UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Síntese de Circuitos Quânticos usando *Projective Simulation*

Otto Menegasso Pires

Orientador: Prof. Dr. Eduardo Inacio Duzzioni
Coorientador: Profa. Dra. Jerusa Marchi

Florianópolis
2019

Otto Menegasso Pires

Síntese de Circuitos Quânticos usando *Projective Simulation*

Trabalho de Conclusão de Curso submetido ao Departamento de Informática e Estatística da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Eduardo Inacio Duzzioni
Coorientador: Profa. Dra. Jerusa Marchi

Florianópolis
2019

Resumo

A computação quântica é uma área que vem evoluindo muito nos últimos anos. Embora os algoritmos quânticos desenvolvidos atualmente tenham demostrado superioridade em relação às suas contrapartes clássicas, fatores como tempo de decoerência de um qubit e a necessidade de qubits auxiliares para rotinas de tolerância a erro tem se mostrado grandes barreiras no uso efetivo de algoritmos quânticos. Por causa dessas restrições e da escassez de recursos computacionais nos computadores quânticos atuais, buscam-se maneiras de minimizar o custo envolvido em um algoritmo. Para isso são desenvolvidas técnicas para síntese e otimização de circuitos quânticos. Síntese de circuitos quânticos engloba técnicas para se produzir um circuito que seja capaz de realizar uma determinada tarefa. Muitas técnicas de síntese não garantem a otimalidade de seu circuito criado, sendo necessário um processo de otimização do circuito após a síntese. Esse trabalho busca estudar as técnicas existentes de síntese de circuitos quânticos, indicando o atual estado da arte e por fim implementa seu próprio sintetizador usando uma técnica recente e pouco explorada conhecida como *Projective Simulation*. O novo sintetizador demonstrou-se capaz de sintetizar circuitos quânticos de dois qubits, tendo seu desempenho avaliado a partir de sua capacidade de criar os circuitos geradores dos estados de Bell.

Palavras-chave:
computação quântica, algoritmos quânticos, circuitos quânticos, síntese de circuