em uma área relacionada à teoria da informação quântica, aprendam conceitos básicos de computação quântica.

Nesse contexto, a apresentação da área de computação e informação quântica para alunos de graduação tem atraído a atenção da comunidade científica nos últimos anos [2–4, 6, 11–15]. Além de contextualizar o processo de ensino e aprendizagem no cotidiano, alguns estudos apontam que o uso de tecnologias como recurso auxiliar de aprendizagem constitui uma realidade para a maior parte dos estudantes, sendo um caminho profícuo para a consolidação do que se compreende como democratização e universalização do conhecimento [16, 17], de modo que trabalhos recentes têm apostado na plataforma IBM QE como aliada das práticas pedagógicas, propondo, inclusive, abordagens didáticas para o ensino de computação quântica no nível de graduação [2–4] e até mesmo no ensino médio [15, 18].

[*] Endereço de correspondência: clebson.cruz@ufob.edu.br

Em 2017, a IBM (International Business Machines) disponibilizou o seu kit de desenvolvimento de *software* para informação quântica (Quantum Information Software Development Kit), ou simplesmente *Qiskit* [19–23], permitindo o desenvolvimento de *softwares* para seu serviço de computação quântica em nuvem [1].

Neste trabalho, analisamos o pacote *Qiskit*, usando a linguagem Python 3 [24, 25], como um recurso educacional para aulas de computação quântica para a graduação em Física e áreas afins. Mostramos como essa pode ser uma ferramenta poderosa para o ensino de computação quântica, com foco na implementação de circuitos quânticos simples e algoritmos quânticos bem conhecidos. Apresentamos as principais condições para a construção dos programas e a sua execução em processadores quânticos reais, ou até mesmo em computadores domésticos. Os códigos são disponibilizados nos Boxes ao longo do texto, de modo que os leitores possam adotar os métodos discutidos neste artigo para abordar seus próprios projetos de computação quântica.

Esse trabalho está estruturado seguindo um roteiro básico para a introdução de conceitos fundamentais para a computação quântica, como qubits, portas quânticas, emaranhamento e algoritmos quânticos. Fazemos uma apresentação das ferramentas computacionais necessárias para abordar nossos projetos de computação quântica em computadores domésticos; apresentamos os principais conceitos básicos de computação e informação quântica, como bits quânticos, portas quânticas básicas, medidas e emaranhamento quântico; e fornecemos um conjunto de aplicações, como a construção de portas lógicas clássicas a partir de portas quânticas, o famoso algoritmo de teleporte quântico [3–5, 8] e o algoritmo de busca de Grover [5, 14], executados em processadores quânticos reais. Estas aplicações podem ser usadas como exemplos de implementação de algoritmos quânticos em sala de aula, apresentando o *Qiskit* como uma ferramenta de trabalho útil para a apresentação de conceitos básicos de computação e informação quântica para uma ampla gama de estudantes com um conhecimento básico de mecânica quântica e até mesmo nenhuma experiência em programação científica.

2. Ferramentas Computacionais

A linguagem Python [24, 25] foi projetada para ser de fácil leitura, com rápido desenvolvimento de código e de fácil compreensão, pois tem pouco foco na sintaxe e um foco maior nos conceitos básicos de lógica de programação. No entanto, apesar da flexibilidade, o Python é considerado lento em comparação com outras linguagens, mas isso é compensado por sua biblioteca robusta e fácil de manipular, adequada para cálculos científicos [24–27]. Nesse quesito, a utilização do Python para o *Qiskit* permite que os conteúdos apresentados neste artigo possam ser reproduzidos pela maioria dos leitores, mesmo aqueles que têm pouca ou nenhuma experiência com essa linguagem de programação.

2.1. Jupyter Notebook e Anaconda (Python)

Recomendamos que os leitores usem o *software* livre Jupyter Notebook [28–30] em seus projetos Python de computação quântica, especialmente aqueles que não têm experiência em computação científica ou estão começando a aprender a linguagem Python. Recentemente, diversos trabalhos têm apontado a eficácia do Jupyter Notebook para o aprendizado de computação de alta performace [29, 30]. O Jupyter Notebook facilita a interação entre o usuário e o computador, permitindo a inclusão de textos na formatação LaTeX e a apresentação de resultados gráficos durante a execução dos programas, permitindo ao usuário acompanhar em tempo real cada etapa do código, auxiliando na compreensão dos códigos e seus resultados, sendo uma ferramenta robusta para o ensino de computação quântica. Além disso, uma das vantagens no uso do Jupyter é que o *IBM QE* [1] usa um ambiente Jupyter Notebook, que permite programar com Python na nuvem usando o pacote Qiskit em um computador quântico real a partir de um computador doméstico, e até mesmo emular um processador quântico a partir da unidade de processamento local do usuário mesmo sem acesso à internet.

O Jupyter Notebook, assim como o Python 3, podem ser facilmente encontrado para download gratuito na internet. Ambos fazem parte de uma das plataformas de ciência de dados mais populares da atualidade, o Anaconda [26, 30, 31].

O Anaconda[1] é uma ferramenta computacional que vem completamente pronta para uso, sendo um ambiente de desenvolvimento para várias linguagens populares, como Python, C, Java, R, Julia, entre outras [31]. O Anaconda vem com todas as bibliotecas necessárias para modelar sistemas físicos, como numpy, scipy e matplotlib, entre outros 150 pacotes pré-instalados e mais de 250 pacotes de código aberto que podem ser adicionados [26]. Dentre esses pacotes disponíveis para o repositório do Anaconda encontramos o *Qiskit* [22], elemento fundamental para esse trabalho. A seguir, mostramos uma breve introdução ao pacote *Qiskit*.

2.2. Quantum Information Software Development Kit – *Qiskit*

O Quantum Information Software Development Kit – *Qiskit*[2] [19–23] é uma estrutura computacional de código

[1] A última versão do Anaconda (4.8.3) pode ser baixada gratuitamente no site da plataforma [31], baseado no sistema operacional do computador do usuário. Nós recomendamos utilizar a instalação padrão. Depois de instalado, o usuário pode abrir o Anaconda Navigator no seu computador e atestar que a instalação foi concluída com êxito.

[2] A forma recomendada de instalar o *Qiskit* é utilizando o gerenciador de pacotes do Python, (`pip`), pré-instalado nas últimas versões

 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

aberto desenvolvida para funcionar em diferentes linguagens de programação como Python [21], Swift [32] e JavaScript [33], fornecendo as ferramentas necessárias para a criação de algoritmos quânticos, seguindo um modelo de circuito para computação quântica universal [5], e a sua execução em dispositivos quânticos reais usando o acesso remoto aos *hardwares* disponibilizados através do IBM QE. Além disso, o Qiskit permite emular um computador quântico em processador clássico local, como um laptop ou um desktop comum, permitindo a testagem de algoritmos quânticos simples em qualquer computador doméstico, sem a necessidade de acesso à internet ou criação de uma conta no IBM QE.

O IBM QE oferece a estudantes, pesquisadores e entusiastas da computação quântica acesso rápido e prático por meio de uma interface amigável, permitindo que os usuários executem seus projetos e experimentos [1, 3, 4]. Por outro lado, o *Qiskit* é uma ferramenta profissional para o desenvolvimento de programação quântica de alto nível [19–23], sendo tanto uma plataforma de desenvolvimento de *softwares* quânticos como uma linguagem de programação quântica [23]. Para isso, o *Qiskit* conta com cinco elementos essenciais:

> ***Terra:*** contém os elementos fundamentais que são usados para escrever os circuitos dos algoritmos quânticos;
>
> ***Aer:*** contém os recursos para as simulações quânticas por meio de computação de alto desempenho;
>
> ***AQUA:*** fornece bibliotecas de algoritmos pré programados para aplicação em Química, Finanças e Machine Learning.
>
> ***Ignis:*** contém ferramentas específicas para algoritmos de correção de erros, ruídos quânticos e verificação de *hardware* quântico.
>
> ***IBM Q Provider:*** não é necessariamente um elemento fundamental, mas fornece as ferramentas para acessar *IBM Q Experience* a fim de executar programas de usuários em um processador quântico real.

Neste artigo usamos o *Qiskit* na linguagem Python 3 para construir os circuitos quânticos e para as simulações dos algoritmos em computadores quânticos reais, usando apenas os elementos *Terra*, *Aer* e *IBM Q Provider*.

2.3. Importando os pacotes

Uma vez instalados o Anaconda (Python) e o *Qiskit* em seus computadores, os usuários estão prontos para aprender como escrever códigos para simular seus próprios algoritmos quânticos, construindo circuitos e executando-os em seus próprios computadores domésticos. Para iniciar o programa, é necessário adicionar estes recursos no ambiente Python no Jupyter Notebook, importando os seguintes módulos:

> `qiskit`: para projetar os circuitos quânticos e executar algoritmos quânticos [19–22];
>
> `numpy`: para construir um ambiente matemático com arrays e matrizes multidimensionais, usando sua grande coleção de funções matemáticas [27];
>
> `matplotlib`: para a criação de gráficos e visualizações de dados em geral [34];
>
> `qiskit.tools.monitor`: para utilizarmos a função `job_monitor` para monitorar em tempo real a execução dos nossos algoritmos [19–22];
>
> `qiskit.visualization`: para utilizar as funções `plot_histogram` para visualizar os resultados através das distribuições de probabilidade e `plot_bloch_multivector` para visualizar os estados na representação da esfera de Bloch [5].

Esses módulos básicos podem ser importados logo na primeira célula do notebook do Jupyter e executado com o comando `shift+enter` no teclado[3], sempre que um novo notebook for criado. Para isso, usamos os seguintes comandos:

Box 1: Importando os Pacotes

```
from qiskit import *
import numpy as np
import matplotlib.pyplot as plt
from qiskit.tools.monitor import job_monitor
from qiskit.visualization import␣
 ↪plot_histogram
from qiskit.visualization import␣
 ↪plot_bloch_multivector
%matplotlib inline
```

Vale destacar que o comando `%matplotlib inline` serve para definir o processo interno do `matplotlib`, permitindo que as saídas dos comandos de plotagem sejam exibidas de forma embutida na interface frontal, como o Jupyter Notebook, abaixo da célula em que o código é escrito [34].

Uma vez que os pacotes estão importados, temos todas as condições de começar a programar algoritmos quânticos em nosso computador pessoal e executá-los de forma remota nos computadores quânticos disponibilizados pela IBM [1–4].

3. Fundamentos

Nesta seção, fornecemos uma breve introdução aos conceitos fundamentais de informação quântica e computação quântica, usando os ambientes computacionais

do Python e Anaconda, utilizando o comando no terminal `> pip install qiskit`. Para uma instalação detalhada, recomendamos acessar a seção de instalação na página do github dos projetos [21].

[3] As células do Jupyter Notebook são sempre executadas através do comando `shift+enter` no teclado.

descritos na última seção. Descrevemos os conceitos de qubits, emaranhamento quântico, portas lógicas quânticas, circuitos e algoritmos. Esses tópicos foram amplamente estudados e discutidos na literatura nas últimas décadas [5, 7, 35]. Para os leitores que têm somente um conhecimento básico em mecânica quântica, recomendamos a leitura complementar das referências [2–4, 8, 14, 15]. Para leitores com conhecimento avançado em mecânica quântica, recomendamos as referências [5, 35] para uma descrição mais detalhada dos tópicos abordados nesta seção.

3.1. Bits quânticos (Qubits)

BInary digiT, ou *bit* é a menor unidade de informação em uma teoria da informação clássica, e a teoria da computação clássica é fundamentada neste conceito [5]. O *bit* clássico é um estado lógico que assume um dos dois valores possíveis $\{0, 1\}$. Outras representações úteis são $\{\texttt{sim}, \texttt{não}\}$, $\{\texttt{verdadeiro}, \texttt{falso}\}$ ou $\{\texttt{ligado}, \texttt{desligado}\}$. Em computadores clássicos, essas duas possibilidades podem ser implementadas usando componentes eletrônicos clássicos de dois estados, como dois níveis de tensão ou corrente distintos e estáveis em um circuito, duas posições de interruptores elétricos, dois níveis de intensidade de luz ou polarização e dois estados elétricos diferentes de um circuito flip-flop [36], por exemplo. Assim, os computadores são projetados com instruções para manipular e armazenar múltiplos *bits*, chamados bytes (conjunto de 8 bits).

Da mesma forma, a teoria da informação quântica e a computação quântica são construídas através de uma unidade de informação fundamental, análoga ao *bit* (clássico): os bits quânticos, ou simplesmente qubits [5]. No entanto, enquanto os bits clássicos podem assumir uma das duas possibilidades acima mencionadas, os qubits podem ser representados como uma combinação linear da base ortonormal de um sistema quântico de dois níveis, convencionalmente representada como $\{|0\rangle, |1\rangle\}$, chamada de *base computacional* [3, 5], onde em uma representação matricial:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{1}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{2}$$

Portanto, a principal vantagem dos qubits sobre os bits está no princípio de superposição [5, 8, 35, 37], o que possibilita combinações lineares entre os vetores que compõem a base computacional. Desta forma, a representação mais geral para um qubit é um vetor $|\psi\rangle$ definido no espaço de Hilbert [5], como:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{3}$$

onde α e β são amplitudes complexas que obedecem à condição de normalização $|\alpha|^2 + |\beta|^2 = 1$, com $|\alpha|^2$ correspondendo à probabilidade de obter o estado $|0\rangle$ e $|\beta|^2$ a probabilidade de obter o estado $|1\rangle$, através de uma medida no estado $|\psi\rangle$.

Após a importação dos pacotes apresentados no Box 1, temos todas as condições de criar o conjunto de regras ou operações que, aplicadas nos qubits, permitem solucionar algum problema preestabelecido, ou seja, os algoritmos quânticos.

O primeiro passo é definir as bases do circuito que implementará o algoritmo desejado, começando pelo conjunto de qubits que será utilizado no problema. Para isso, definimos uma variável[4] `q` usando a função `QuantumRegister` da seguinte forma:

Box 2: Registrando os qubits

```
q = QuantumRegister(N, 'q')

# Obs.: a variável N deve ser declarada␣
 ↪pelo usuário com o número de qubits que␣
 ↪serão utilizados.
```

No Box 2, `N` é um número inteiro e representa o número de qubits que será usado no circuito. Vale salientar que o valor dessa variável deve ser declarado pelo usuário dependendo do número de qubits que se deseja utilizar. Ao longo do texto, essa variável será substituida diretamente na linha de código. Por definição, os qubits são sempre registrados no estado $|0\rangle^{\otimes N}$, ou seja, cada um dos `N` qubits no estado $|0\rangle$.

Outro elemento importante na construção do circuito quântico é a definição do conjunto de bits clássicos onde registramos a informação oriunda das medidas realizadas nos qubits após a execução de algum algoritmo, por exemplo. Para isso, de maneira análoga aos qubits, definimos uma variável `b` usando a função `ClassicalRegister`:

Box 3: Registrando os bits clássicos

```
b = ClassicalRegister(N, 'b')

# Obs.: variável N deve ser declarada pelo␣
 ↪usuário com o número de bits que serão␣
 ↪utilizados.
```

Neste caso, `N` representa o número de bits que serão utilizados. Utilizamos a mesma variável aqui e no Box 2 assumindo que o número de qubits e de bits será igual, o que nem sempre precisa acontecer. Novamente, essa variável precisa ser declarada pelo usuário com o número de bits clássicos que se deseja utilizar.

Finalmente, podemos então declarar a variável `circuito` para construir o nosso circuito usando o conjunto de bits clássicos e quânticos definidos anteriormente através da função `QuantumCircuit`:

[4] O nome das váriáveis é de livre escolha do usuário.

 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

Box 4: Criando o circuito

```
circuito = QuantumCircuit(q, b)
```

Onde 'q' representa os qubits do sistema e 'b' os bits clássicos.

Nesse ponto, temos a base para o nosso circuito e temos todas as condições de definir os três componentes pincipais de todo algoritmo quântico:

> ***Inicialização:*** primeiro, precisamos iniciar nosso processo de computação em um estado bem definido;
>
> ***Portas Quânticas:*** em seguida, aplicamos a sequência de operações (portas) quânticas que permitem solucionar o problema preestabelecido;
>
> ***Medidas:*** finalizamos medindo os estados de cada qubit, registramos as medidas nos bits clássicos e, usando um computador clássico, interpretamos as medições através das distribuições de probabilidade correspondente a cada resultado das medidas.

A seguir, apresentamos cada etapa da construção de um algoritmo quântico.

3.2. Inicialização

Usando o *Qiskit*, podemos definir os coeficientes α e β e inicializar cada qubit do circuito no estado descrito na equação (3). Para isso, usamos os seguintes comandos:

Box 5: Inicializando um qubit em um determinado estado $|\psi\rangle$

```
psi = [alpha,beta]
circuito.initialize(psi,q[i])

#Obs.: o valor do índice i em q[i] deve ser␣
 ↪substituído pelo valor que corresponde ao␣
 ↪índice do qubit que se deseja inicializar.
```

Onde a variável `psi` é uma matriz que representa o estado descrito na equação (3), com as variáveis `alpha` e `beta` correspondendo aos coeficientes α e β, respectivamente, e `q[i]` o qubit `q` índice `i` que será inicializado no estado $|\psi\rangle$. Vale salientar que o valor do índice `i` em `q[i]` deve ser substituído pelo valor que corresponde ao índice do qubit que se deseja inicializar no estado $|\psi\rangle$ para que a linha de comando funcione.

3.2.1. Esfera de Bloch

Nesse ponto, vale destacar uma representação útil para o estado de um qubit, que pode ser obtida através do mapeamento das componentes α e β como funções de ângulos θ e ϕ. Dessa maneira, devido ao fato de α e β obedecerem à condição de normalização $|\alpha|^2 + |\beta|^2 = 1$, a equação (3) pode ser reescrita como

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle, \qquad (4)$$

Assim, o par $\{\theta, \phi\}$ define um ponto em uma esfera de raio unitário conhecida na literatura como *Esfera de Bloch* [5], que nos dá uma representação geométrica para o espaço de Hilbert de um qubit.

Nessa representação, o estado de um qubit corresponde a um ponto na superfície da esfera de Bloch e estados ortogonais são diametralmente opostos[5]. Através da importação do pacote `qiskit.visualization`, previamente instalado junto ao *Qiskit*, podemos usar a função `plot_bloch_multivector` para obtermos a visualização do qubit de interesse na esfera de Bloch.

Assim, escolhendo o par $\{\theta, \phi\}$ na equação (4), podemos obter a representação geométrica do qubit descrito por $|\psi\rangle$. Vamos analizar a inicialização dos qubits através de alguns exemplos. Primeiramente, importamos os pacotes necessários conforme descrito no Box 1; em seguida, registramos um qubit (`N` = 1) conforme descrito no Box 2; criamos um circuito conforme o Box 4, sem a necessidade de um bit clássico auxiliar, pois não serão feitas medidas nesse qubit. Finalmente, podemos inicializar nosso qubit a partir dos ângulos θ e ϕ. Usando o pacote `numpy` (chamado por `np`), definimos os coeficientes α e β a partir dos ângulos θ e ϕ e, conforme apresentado no Box 5, inicializamos o nosso estado. Todo esse processo é apresentado no Box 6, a seguir:

Box 6: Inicializar o qubit a partir dos ângulos θ e ϕ

```
theta = (float(input("Insira o ângulo␣
 ↪theta(°): ")))*np.pi/(180)
phi   = (float(input("Insira o ângulo␣
 ↪phi(°): ")))*np.pi/(180)
alpha = np.cos(theta/2)
beta = (np.exp(1j*phi))*np.sin(theta/2)
estado_inicial = [alpha,beta]
circuito.initialize(estado_inicial,q[0])
```

Finalmente, podemos usar o elemento `Aer` do *Qiskit* para simular o estado inicializado em nosso computador local, obter o vetor de estado e plotá-lo na representação da esfera de Bloch, usando o pacote `plot_bloch_multivector`.

Box 7: Plotando o qubit na Esfera de Bloch

```
processo = Aer.
 ↪get_backend('statevector_simulator')
vector_de_estado  = execute(circuito,␣
 ↪backend=processo).result().
 ↪get_statevector()
plot_bloch_multivector(vector_de_estado)
```

A Fig. 1 mostra a representação da Esfera Bloch para qubits inicializados em ângulos específicos.

Um outro caminho para a inicialização é a aplicação de operações que transformam o sistema de qubits inicialmente registrado no estado $|0\rangle^{\otimes N}$. Essas operações são conhecidas como portas quânticas.

[5] Isso explica o fato de usarmos $\frac{\theta}{2}$ na equação (4).

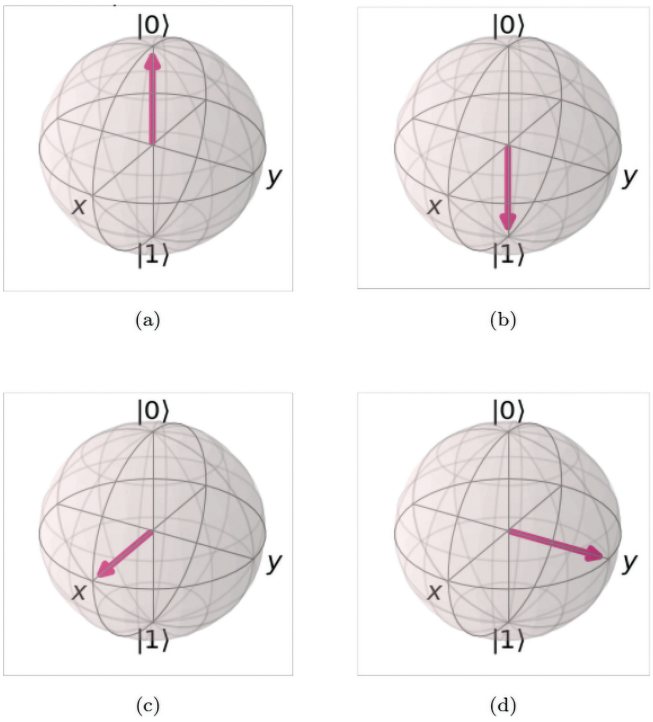


Figura 1: Representação da esfera de Bloch de um qubit. Escolhendo os ângulos θ e ϕ na equação (4), obtemos a representação da esfera de Bloch para os estados (a) $|\psi\rangle = |0\rangle$ ($\theta = 0^o$); (b) $|\psi\rangle = |1\rangle$ ($\theta = 180^o$); (c) $|\psi\rangle = |+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ $\{\theta = 90^o,\ \phi = 0^o\}$; e (d) $|\psi\rangle = |+i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$ $\{\theta = 90^o,\ \phi = 90^o\}$.

3.3. Portas quânticas

Uma vez definido os elementos básicos de informação quântica (os qubits), temos todas as condições de introduzir os conjuntos de operações que atuam sobre eles. Em computação clássica, essas operações são implementadas pelo que conhecemos como portas lógicas [8]. As Portas Lógicas Clássicas seguem uma Álgebra Booleana [38] e são implementadas a partir de circuitos eletrônicos [8], geralmente usando diodos ou transistores que atuam como interruptores eletrônicos, permitindo a implementação de alguma operação lógica através de uma determinada função booleana [38]. Assim, essas portas são aplicadas em circuitos lógicos para a implementação de processos computacionais, levando à solução de problemas através de algoritmos.

Na Computação Quântica, analogamente à computação clássica, as operações que atuam sobre os qubits são conhecidos como Portas Lógicas Quânticas, ou simplesmente Portas Quânticas. Ao contrário das portas lógicas clássicas, as portas quânticas são sempre reversíveis [8][6], uma vez que todas elas são descritas por operadores unitários U que obedecem à relação $U^\dagger U = \mathbb{1}$ [5], onde $\mathbb{1}$ é a matriz identidade.

[6] Em computação clássica, a única porta reversível é a porta `NOT` [5, 8].

Devido à grande quantidade de portas quânticas e às suas semelhanças de implementação no *Qiskit*, apresentamos a seguir as principais Portas Quânticas que utilizaremos ao longo desse trabalho, em sua forma matricial.

3.3.1. Portas de 1 qubit

Vamos começar com as portas quânticas de 1 qubit, a partir do que conhecemos como portas quânticas elementares, ou portas de Pauli [8], que correspondem às matrizes de Pauli [8, 37]:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \tag{5}$$

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \tag{6}$$

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{7}$$

Consideremos o estado descrito na equação (3). A atuação dessas portas nesse estado é:

$$X|\psi\rangle = \alpha|1\rangle + \beta|0\rangle, \tag{8}$$

$$Y|\psi\rangle = i\alpha|1\rangle - i\beta|0\rangle, \tag{9}$$

$$Z|\psi\rangle = \alpha|0\rangle - \beta|1\rangle. \tag{10}$$

Assim, pode-se perceber que as portas de Pauli correspondem a uma rotação na esfera de Bloch de π rad no eixo correspondente à direção representada pela porta.

Porta `NOT` (X)

No Qiskit, podemos verificar a atuação dessas portas em um qubit genérico. Por simplicidade, vamos verificar a atuação da porta X nos estados da base computacional $\{|0\rangle, |1\rangle\}$ como um exemplo:

Box 8: Aplicando a porta X

```
q = QuantumRegister(1, 'q')
circuito = QuantumCircuit(q)
estado_inicial = [1,0]
circuito.initialize(estado_inicial,q)
circuito.x(q)
```

A Fig. 2 mostra a atuação da porta X sobre os qubits da base computacional na representação da esfera de Bloch, implementado no *Qiskit* conforme foi apresentado no Box 7.

Como podemos ver, a aplicação da porta X corresponde a um inversor lógico, uma vez que ela nega o valor do bit de entrada. Isso pode ser interpretado como um análogo quântico para a porta `NOT` clássica [5, 8]. Por esse motivo, convencionou-se chamar a porta X como Porta `NOT` quântica [4].

Analogamente, para atuar as portas Y ou Z, basta somente trocar o `x` pela letra `y` ou `z` na última linha do Box 8, respectivamente.

 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

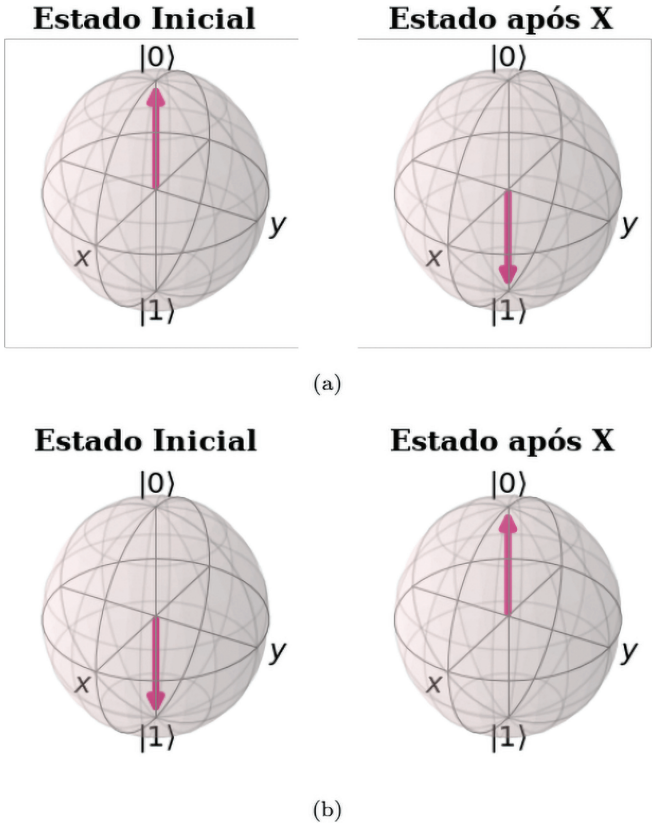


Figura 2: Representação na esfera de Bloch da atuação da porta X sobre os estados da Base Computacional (a) $|0\rangle$ e (b) $|1\rangle$. Como pode ser visto, a aplicação da porta X corresponde a um inversor lógico, implementando uma operação de negação lógica.

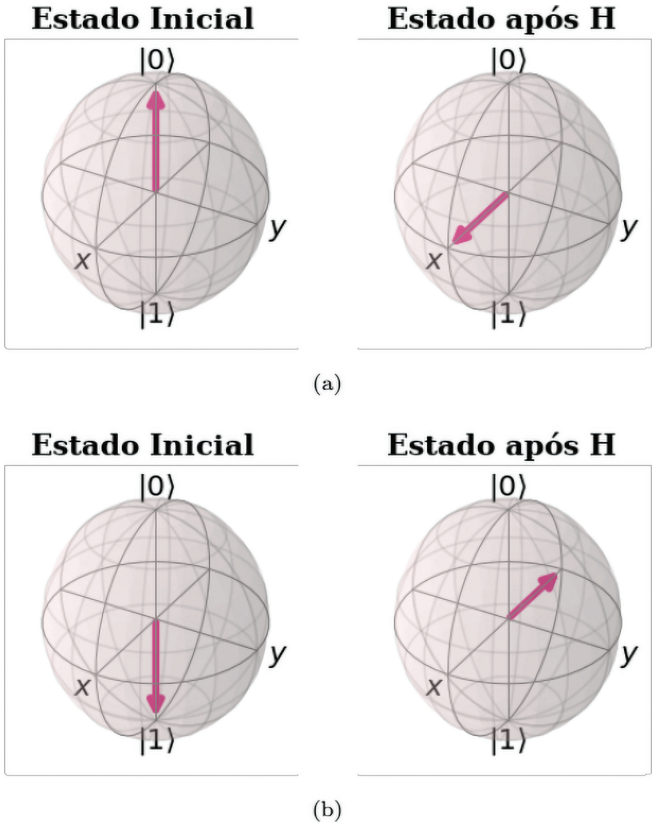


Figura 3: Representação na esfera de Bloch da atuação da porta H sobre os estados da Base Computacional (a) $|0\rangle$ e (b) $|1\rangle$.

Porta Hadamard (H)

Outra porta quântica extremamente importante que atua sobre 1 qubit é a porta Hadamard (H).

$$H = \frac{1}{\sqrt{2}}(X + Z) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{11}$$

Essa importância é devido ao fato da operação implementada pela porta Hadamard ser responsável pela criação de superposição. Considerando os estados da base computacional, a atuação da porta Hadamard resulta em:

$$H|0\rangle = |+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \tag{12}$$

$$H|1\rangle = |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \tag{13}$$

Analogamente à aplicação da porta `NOT`, para atuar a porta Hadamard basta somente trocar o `x` pela letra `h` na última linha do Box 8. A Fig. 3 mostra a atuação da porta H sobre os qubits da base computacional na representação da esfera de Bloch.

Um fato interessante é que a porta Hadamard pode ser combinada com a porta Z para formar a porta X, e combinada com a porta X para formar a porta Z, através das seguintes sequências:

$$X = HZH, \tag{14}$$

$$Z = HXH. \tag{15}$$

Essas combinações se mostram bastante úteis quando precisamos criar portas que não estão presentes na biblioteca do *Qiskit*, como veremos a seguir na seção de aplicações[7].

3.3.2. Desenhando circuitos quânticos

Assim como as portas lógicas clássicas são combinadas para formar circuitos lógicos para a implementação de algoritmos, as portas lógicas quânticas também podem ser combinadas para a construção de circuitos para a implementação de algoritmos quânticos. Nesse contexto, faz-se necessário introduzir a simbologia das portas lógicas usadas em computação quântica. Uma vez construído o

[7] Vale destacar que segundo a literatura [5], as portas Hadamard e T ($Z^{1/4}$) compõem o que chamamos de conjunto universal de portas de 1 qubit, pois através delas é possível implementar qualquer transformação unitária do estado de 1 qubit apresentado na equação (3). Entretanto, como a porta T não será utilizada nas aplicações desse artigo ela não será descrita aqui, para maiores detalhes indicamos as referências [5, 19].

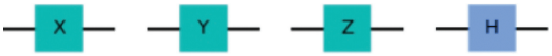


Figura 4: Representação das Portas quânticas de 1 qubit utilizadas ao longo desse trabalho: Pauli $\{X, Y, Z\}$ e Hadamard H.

nosso circuito, podemos desenhá-lo no Jupyter notebook e exportá-lo na forma de figura, com o comando[8]:

Box 9: Desenhando circuitos quânticos

```
circuito.draw(output = 'mpl')
```

A Fig. 4 mostra a representação dasportas de 1 qubit utilizadas nesse artigo e apresentadas nessa seção.

3.3.3. Portas de múltiplosqubits

A grande vantagem da computação quântica aparece quando trabalhamos com sistemas de múltiplos qubits [3, 12].

Como vimos no início dessa seção, um único bit tem dois estados possíveis $\{0, 1\}$. Diferentemente, o estado de 1-qubit tem duas amplitudes complexas $\{\alpha, \beta\}$, definido em um espaço vetorial complexo (espaço de Hilbert), equação (3), cujo módulo ao quadrado corresponde às amplitudes de probabilidade associada a cada um dos dois estados da base computacional para 1-qubit $\{|0\rangle, |1\rangle\}$. Portanto, da mesma forma que dois bits têm quatro estados possíveis $\{00, 01, 10, 11\}$, a base computacional para o espaço de Hilbert de dois qubits terá dimensão quatro e pode ser representada como $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. De maneira geral, um sistema de N-qubits é descrito por um estado quântico

$$|\Psi\rangle = c_1|0...00\rangle + c_2|0...01\rangle + c_3|00...10\rangle + \cdots + c_{2^N}|1...11\rangle, \tag{16}$$

onde os N-qubits são considerados como um único sistema composto com 2^N estados na sua base computacional, com

$$\sum_{i=1}^{N} |c_i|^2 = 1. \tag{17}$$

Trabalhar com vários qubits permite realizar operações em subconjuntos de qubits [5] e ainda assim fazer uso das propriedades quânticas de $|\Psi\rangle$ como a superposição, por exemplo. Apresentamos a seguir as principais portas quânticas que operam em múltiplos qubits usando o *Qiskit*.

Porta CNOT (CX)

Uma das portas mais importantes de múltiplos qubits é a conhecida porta CNOT (NOT Controlado ou CX).

A porta CNOT é uma porta de dois qubits, e sua atuação ocorre se, e somente se, o qubit, que chamamos de qubit de controle, for igual a $|1\rangle$. Nessa ocasião, atua-se a porta NOT no estado do outro qubit, que chamamos de qubit alvo. Assim, podemos representar matricialmente a porta CNOT como:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{18}$$

Considerando os estados da base computacional para dois qubits: $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, a atuação da porta CNOT resulta em:

$$CNOT|00\rangle = |00\rangle; \tag{19}$$
$$CNOT|01\rangle = |01\rangle; \tag{20}$$
$$CNOT|10\rangle = |11\rangle; \tag{21}$$
$$CNOT|11\rangle = |10\rangle. \tag{22}$$

A porta CNOT pode ser implementada em um circuito com o `qubit[0]`[9] como qubit de controle e `qubit[1]` como qubit alvo da seguinte maneira:

Box 10: Implementação da porta CNOT

```
q = QuantumRegister(2, 'q')
circuito = QuantumCircuit(q)
circuito.cx(q[0],q[1])
circuito.draw(output = 'mpl')
```

A Fig. 5 mostra a representação da porta CNOT em um circuito quântico. Usando o Box 7, podemos visualizar a atuação dessa porta na representação da Esfera de Bloch, conforme pode ser visualizado na Fig. 6:

Porta Toffoli (CCX)

Uma porta de múltiplos qubits bastante presente em diversos circuitos é a porta Toffoli (CCX) [5]. Sua atuação é executar a porta NOT no qubit alvo somente se dois qubits de controle estiverem no estado $|1\rangle$, sendo portanto uma porta de 3 qubits. Assim, podemos

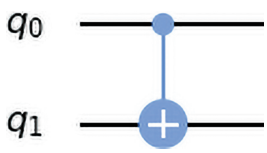


Figura 5: Representação da porta CNOT em um circuito quântico.

[8] Recomenda-se a instalação do pacote `pylatexenc` para a vizualização dos circuitos, utilizando o gerenciador de pacotes do Python (`pip`), digitando o comando no terminal `> pip install pylatexenc`.

[9] Na Linguagem Python 3, o primeiro índice de uma lista é sempre o 0.

 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

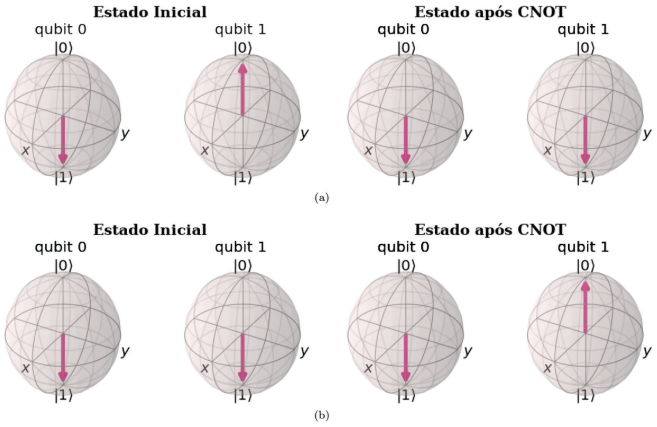


Figura 6: Representação na esfera de Bloch da atuação da porta $CNOT$ sobre os estados da Base Computacional (a) $|10\rangle$ e (b) $|11\rangle$.

representar matricialmente a porta Toffoli como:

$$CCX = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \quad (23)$$

A porta Toffoli pode ser implementada em um circuito com o `qubit[0]` e `qubit[1]` como qubits de controle e `qubit[2]` como qubit alvo da seguinte maneira:

Box 11: Implementação da porta Toffoli

```
q = QuantumRegister(3, 'q')
circuito = QuantumCircuit(q)
circuito.ccx(q[0],q[1],q[2])
circuito.draw(output = 'mpl')
```

Fig. 7 mostra a representação da porta Toffoli em um circuito quântico:

Novamente, podemos obter a atuação da porta Toffoli na representação da esfera de Bloch. Devido à semelhança operacional com outras portas controladas como a porta CNOT não apresentamos essa atuação neste texto.

Vale salientar que o conjunto de portas de Pauli, Hadamard, CNOT e Toffoli, apesar de não ser universal [5], é suficiente para implementarmos os algoritmos que iremos abordar na seção IV deste trabalho: (a) simulação de portas lógicas clássicas usando portas quânticas; (b) teleporte quântico; e (c) Algoritmo de Busca.

3.4. Medidas e distribuições de probabilidade

Como mencionamos anteriormente, definimos um conjunto de bits clássicos auxiliares de modo que as medições nos bits quânticos são salvas como resultados

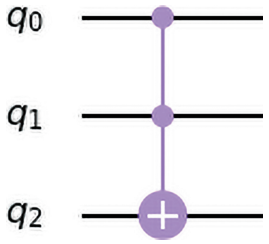


Figura 7: Representação da porta Toffoli em um circuito quântico.

clássicos $\{0, 1\}$. Logo, uma vez que vimos como inicializar os qubits e realizar operações sobre eles (portas), temos todas as condições para implementar os nossos algoritmos quânticos. Entretanto, ainda falta um passo fundamental para completarmos nosso processo de computação: a caracterização do estado final, através distribuição de probablidade correspondente.

Primeiramente, precisamos iniciar nosso processo computacional quântico com um estado bem definido, através das operações de inicialização discutidas anteriormente. Por simplicidade, vamos considerar o exemplo de um sistema de 2 qubits inicializados no estado $|00\rangle$ usando o comando `circuito.reset(q)`. Esse comando proporciona uma representação visual que cada qubit do sistema foi inicializado no estado $|0\rangle$ e é bastante útil para a organização dos circuitos. Em seguida, aplicamos uma sequência de portas quânticas que manipulam os dois qubits, conforme exigido pelo algoritmo. Por exemplo, consideremos um algoritmo que coloque todos os estados da base computacional de 2 qubits $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ em um estado de superposição, com mesma probabilidade de medida. Isso pode ser obtido aplicando a porta Hadamard em cada qubit. O Box abaixo traz a sequência de inicialização, aplicação de portas e a realização das medidas do nosso exemplo:

Box 12: Exemplo – Criação de um estado de superposição para 2 qubits

```
# Preparativos:
q = QuantumRegister(2, 'q') #Registrando os␣
 ↪qubits
b = ClassicalRegister(2, 'b') #Registrando␣
 ↪os Bits
circuito = QuantumCircuit(q,b) #Criando o␣
 ↪Circuito

# Inicialização dos Estados:
circuito.reset(q)

# Aplicação das Portas:
circuito.h(q)

# Realização das Medidas
circuito.measure(q,b)
circuito.draw(output = 'mpl')
```

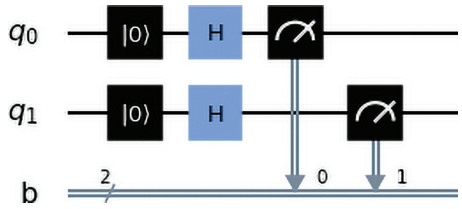


Figura 8: Circuito quântico gerador de superposição equiprovável para os estados da base computacional de 2 qubits.

A Fig. 8 mostra o circuito quântico gerador de superposição equiprovável para os estados da base computacional de 2 qubits apresentado no Box 12.

Finalmente, após medir os qubits, podemos utilizar um computador clássico para interpretar as medições de cada qubit como resultados clássicos (0 e 1) e armazená-los nos bits clássicos definidos para esse circuito. O *Qiskit* contém o simulador `QASM`, um dos principais componentes do elemento `Aer` para as simulações quânticas por meio de computação de alto desempenho. Este simulador emula a execução de circuitos quânticos em um processador local e retorna as contagens de cada medida no estado final para um dado conjunto de repetições (ou `shots`) do circuito definido pelo usuário. Podemos usar esse recurso para simular nossos circuitos quânticos usando nosso computador pessoal para emular numericamente[10] um processador quântico ideal, sem nenhuma influência de pertubações externas devido ao acoplamento inevitável entre os sistemas quânticos de processamento da informação e o ambiente externo [1, 3], conhecido como decoerência [5, 8]. Além disso, ruídos e outros processos podem levar a imprecisão no controle fino dos qubits, ocasionando erros sistemáticos na implementação das portas lógicas [1][11].

Há pequenas variações nas probabilidades de cada estado associadas ao processo estatístico que o *Qiskit* utiliza para emular um processador quântico. Entretanto, há maneiras de otimizar esse processo para que a emulação e a realização experimental se aproximem cada vez mais do resultado predito teoricamente. Uma maneira bastante simples e eficiente é aumentar o número de repetições de cada circuito (`shots`) [4], conforme apresentado a seguir no Box 13, onde apresentamos o código que simula o circuito mostrado no Box 12 em um processador ideal, em que o número de shots (variável `s`) deve ser definido pelo usuário.

[10] A descrição dos métodos numéricos utilizados nessa emulação foge dos objetivos principais deste trabalho. Para maiores detalhes sugerimos as referências [19–22].

[11] A descrição em detalhes dos processos que levam a essas imprecisões foge do escopo geral desse trabalho e serão abordadas em publicações futuras. Na aba *User Guide* do IBM QE [1], é possível encontrar detalhes acerca desses processos que ocasionam imprecisões no controle dos qubits e na implementação das portas lógicas quânticas. Para mais detalhes sobre a arquitetura do computador quântico da IBM, indicamos as referências [3, 4, 8].

Box 13: Simulando um circuito em um processador quântico ideal emulado numericamente

```
simular = Aer.get_backend('qasm_simulator')
resultadolocal = execute(circuito, backend
 ↪= simular, shots = s).result()
```

Caso o comando `shots = s` não seja inserido, o *Qiskit* utilizará por padrão 1024 shots. Assim, uma vez definido o número de `shots` obtemos as contagens dos estados e a distribuição de probabilidade para cada estado. Podemos usar a função `plot_histogram`, presente no módulo `qiskit.visualization`, importado no início do notebook (Box 1), para visualizar o resultado da contagem dos estados, da seguinte maneira:

Box 14: Plotando a distribuição de probabilidade do estado final de um circuito

```
titulo = 'Probabilidades'
plot_histogram(resultadolocal.
 ↪get_counts(circuito), title=titulo)
```

Vale detacar que mesmo que nenhuma decoerência atue e nenhum erro adicional afete a estatística do sistema, se o número de `shots` não for grande o suficiente, haverá uma discrepância estatística entre o resultado obtido e o esperado teoricamente. Definindo um número de shots grande o suficiente, garantimos a estatística esperada para o processo ideal de medida dos qubits. A Fig. 9(a) apresenta a distribuição de probabilidade do circuito da Fig. 8 simulado em um processador ideal para diferentes shots. Como pode ser visto, quanto menor o número de repetições do circuito, mais discrepante será a distribuição de probabilidade entre os estados da base computacional. À medida que os shots aumentam, nos aproximamos da distribuição de probabilidade equiprovável do estado previsto teoricamente:

$$|\Psi\rangle = \frac{1}{2}\left[|00\rangle + |01\rangle + |10\rangle + |11\rangle\right]. \qquad (24)$$

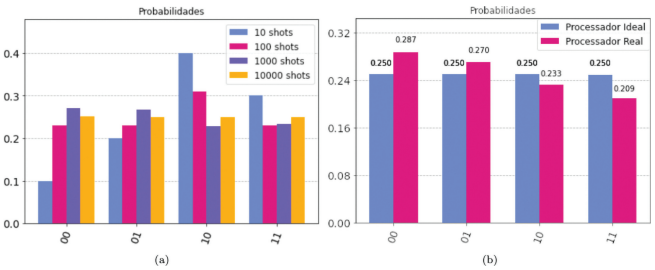


Figura 9: (a) Comparação entre as probabilidades obtidas para o circuito gerador de superposição equiprovável para os estados da base computacional de 2 qubits simulado em um processador ideal, com diferentes números de repetições (`shots`). (b) Comparação entre as distribuições de probabilidade para o circuito apresentado na Fig. 8 emulado numericamente (processador ideal) e em um processador real.

 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

3.4.1. Acessando o *IBM QE* usando o *Qiskit*

Além de podermos simular nosso circuito quântico em um processador ideal emulado numericamente em um computador clássico doméstico, podemos também executar nossos projetos em processadores quânticos reais usando o *IBM Q Experience* por meio do elemento *IBM Q Provider*, que vem com o *Qiskit*. Para isso, é preciso criar uma conta gratuita no *IBM Q Experience* [1]. Acessando as configurações de *Minha Conta*, o usuário encontra seu token de API, que é necessario para acessar dispositivos IBM Q de seu computador doméstico usando o *Qiskit*. No Notebook Jupyter, podemos usar os seguintes comandos:

Box 15: Salvando a conta IBM QE no computador

```
IBMQ.save_account('Users_Token')
```

Este comando irá salvar o token de API do usuário em seu computador, permitindo acessar dispositivos quânticos disponibilizados pela IBM. Esse passo só precisa ser realizado uma vez.

Para carregar a conta, usamos o comando:

Box 16: Carregando a conta IBM QE no notebook do Jupyter

```
IBMQ.load_account()
```

Após o comando apresentado no Box 16 ser executado, a conta será carregada com êxito e poderemos ver a conta de acesso através da saída:

Box 17: Saída padrão atestando o acesso aos *hardwares* disponibilizados pela IBM

```
<AccountProvider for IBMQ(hub='ibm-q',␣
 ↪group='open', project='main')>
```

Ao concluir esta etapa podemos executar nossos projetos, não apenas em um processador emulado em um computador doméstico, mas também enviar circuitos quânticos para dispositivos da IBM e obter os resultados em *hardware* quântico real. Vejamos o exemplo do circuito gerador de superposição equiprovável para os estados da base computacional de 2 qubits, apresentado na Fig. 8.

O Box 18 lista os comandos que selecionam o provedor e os sistemas quânticos e simuladores aos quais temos acesso pelo IBM QE para realizarmos a computação em processadores reais[12].

Box 18: Selecionando o provedor e executando o trabalho

```
provedor = IBMQ.get_provider('ibm-q')
comput = provedor.
 ↪get_backend('ibmq_valencia')
trabalho = execute(circuito,␣
 ↪backend=comput, shots = 8000)
job_monitor(trabalho)
```

Usando o comando `job_monitor()`, podemos monitorar o nosso circuito na fila de execução do processador em tempo real. Após finalizar a execução, recebemos a mensagem (em inglês):

```
Job Status: job has successfully run
```

Indicando que o trabalho foi executado com sucesso. Assim, obtemos a contagem dos estados e podemos plotar as distribuições de probabilidade como apresentado no Box 19.

Vale destacar que, devido a limitações de *hardware*, o número de repetições permitidas (`shots`) no processador quântico real é de 8192. Assim, para fazer um comparativo entre o processador quântico real e ideal será usado o mesmo número de `shots` (8000) em ambas as implementações.

Usando os comandos apresentados no Box 19 (a seguir), fazemos o comparativo das distribuições de probabilidade obtidas para o circuito da Fig. 8, simulado numericamente em um computador doméstico e executado em um processador quântico real.

Box 19: Plotando as distribuições de probabilidade de um circuito emulado numericamente e em um processador real.

```
resultadoIBM = trabalho.result()
legenda = ['Processador Ideal',␣
 ↪'Processador Real']
titulo = 'Probabilidades'
plot_histogram([resultadolocal.
 ↪get_counts(circuito), resultadoIBM.
 ↪get_counts(circuito)],␣
 ↪legend=legenda,title=titulo)
```

Na Fig. 9(b) pode ser observada, em azul, a distribuição de probabilidade correspondente àquela obtida pela execução dos Boxes 13 e 14 para o estado final obtido para o circuito apresentado no Box 12 (Fig. 8) com o número de `shots s=8000`. Como pode ser visto, a distribuição de probabilidade obtida numericamente corresponde exatamente ao estado quântico, equação (24).

Fica clara a diferença entre um processador quântico ideal (emulado em um computador doméstico) e um processador quântico real, com o mesmo número de shots. As referências [3, 4] trazem uma análise detalhada do processador quântico de 5 qubits da IBM QE, cuja arquitetura é a mesma do utilizado nesse trabalho

[12] Há 8 sistemas disponíveis para a execução. O usuário sempre pode conferir quais *hardwares* estão disponíveis ou com a menor quantidade de trabalhos em execução diretamente no painel de controle do IBM QE [1]. Todos os algoritmos apresentados nesse trabalho foram executados no mesmo processador quântico de 5 qubits `ibmq_valencia`, conforme mostramos no Box 18. Diante da possibilidade desse sistema estar indisponível no momento da execução dos comandos pelos leitores, nós recomendamos o uso de outro processador quântico de 5 qubits como, por exemplo, o `ibmq_belem`.

(`ibmq_valencia`). Uma descrição da análise de dados e o erro padrão associado a essa arquitetura de processador quântico pode ser encontrada nas referências [3, 4].

3.5. Emaranhamento quântico

Uma vez que conhecemos como inicializar nossos qubits e vimos as principais operações que atuam sobre eles, podemos introduzir uma das principais propriedades da mecânica quântica e um recurso fundamental para o processamento da informação quântica, o *Emaranhamento*.

O emaranhamento quântico é um dos fenômenos mais interessantes da mecânica quântica que emerge da interação entre múltiplos qubits. Nos últimos anos, o emaranhamento quântico tem recebido atenção considerável como um recurso notável para o processamento de informação quântica [5, 39]. Einstein, Podolsky, e Rosen (EPR) introduziram a ideia de que estados quânticos de um sistema composto podem apresentar correlações não locais entre seus componentes [40]. Schrödinger analisou algumas consequências físicas da mecânica quântica, observando que alguns estados quânticos bipartidos (estados EPR [39]) não admitiam atribuir estados individuais de subsistemas [41]. Portanto, o emaranhamento implica a existência de estados quânticos globais de sistemas compostos que não podem ser escritos como um produto dos estados quânticos de subsistemas individuais [5, 39].

Consideremos um estado quântico de um sistema composto perfeitamente descrito pela função de onda

$$|\Psi\rangle \neq |\phi_1\rangle \otimes |\phi_2\rangle \otimes \cdots \otimes |\phi_n\rangle, \tag{25}$$

não podemos especificar qualquer estado quântico puro $|\phi_i\rangle$ $(i = 1, \ldots, n)$ dos subsistemas separadamente; isto é, o conhecimento de um todo não implica conhecimento das partes.

Portanto, não sabemos nada sobre os subsistemas, embora tenhamos conhecimento do sistema como um todo, uma vez que conhecemos $|\Psi\rangle$. Isso contrasta com a situação clássica, em que sempre podemos considerar os estados individuais dos subsistemas. Esta é uma pista de que estados emaranhados são estados correlacionados especiais, cuja natureza física não pode ser simulada ou representada a partir de correlações clássicas.

Em um circuito quântico, podemos emaranhar dois qubits através da combinação das portas Hadamard e CNOT, apresentadas anteriormente. Dependendo dos valores de inicialização dos qubits `q[0]` e `q[1]`, obtemos um dos quatro estados maximamente emaranhados para 2 qubits ou *Estados de Bell* [8], conforme apresentado na Tabela 1.

Vejamos um exemplo para o sistema inicializado no estado `q[0]` $= |0\rangle$ e `q[1]` $= |0\rangle$. Ao final do processo executado em um *hardware* quântico real, realizamos medidas para um conjunto de repetições e obtemos a distribuição de probabilidade correspondente ao estado final.

Tabela 1: Estados maximamente emaranhados correspondente a cada inicialização dos qubits `q[0]` e `q[1]`.

`q[0]`	`q[1]`	$\|\Psi\rangle$
$\|0\rangle$	$\|0\rangle$	$\frac{1}{\sqrt{2}}[\|00\rangle + \|11\rangle]$
$\|0\rangle$	$\|1\rangle$	$\frac{1}{\sqrt{2}}[\|01\rangle + \|10\rangle]$
$\|1\rangle$	$\|0\rangle$	$\frac{1}{\sqrt{2}}[\|00\rangle - \|11\rangle]$
$\|1\rangle$	$\|1\rangle$	$\frac{1}{\sqrt{2}}[\|01\rangle - \|10\rangle]$

Box 20: Circuito quântico gerador de estados quânticos emaranhado para 2 qubits

```
# Preparativos:

q = QuantumRegister(2, 'q') #Registrando os␣
 ↪qubits
b = ClassicalRegister(2, 'b') #Registrando␣
 ↪os Bits
circuito = QuantumCircuit(q,b) #Criando o␣
 ↪Circuito

# Inicialização dos Estados:
circuito.reset(q[0])
circuito.reset(q[1])

# Aplicação das Portas:
circuito.h(q[0])
circuito.cx(q[0],q[1])

# Realização das Medidas
circuito.measure(q,b)
circuito.draw(output = 'mpl')

provedor = IBMQ.get_provider('ibm-q')
comput = provedor.
 ↪get_backend('ibmq_valencia')
trabalho = execute(circuito,␣
 ↪backend=comput, shots = 8000)
job_monitor(trabalho)

resultadoIBM = trabalho.result()
titulo = 'Probabilidades'
plot_histogram(resultadoIBM.
 ↪get_counts(circuito), title=titulo)
```

A Fig. 10 mostra a representação do circuito representado no Box 20 e a distribuição de probabilidade correspondente ao resultado desse circuito executado em um hardware quântico real, utilizando o IBM QE:

4. Aplicações

Uma vez que sabemos inicializar os qubits, emaranhá-los, aplicar as portas quânticas e obter os resultados através de medidas, temos todas as condições de construir algoritmos para a solução de problemas

 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

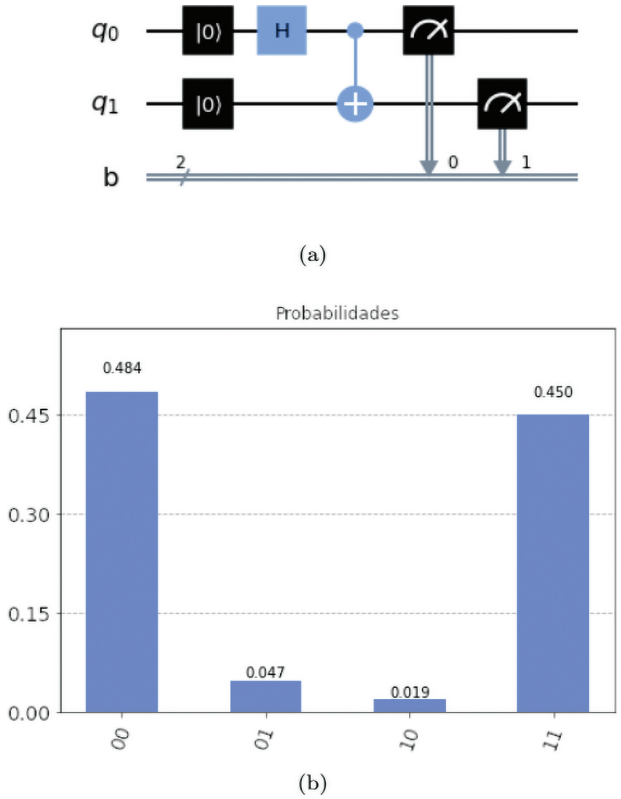


Figura 10: (a) Representação do circuito gerador de estados quânticos emaranhado para 2 qubits. (b) Distribuição de probabilidade correspondente ao resultado desse circuito executado em um *hardware* quântico real usando o IBM QE, com os qubits inicializados no estado `q[0]` $= |0\rangle$ e `q[1]` $= |0\rangle$.

quânticos simples. Nessa seção, traremos algumas aplicações de algorimos quânticos executados em computadores quânticos reais. Apresentaremos a construção de portas lógicas clássicas a partir de portas quânticas, o famoso algoritmo de teleporte quântico [3–5, 8] e o algoritmo de busca de Grover [5, 14]. Os códigos são apresentados ao longo do texto, de modo que os leitores possam reproduzi-los em seus computadores, podendo inclusive construir os seus próprios projetos a partir deles.

4.1. Simulando portas lógicas clássicas usando portas quânticas

Uma porta lógica clássica pode ser definida como um modelo ou dispositivo físico que implementa uma determinada função booleana [38], realizando assim aquilo que é conhecido como operação lógica. Essa operação é realizada em uma (porta `NOT`, por exemplo) ou mais entradas binárias (bits), produzindo somente uma única saída $\{0, 1\}$.

Existe um conjunto de portas lógicas clássicas a partir das quais podemos construir qualquer operação computacional em um computador clássico [8]. Essas são as portas `AND`, `OR` e `NOT`, também conhecidas como conjunto de portas universais da Álgebra Booleana.

Tabela 2: Tabela verdade para a porta lógica clássica AND.

Entrada		Saída
q[0]	q[1]	q[2]
0	0	0
0	1	0
1	0	0
1	1	1

Como vimos anteriormente, a porta quântica X corresponde ao análogo quântico da porta `NOT` clássica. A seguir, apresentamos como podemos construir as portas `AND` e `OR`, e os seus resultados executados em um computador quântico real.

4.1.1. Porta AND

A porta clássica `AND` implementa o que chamamos de conjunção lógica [38]. A Tabela 2 traz o que chamamos de *tabela verdade* para essa operação lógica, a partir da qual é possível definir o resultado lógico dessa operação.

Como pode ser visto, a partir de dois bits de entrada, a saída `1` é obtida somente se as duas entradas também forem 1. Assim, podemos dizer que a porta `AND` encontra o valor mínimo entre dois bits.

Quanticamente, a porta `AND` pode ser implementada a partir da porta Toffoli, conforme apresentamos no Box 11. Como todas as portas clássicas, exceto a porta `NOT`, a porta `AND` não é reversível. Entretanto, como toda porta quântica, a porta Toffoli é reversível, o que significa que implementar a porta `AND` em computadores quânticos permite a construção de circuitos reversíveis. O Box 21 apresenta a construção do circuito quântico para a porta AND. Como as portas clássicas têm somente uma saída, a medida é realizada apenas no qubit alvo da porta Toffoli.

Box 21: Criando o circuito para a porta `AND`

```
# Registrando os Qubits e os Bits

q = QuantumRegister(3, 'q')
b = ClassicalRegister(1, 'b')

# Circuito 1 - Entrada: A=0 B=0

circuito1 = QuantumCircuit(q, b)
circuito1.ccx(q[0],q[1],q[2])
circuito1.measure(q[2], b)
circuito1.draw(output = 'mpl')

# Circuito 2 - Entrada: A=0 B=1

circuito2 = QuantumCircuit(q, b)
circuito2.x(q[1])
circuito2.ccx(q[0],q[1],q[2])
circuito2.measure(q[2], b)
```

```
# Circuito 3 - Entrada: A=1 B=0

circuito3 = QuantumCircuit(q, b)
circuito3.x(q[0])
circuito3.ccx(q[0],q[1],q[2])
circuito3.measure(q[2], b)


# Circuito 4 - Entrada: A=1 B=1

circuito4 = QuantumCircuit(q, b)
circuito4.x(q[0])
circuito4.x(q[1])
circuito4.ccx(q[0],q[1],q[2])
circuito4.measure(q[2], b)

# Executando os circuitos e plotando os␣
 ↪resultados

provedor = IBMQ.get_provider('ibm-q')
comput = provedor.
 ↪get_backend('ibmq_valencia')
trabalho =␣
 ↪execute([circuito1,circuito2,circuito3,
circuito4],backend=comput, shots= 8000)
job_monitor(trabalho)

resultadoIBM = trabalho.result()
legenda = ['Entrada: A=0 B=0; Saída:␣
 ↪0','Entrada: A=0 B=1; Saída: 0','Entrada:␣
 ↪A=1 B=0; Saída: 0','Entrada: A=1 B=1;␣
 ↪Saída: 1']
titulo = 'Probabilidades'
plot_histogram([resultadoIBM.
 ↪get_counts(circuito1),resultadoIBM.
 ↪get_counts(circuito2),resultadoIBM.
 ↪get_counts(circuito3),resultadoIBM.
 ↪get_counts(circuito4)],legend=legenda,
title=titulo)
```

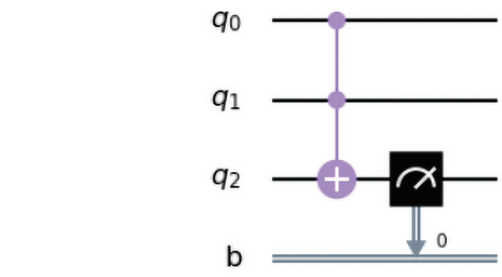
A Fig. 11 apresenta o circuito quântico que implementa a porta AND e a distribuição de probabilidade correspondente à aplicação do circuito em um processador quântico real.[13] Apresentamos os resultados correspondentes à tabela verdade da porta `AND` clássica (Tabela 2).

4.1.2. Porta OR

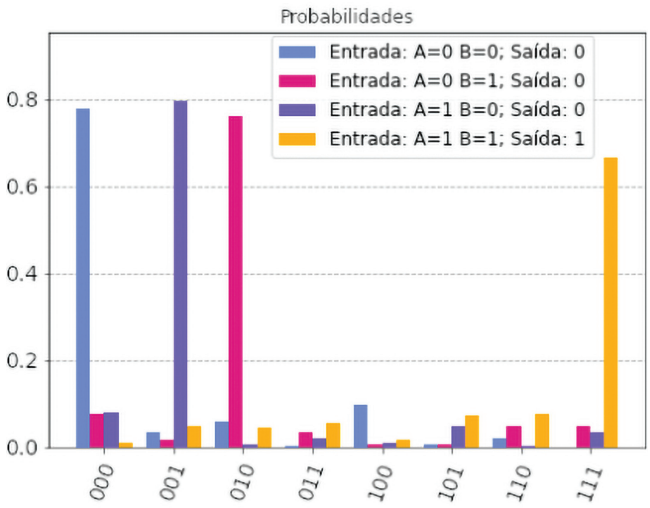
A porta `OR` é uma porta clássica universal que implementa o que chamamos em álgebra booleana de disjunção lógica [38]. A tabela verdade para a aplicação da porta clássica `OR` é apresentada na Tabela 3.

Como pode ser visto, uma saída `1` é obtida se pelo menos uma das entradas for `1`. Assim, dizemos que a porta `OR` encontra o máximo entre duas entradas binárias.

O análogo quântico para a `OR` pode ser construído através da combinação das portas Toffoli e CNOT. O Box 22 traz o código de construção do circuito para a implementação da porta `OR`.



(a)



(b)

Figura 11: (a) Representação do circuito para a aplicação da porta AND. (b) Distrituição de probabilidade para o circuito de aplicação da porta AND executado em um computador quântico real. Apresentamos os resultados da tabela verdade correspondente à porta clássica `AND` (Tabela 2).

Tabela 3: Tabela verdade para a porta lógica clássica OR.

Entrada		Saída
`q[0]`	`q[1]`	`q[2]`
0	0	0
0	1	1
1	0	1
1	1	1

[13] Nesse ponto, vale destacar que a apresentação dos bits no eixo x em todas as distribuições de probabilidade desse artigo segue o padrão do *Qiskit*: `bit[0]bit[1]bit[2]` apresentados de cima para baixo.

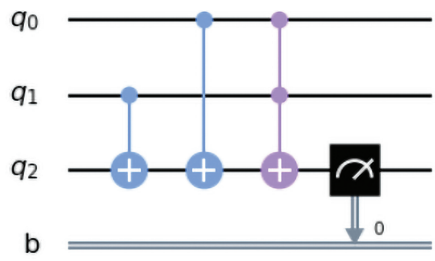
 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

Box 22: Criando o circuito para a porta OR

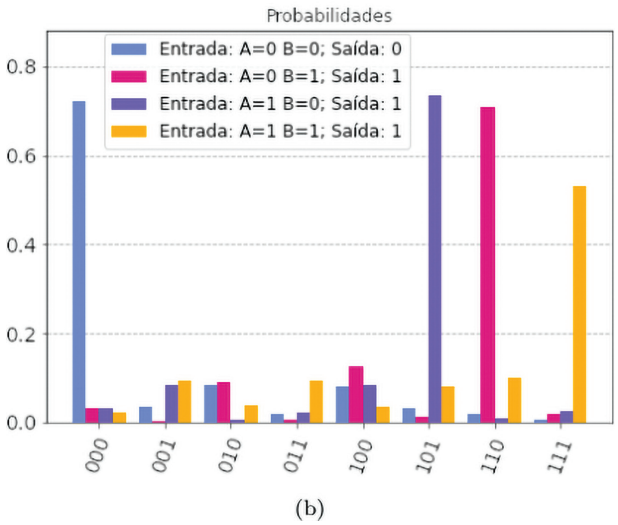
```
# Registrando os Qubits e os Bits

q = QuantumRegister(3, 'q')
b = ClassicalRegister(1, 'b')

# Circuito 1 - Entrada: A=0 B=0

circuito1 = QuantumCircuit(q, b)
circuito1.cx(q[1],q[2])
circuito1.cx(q[0],q[2])
circuito1.ccx(q[0],q[1],q[2])
circuito1.measure(q[2], b)
circuito1.draw(output = 'mpl')

# Circuito 2 - Entrada: A=0 B=1

circuito2 = QuantumCircuit(q, b)
circuito2.x(q[1])
circuito2.cx(q[1],q[2])
circuito2.cx(q[0],q[2])
circuito2.ccx(q[0],q[1],q[2])
circuito2.measure(q[2], b)


# Circuito 3 - Entrada: A=1 B=0

circuito3 = QuantumCircuit(q, b)
circuito3.x(q[0])
circuito3.cx(q[1],q[2])
circuito3.cx(q[0],q[2])
circuito3.ccx(q[0],q[1],q[2])
circuito3.measure(q[2], b)


# Circuito 4 - Entrada: A=1 B=1

circuito4 = QuantumCircuit(q, b)
circuito4.x(q[0])
circuito4.x(q[1])
circuito4.cx(q[1],q[2])
circuito4.cx(q[0],q[2])
circuito4.ccx(q[0],q[1],q[2])
circuito4.measure(q[2], b)

# Executando os circuitos e plotando os␣
 ↪resultados


IBMQ.load_account()
provedor = IBMQ.get_provider('ibm-q')
comput = provedor.
 ↪get_backend('ibmq_valencia')
trabalho =␣
 ↪execute([circuito1,circuito2,circuito3,
circuito4],backend=comput, shots= 8000)
job_monitor(trabalho)

resultadoIBM = trabalho.result()
legenda = ['Entrada: A=0 B=0; Saída:␣
 ↪0','Entrada: A=0 B=1; Saída: 1','Entrada:␣
 ↪A=1 B=0; Saída: 1','Entrada: A=1 B=1;␣
 ↪Saída: 1']
titulo = 'Probabilidades'
plot_histogram([resultadoIBM.
 ↪get_counts(circuito1),resultadoIBM.
 ↪get_counts(circuito2),resultadoIBM.
 ↪get_counts(circuito3),resultadoIBM.
 ↪get_counts(circuito4)],legend=legenda,
title=titulo)
```

O circuito para essa operação é apresentado na Figura 12(a).

Assim, através das portas NOT (Fig. 2), AND (Fig. 11) e OR (Fig. 12), é possivel implementar qualquer porta lógica clássica em um computador quântico, com a vantagem de que as portas AND e OR quânticas são reversíveis, ao contrário de seus análogos clássicos [8].



(a)



(b)

Figura 12: (a) Representação do circuito para a aplicação da porta OR. (b) Distribuição de probabilidade para o circuito de aplicação da porta OR executado em um computador quântico real. Apresentamos os resultados da tabela verdade correspondente à porta clássica OR (Tabela 3).

4.2. Teleporte quântico

Outra aplicação muito interessante e bastante discutida na literatura da computação quântica é o *Teleporte Quântico* [3–5, 8]. O teleporte quântico consiste na transmissão de um estado quântico desconhecido (totalmente arbitrário) entre duas partes, convencionalmente conhecidas como Alice e Bob, separadas espacialmente [3–5, 8] através do seguinte protocolo: (*i*) Alice pussui somente uma única cópia de um qubit (totalmente arbitrário), e quer enviar esse estado quântico para Bob; (*ii*) Para isso, Alice deverá preparar o qubit cuja informação será enviada, e possuir um segundo qubit (auxiliar) que será maximamente emaranhado a um terceiro qubit (auxiliar) pertencente a Bob, que receberá a informação do estado enviado; (*iii*) Assim, Alice executa medidas em seus dois qubits e informa, através de um canal clássico, o resultado de suas medidas para Bob; (*iv*) Dessa maneira, com essa informação (clássica) Bob realiza adequadamente um conjunto de operações (quânticas) em seu qubit para recuperar o estado enviado por Alice e o protocolo funcionar perfeitamente.

Vamos considerar em nosso exemplo que Alice pretende teleportar o estado:

$$|\psi\rangle = \sqrt{\frac{1}{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle, \tag{26}$$

para isso iremos inicializar o qubit de Alice nesse estado. O Box a seguir apresenta o registro dos qubits de Alice e Bob, a inicialização do estado que será teleportado e o registro do bit clássico no qual Bob armazena o resultado da medida feita no seu estado recebido.

Box 23: Registrando os qubits e inicializando o estado que será teleportado

```
Alice = QuantumRegister(2, 'alice')
Bob = QuantumRegister(1, 'bob')
b = ClassicalRegister(1, 'c_bob')
teleporte = QuantumCircuit(Alice,Bob,b)
estado_inicial = [np.sqrt(1/3),np.sqrt(2/3)]
teleporte.
 ↪initialize(estado_inicial,Alice[0])
teleporte.barrier()
```

O próximo passo é emaranhar o qubit auxiliar de Alice com o qubit de Bob em um dos Estados de Bell apresentados na Tabela 1, usando o circuito quântico gerador de estados quânticos emaranhado para 2 qubits (Box 20). O Box 24 apresenta o circuito gerador de emaranhamento entre o qubit de Alice e Bob:

Box 24: Emaranhando o qubit auxiliar de Alice com o qubit de Bob

```
teleporte.h(Bob[0])
teleporte.cx(Bob[0],Alice[1])
teleporte.barrier()
```

Em seguida, Alice inicia o processo de envio do estado preparado, no Box 25.

Box 25: Alice prepara o envio do estado que será teleportado

```
teleporte.cx(Alice[0],Alice[1])
teleporte.h(Alice[0])
teleporte.barrier()
```

No protocolo original [5, 8], o próximo passo seria Alice realizar medidas em seus qubits e, a depender dos resultados, entrar em contato com Bob através de um canal clássico para informar as correções que Bob deve aplicar em seu estado para que o teleporte seja executado e ele consiga resgatar o estado enviado por Alice. Esse passo pode ser executado através de uma operação condicionada ao resultado das medidas de Alice. Entretanto, o IBM QE não permite a implementação desse tipo de porta condicionada a um canal clássico. Nesse caso, podemos substituí-la pela porta CNOT e Z-Controlada (construída a partir da combinação das portas Hadamard e CNOT a partir da equação (15)) [4, 8]. Assim, conseguimos modificar o circuito original sem mudar seu objetivo. O Box 26 apresenta a construção da correção do protocolo de teleporte.

Box 26: Correção do algoritmo de teleporte para o resgate do estado enviado por Alice

```
teleporte.h(Bob[0])
teleporte.cx(Alice[0], Bob[0])
teleporte.h(Bob[0])
teleporte.cx(Alice[1], Bob[0])
teleporte.measure(Bob, b)
teleporte.draw(output = 'mpl')

provedor = IBMQ.get_provider('ibm-q')
comput = provedor.
 ↪get_backend('ibmq_valencia')
trabalho = execute(teleporte,␣
 ↪backend=comput, shots = 8000)
job_monitor(trabalho)

resultadoIBM = trabalho.result()
titulo = 'Probabilidades'
plot_histogram(resultadoIBM.
 ↪get_counts(teleporte), title=titulo)
```

A Fig. 13 (a) apresenta o circuito que foi construído a partir dos Boxes 23 a 26. Ao final do processo, realizamos uma medida no qubit de Bob e obtemos a distribuição de probabilidade correspondente. A Fig. 13 (b) apresenta a distribuição de probabilidade para o qubit de Bob após a realização do circuito em um processador quântico real.

Como pode ser visto, nota-se que o estado medido no qubit de Bob foi, em boa aproximação, o estado enviado por Alice, conforme a equação (26). O resultado

 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

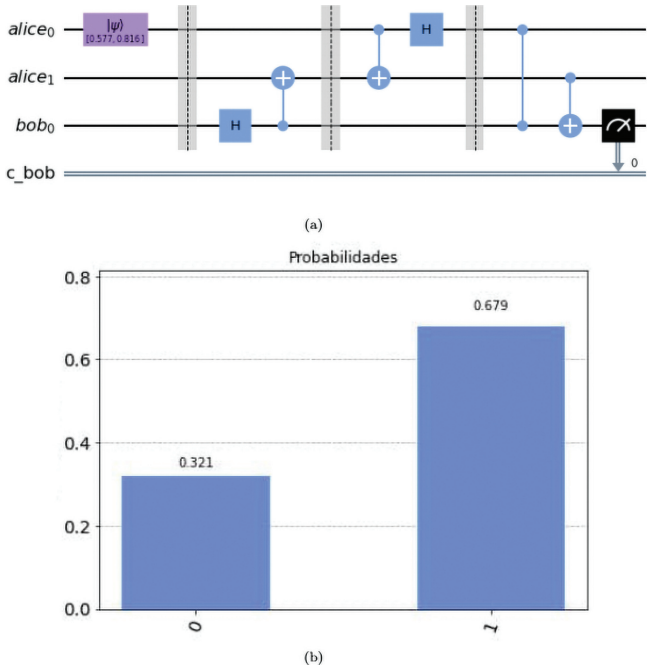


Figura 13: (a) Representação do circuito de teleporte. (b) Distribuição de probabilidade para o algoritmo de teleporte quântico do estado $\sqrt{\frac{1}{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle$ executado em um processador quântico real.

apresentado está de acordo com a margem de erro esperada para o algoritmo de teleporte, executado no processador IBM QE de 5 qubits, conforme reportado na literatura [3, 4].

4.3. Algoritmo de busca

Um dos algoritmos mais importantes da computação quântica, e uma das principais aplicações do poder computacional do computador quântico quando comparado com um computador clássico, é o Algoritmo de Grover [5, 14, 42].

A busca em uma lista não estruturada é um problema bastante comum nos cursos de programação. Consideremos um banco de dados não estruturado com N entradas. Nosso problema é determinar o índice da entrada (x) do banco de dados que satisfaça algum critério de pesquisa. Para isso, definimos a função resposta $(r(x))$, uma função que mapeia classicamente as entradas do banco de dados para `True` (`0`) ou `False` (`1`), onde $r(x) = 0$ se, e somente se, x satisfaz o critério de pesquisa $(x = p)$, onde p é o elemento procurado. Para isso, usamos uma subrotina conhecida como Oráculo, que realiza consultas à lista até encontrar o elemento p. Quanto mais distante o elemento procurado estiver na lista, maior será o número de consultas que o Oráculo precisará fazer para encontrar o elemento. Em média, a complexidade desse problema requer que o Oráculo consulte a lista $\frac{N}{2}$ vezes [5, 14, 43]. Se o elemento estiver no final da lista, o Oráculo precisará consultá-la N vezes. Logo, dizemos que o grau de complexidade desse problema é de ordem $\mathcal{O}(N)$. Quanticamente, o problema de busca em uma lista não estruturada é abordado no famoso Algoritmo de Grover [5, 14]. Explorar a superposição dos estados quânticos inspecionando os N itens da lista simultaneamente permite acelerar quadraticamente o problema de busca. O algoritmo de Grover é um algoritmo poderoso e sua utilidade vai além desse uso, sendo empregado como subrotina de otimização em uma grande variedade de outros algoritmos [5, 43–46], através do que chamamos de processo de amplificação de amplitude [5].

Como exemplo, apresentaremos a construção do Algoritmo de Grover no *Qiskit* para a implementação do algoritmo de busca simples para 3 qubits [5, 43], em um processador quântico real. Os elementos da lista, nesse caso, são codificados na base computacional para 3 qubits $\{|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle\}$. O Algoritmo de Grover é dividido em 4 partes principais: Superposição, Oráculo, Amplificação e a Medida [5, 43].

Para inicializar os qubits em uma superposição balanceada, utilizamos o método apresentado na seção 3.3, aplicando a porta Hadamard em todos os qubits no processo de inicialização e obtendo o estado

$$|\Psi_1\rangle = \frac{1}{2\sqrt{(2)}}[|000\rangle + |001\rangle + |010\rangle + |011\rangle + \\ + |100\rangle + |101\rangle + |110\rangle + |111\rangle]. \quad (27)$$

Entretanto, podemos implementar o código que gera o estado da equação (27) durante a inicialização do algoritmo principal. Antes, podemos construir as duas subrotinas auxiliares que formam o Algoritmo de Grover: o Oráculo e a Amplificação.

4.3.1. Oráculo

A função principal do Oráculo é marcar o elemento procurado na superposição [5]. Existem diferentes métodos que implementam essa subrotina [5], os dois principais são o booleano e o de inversão de fase [5, 43]. No método booleano, é necessário a presença de um qubit auxiliar (*ancilla*) inicializado no estado $|1\rangle$, sendo alterado somente se a entrada para o circuito for o estado procurado. Entretanto, este método equivalente ao método de marcação do problema de busca clássica [5, 43] é útil para comparar o poder de computação de um computador clássico frente a um computador quântico [43].

Como o objetivo desse trabalho é mostrar a aplicação de algoritmos quânticos em um processador quântico real usando o *Qiskit* como uma ferramenta de ensino de computação quântica, optamos pelo método mais simples, o método de inversão de fase [5, 43]. Nesse método não precisamos de uma *ancilla*. A função do Oráculo nesse processo é identificar o elemento procurado na superposição equiprovável dos estados da base computacional descrita acima e adicionar uma fase negativa. Nesse contexto, o Oráculo pode ser representado pela

operação unitária:

$$U_p|x\rangle = \begin{cases} -|x\rangle & \text{se } x = p, \\ |x\rangle & \text{se } x \neq p, \end{cases} \quad (28)$$

onde U_p é uma matriz diagonal que adiciona uma fase negativa à entrada que corresponde ao item procurado. U_p pode ser codificada em um circuito quântico dependendo do item desejado. Os circuitos que implementam a subrotina Oráculo descrita na equação (28) em cada estado da base computacional para 3 qubits é apresentado na referência [43].

Suponhamos que o elemento procurado seja $|p\rangle = |111\rangle$. O circuito que implementa U_p é a porta Z-multicontrolada, que pode ser construída pela combinação da porta Toffoli e Hadamard, conforme apresentado no Box 27[14].

Box 27: Iniciando o circuito e definindo o Oráculo

```
# Registrando os Qubits e os Bits

q = QuantumRegister(3, 'q')
b = ClassicalRegister(3, 'b')

# Definindo a subrotina Oráculo

oraculo = QuantumCircuit(q,name = "Oráculo")
oraculo.h(q[2])
oraculo.ccx(q[0],[1],q[2])
oraculo.h(q[2])
oraculo.draw(output = 'mpl')
```

Assim, aplicando a equação (28) na equação (27), o estado após a implementação do Box 27 será:

$$|\Psi_{oraculo}\rangle = \frac{1}{2\sqrt{(2)}}[|000\rangle + |001\rangle + |010\rangle + |011\rangle + \\ + |100\rangle + |101\rangle + |110\rangle - |111\rangle], \quad (29)$$

adicionando uma fase negativa ao elemento $|111\rangle$.

A Fig. 14 apresenta o circuito que implementa a subrotina Oráculo construído no Box 27.

Nesse ponto, mesmo tendo indicado o elemento procurado com uma fase negativa, a rotina Oráculo é insuficiente para obtermos o estado procurado se realizarmos

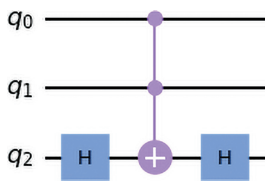


Figura 14: Oráculo para encontrar o estado $|111\rangle$.

[14] Vale destacar que não é necessário adicionar bits clássicos ao circuito pois as medidas só são executadas ao final do algoritmo principal.

uma medida em nossa superposição, uma vez que a fase adicionada pelo Oráculo não muda a distribuição de probabilidade.

Precisamos amplificar a probablilidade do elemento procurado $|p\rangle$ para aumentar a chance de encontrá-lo em uma medida no estado superposto, e reduzir as probabilidades dos demais estados da base $|x\rangle$, qualquer que seja $x \neq p$. Para isso, vamos usar o conhecido processo de Amplificação de Amplitude [5, 14, 43].

4.3.2. Amplificação de amplitude

A função da amplificação de amplitude é, como o próprio nome indica, aumentar a probabilidade do elemento marcado pelo Oráculo no estado $|\Psi_{oraculo}\rangle$, equação (29), reduzindo, consequentemente, as probabilidades dos demais itens [43]. Esse processo pode ser descrito em 5 subetapas [5, 43]:

1. Aplicar a porta Hadamard em todos os qubits do estado $|\Psi_{oraculo}\rangle$, equação (29), obtendo:

$$|\Psi_1\rangle = \frac{3}{4}|000\rangle\frac{1}{4}[|001\rangle + |010\rangle - |011\rangle - \\ -|100\rangle + |101\rangle - |110\rangle + |111\rangle]; \quad (30)$$

2. Aplicar a porta X em todos os qubits do estado $|\Psi_1\rangle$, obtendo:

$$|\Psi_2\rangle = \frac{3}{4}|111\rangle\frac{1}{4}[|000\rangle - |001\rangle - |010\rangle + \\ +|011\rangle - |100\rangle + |101\rangle + |110\rangle]; \quad (31)$$

3. Aplicar a porta Z-multicontrolada no estado $|\Psi_2\rangle$, obtendo:

$$|\Psi_3\rangle = -\frac{3}{4}|111\rangle\frac{1}{4}[|000\rangle - |001\rangle - |010\rangle + \\ +|011\rangle - |100\rangle + |101\rangle + |110\rangle]; \quad (32)$$

4. Aplicar novamente a porta X em todos os qubits do estado $|\Psi_3\rangle$, obtendo:

$$|\Psi_4\rangle = -\frac{3}{4}|000\rangle\frac{1}{4}[|001\rangle + |010\rangle - |011\rangle + \\ +|100\rangle - |101\rangle - |110\rangle + |111\rangle]; \quad (33)$$

5. Finalizando o processo aplicando novamente a porta Hadamard em todos os qubits do estado $|\Psi_4\rangle$, obtendo o estado final

$$|\Psi_5\rangle = \frac{5}{4\sqrt{2}}|111\rangle + \frac{1}{4\sqrt{2}}[|000\rangle + |001\rangle + |010\rangle + \\ +|011\rangle + |100\rangle + |101\rangle + |110\rangle]. \quad (34)$$

O Box 28 apresenta a construção do circuito de amplificação, conforme descrito nessas 5 etapas:

 DOI: https://doi.org/10.1590/1806-9126-RBEF-2021-0033

Box 28: Criando a rotina de reflexão

```
# Definindo a subrotina Amplificação

ampl = QuantumCircuit(q,name =␣
 ↪"Amplificação")

# Aplicar transformação | s> -> | 00..0>␣
 ↪(porta H em todos os qubits)
ampl.h([q[0],q[1],q[2]])
# Aplicar transformação | 00..0> -> | 11..
 ↪1> (portas X)
ampl.x([q[0],q[1],q[2]])
ampl.barrier()
# Construindo a porta CCZ
ampl.h(q[2])
ampl.ccx(q[0],q[1],q[2])
ampl.h(q[2])
ampl.barrier()
# Transformando o estado de volta

# Aplicar transformação | 11..1> -> | 00..
 ↪0> (portas X)
ampl.x([q[0],q[1],q[2]])
# Aplicar transformação | 00..0> -> | s> ␣
 ↪(porta H em todos os qubits)
ampl.h([q[0],q[1],q[2]])
ampl.draw(output = 'mpl')
```

A Fig. 15 apresenta o circuito que implementa a subrotina de amplificação de amplitude construída no Box 28.

Assim, chegamos ao estado final da subrotina de amplificação de amplitude, equação (34). O algoritmo de Grover é finalizado realizando uma medida sobre esse estado. Como pode ser visto na equação (34), a probaiilidade de encontrarmos o estado procurado $|111\rangle$ aumenta em detrimento das probabilidades dos demais estados da base computacional para 3 qubits, caracterizando o processo de amplificação de amplitude. Se realizarmos uma medida sobre o estado $|\Psi_5\rangle$, equação (34), a chance de obtermos o estado $|111\rangle$ é de aproximadamente $78,1\%$. Se quisermos aumentar ainda mais essa probabilidade, repetimos as subrotinas do Oráculo e de amplificação até atingir 100%. De maneira geral, para uma lista não estruturada de N itens, a maximização da probabilidade de encontrar o estado procurado é obtida repetindo essas

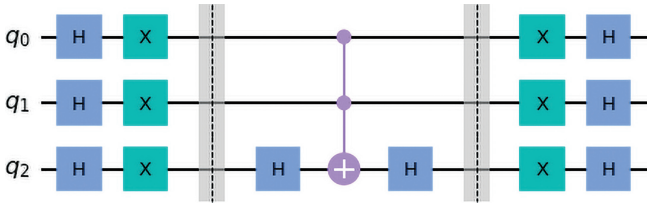


Figura 15: Circuito de amplificação de probabilidades para o algoritmo de Grover de 3 qubits.

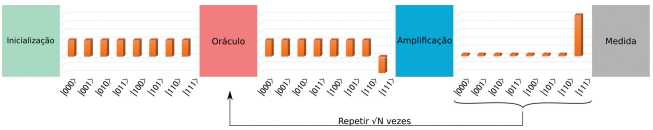


Figura 16: Esquematização de cada etapa do algoritmo de grover, mostrando a evolução das amplitudes para cada estado da base computacional para 3 qubits.

duas subrotinas $\mathcal{O}\left(\sqrt{N}\right)$ vezes [5, 43]. Por outro lado, o algoritmo clássico de busca em uma lista não estruturada precisa realizar uma média de $\frac{N}{2}$ consultas à lista para obter o elemento procurado [5, 43].

4.3.3. Executando o algoritmo

A Fig. 16 mostra uma representação esquemática para a evolução das amplitudes para cada estado da base computacional para 3 qubits em cada etapa do algoritmo de Grover: (i) A inicialização cria uma superposição balanceada de todos os estados de entrada possíveis $\{|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle\}$; (ii) O Oráculo marca o estado desejado de modo que a amplitude do estado procurado $|p\rangle$ será negativa enquanto as demais amplitudes $|x\rangle$ são mantidas inalteradas; (iii) A amplificação aumenta a probabilidade de encontrarmos o estado marcado pelo Oráculo; (iv) O processo pode agora ser finalizado realizando medidas sobre todos os qubits obtendo-se o estado procurado após repetir os passos ii e iii $\mathcal{O}\left(\sqrt{N}\right)$ vezes.

Vale destacar que aumentar o número de repetições dos estágios de Oráculo e amplificação maximizará a amplitude da resposta correta [5, 14, 43]. Além disso, esse algoritmo também pode ser generalizado para marcar e amplificar a amplitude de mais de um estado [5, 43].

Vamos agora criar o circuito principal que implementa o algoritmo de busca através da união das subrotinas Oráculo e Amplificação. Primeiramente precisamos inicializar os qubits em uma superposição balanceada, como vimos na seção 3.3, aplicando a porta Hadamard em todos os qubits no processo de inicialização para criar o estado $|\Psi_i\rangle$, equação (27). Em seguida, usando o comando `grover.append()` adicionamos as subrotinas Oráculo e Amplificação, criadas nos Boxes 27 e 28. Finalmente realizamos as medidas e finalizamos o Algoritmo de Grover conforme apresentado no Box 29, a seguir.

Box 29: Criando o circuito de busca

```
grover = QuantumCircuit(q,b)
grover.h([q[0],q[1],q[2]])
grover.barrier()
grover.append(oraculo,q)
grover.barrier()
grover.append(ampl,q)
grover.barrier()
grover.measure(q,b)
```

```
grover.draw(output = 'mpl')

provedor = IBMQ.get_provider('ibm-q')
comput = provedor.
 ↪get_backend('ibmq_valencia')
trabalho = execute(grover, backend=comput,␣
 ↪shots = 8000)
job_monitor(trabalho)

resultadoIBM = trabalho.result()
titulo = 'Probabilidades'
plot_histogram(resultadoIBM.
 ↪get_counts(grover), title=titulo)
```

A Fig. 17 apresenta o Algoritmo completo de Grover com as subrotinas Oráculo e Amplificação.

Finalmente, após as medidas, executamos o algoritmo de Grover em um processador quântico real e obtemos a distribuição de probabilidade correspondente. Como pode ser visto na Fig. 18, obtemos o item procurado em $76,6\%$ das 1024 repetições. Isso significa que em uma única busca teríamos aproximadamente $76,6\%$ de chances de encontrar o elemento procurado com sucesso. Em contra partida, classicamente a chance de encontrar um item em uma lista não estruturada com $N = 8$ elementos, executando somente uma consulta à lista, é de $12,5\%$, o que mostra a vantagem de usarmos propriedades quânticas como a superposição para o processamento da informação. Enquanto classicamente o Oráculo precisa em média realizar $N/2$ consultas a lista, quanticamente podemos encontrar o item marcado em $\sqrt{N}$ tentativas, com o método de amplificação de amplitude de Grover para o problema de busca [5, 43]. Portanto, a junção das subrotinas Oráculo e Amplificação para a construção do algoritmo de Grover, representa uma aceleração quadrática do problema de busca, mostrando que computadores quânticos possuem uma vantagem significativa se comparados a computadores clássicos.

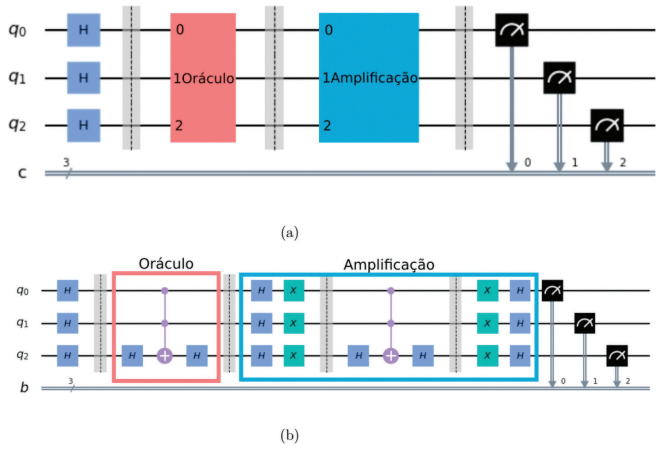


Figura 17: Algoritmo completo de Grover.

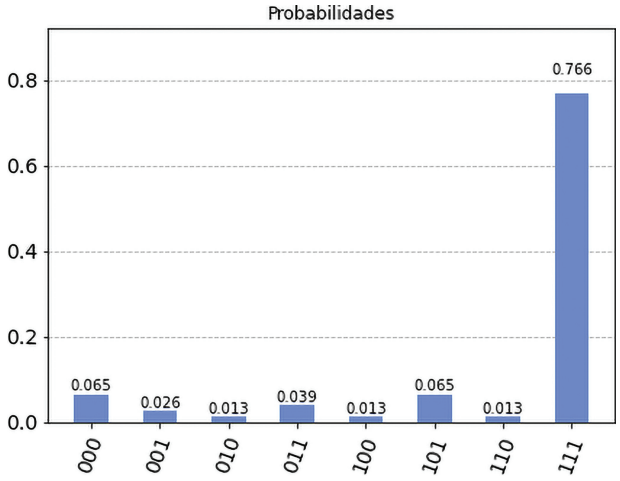


Figura 18: Distribuição de probabilidade obtida para o algoritmo de Grover.

5. Conclusão

Nesse trabalho, apresentamos o kit de desenvolvimento de *software* para informação quântica da IBM (*Qiskit*) como uma ferramenta de trabalho para o ensino de computação e informação quântica para os cursos de graduação em Física e áreas afins. O trabalho está estruturado na forma de um roteiro básico de sala de aula para a introdução de conceitos fundamentais da computação quântica, como qubits, portas quânticas, emaranhamento e algoritmos quânticos. Destacamos as principais condições para a construção dos programas e a sua execução em processadores quânticos reais, mostrando como essa pode ser uma ferramenta poderosa para o ensino de computação quântica de maneira prática, permitindo que os estudantes se tornem agentes ativos na construção do conhecimento. Nossos resultados estão de acordo com as previsões teóricas da literatura para os exemplos abordados, e demonstram que o *Qiskit* é uma ferramenta eficaz tanto para a implementação e a análise de algoritmos quânticos simples, quanto para o desenvolvimento de *softwares* quânticos, atuando como uma linguagem de programação quântica de alto nível acessível aos estudantes.

Agradecimentos

Os autores gostariam de agradecer a toda a equipe do *IBM Research* e do *Quantum Education & Open Science at IBM Quantum* pelo acesso ao *Qiskit*, e toda a comunidade do *Qiskit* pelo suporte prestado ao longo do desenvolvimento desse trabalho. C. Cruz agradece a W. S. Santana pela leitura do material, a E.H.M. Maschio pelas discussões proveitosas. Este artigo traz um resumo das notas de aula da disciplina CET0448 – Tópicos Especiais III: Computação Quântica Aplicada, ministrada para estudantes dos cursos de Licenciatura e Bacharelado em Física da Universidade Federal do

Oeste da Bahia. Os autores agradecem aos demais estudantes da disciplina CET0448 – Tópicos Especiais III: Computação Quântica Aplicada, que mesmo não participando ativamente desse trabalho contribuiram para sua concepção. Os autores agradecem também aos revisores pelo tempo despendido na avaliação desse artigo. Seus comentários e sugestões, contribuiram para melhorar a qualidade do nosso trabalho.

REFERÊNCIAS

[1] https://quantum-computing.ibm.com, acessado em 25/08/2020.

[2] E.M. Alves, F.D.S. Gomes, H.S. Santana e A.C. Santos, Revista Brasileira de Ensino de Física **42**, e20190299 (2020).

[3] A.C. Santos, Revista Brasileira de Ensino de Física **39**, e1301 (2017).

[4] W.R.M. Rabelo e M.L.M. Costa, Revista Brasileira de Ensino de Física **40**, e4306 (2018).

[5] M.A. Nielsen e I. Chuang, *Quantum computation and quantum information* (Cambridge University Press, Cambridge, 2002).

[6] V. Scarani, American Journal of Physics **66**, 956 (1998).

[7] A. Steane, Reports on Progress in Physics **61**, 117 (1998).

[8] I.S. Oliveira, *Física Quântica: fundamentos formalismos e aplicações* (Editora Livraria da Física, São Paulo, 2020), v. 1.

[9] B.M. Terhal, Nature Physics **14**, 530 (2018).

[10] A.W. Harrow e A. Montanaro, Nature **549**, 203 (2017).

[11] M.A. José, J.R.C. Piqueira e R.D. Lopes, Revista Brasileira de Ensino de Física **35**, 1 (2013).

[12] D. Candela, American Journal of Physics **83**, 688 (2015).

[13] S. Fedortchenko, arXiv:1607.02398 (2016).

[14] J.E. Castillo, Y. Sierra e N.L. Cubillos, Revista Brasileira de Ensino de Física **42**, e20190115 (2020).

[15] A. Perry, R. Sun, C. Hughes, J. Isaacson e J. Turner, arXiv:1905.00282 (2019).

[16] A.C. Teixeira e E.J.R. Brandão, Revista Novas Tecnologias na Educação **1**, 1 (2003).

[17] E.V. Faria, Revista Scientia FAER **1**, 18 (2009).

[18] C.C. Tappert, R.I. Frank, I. Barabasi, A.M. Leider, D. Evans e L. Westfall, em: *Proceedings of the 2019 ASCUE Summer Conference* (Myrtle Beach, 2019).

[19] https://qiskit.org/textbook, acessado em 28/08/2020.

[20] https://qiskit.org/documentation, acessado em 29/08/2020.

[21] https://github.com/Qiskit, acessado em 28/08/2020.

[22] Qiskit: An open-source framework for quantum computing, 2019, disponível em: https://doi.org/10.5281/zenodo.2562110.

[23] R. LaRose, Quantum **3**, 130 (2019).

[24] T.E. Oliphant, Computing in Science & Engineering **9**, 10 (2007).

[25] G. Van Rossum e F.L. Drake, *The python language reference manual* (Network Theory, S.l, 2011).

[26] A. Kadiyala e A. Kumar, Environmental Progress & Sustainable Energy **36**, 1580 (2017).

[27] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith et al., Nature **585**, 357 (2020).

[28] https://jupyter.org, acessado em 20/08/2020.

[29] A. Cardoso, J. Leitão e C. Teixeira, *The Challenges of the Digital Transformation in Education. ICL 2018. Advances in Intelligent Systems and Computing* (Springer, Cham, 2018).

[30] Z. Hussain e M.S. Khan, International Journal of Computer Science and Network Security **18**, 26 (2018).

[31] Anaconda Software Distribution, Computer software Anaconda v.2-2.4.0, acessado em 20/08/2020: https://anaconda.com.

[32] https://github.com/qiskit-community/qiskit-swift, acessado em 15/02/2021.

[33] https://github.com/qiskit-community/qiskit-js, acessado em 15/01/2021.

[34] J.D. Hunter, Computing in science & engineering **9**, 90 (2007).

[35] V. Vedral, *Introduction to quantum information science* (Oxford University Press on Demand, Oxford, 2006).

[36] V.A. Pedroni, *Digital electronics and design with VHDL* (Morgan Kaufmann, Amsterdam, 2008).

[37] D.J. Griffiths, *Mecânica Quântica* (Editora Pearson Education, New Jersey, 2011), 2ª ed.

[38] J.E. Whitesitt, *Boolean algebra and its applications* (Courier Corporation, North Chelmsford, 2012).

[39] R. Horodecki, P. Horodecki, M. Horodecki e K. Horodecki, Reviews of modern physics **81**, 865 (2009).

[40] A. Einstein, B. Podolsky e N. Rosen, Physical review **47**, 777 (1935).

[41] E. Schrödinger, Naturwissenschaften **23**, 823 (1935).

[42] L.K. Grover, Physical review letters **79**, 325 (1997).

[43] C. Figgatt, D. Maslov, K.A. Landsman, N.M. Linke, S. Debnath e C. Monroe, Nature communications **8**, 1 (2017).

[44] F. Magniez, M. Santha e M. Szegedy, SIAM Journal on Computing, **37**, 413 (2007).

[45] C. Dürr, M. Heiligman, P. HOyer e M. Mhalla, SIAM Journal on Computing **35**, 1310 (2006).

[46] C.H. Bennett, E. Bernstein, G. Brassard e U. Vazirani, SIAM journal on Computing **26**, 1510 (1997).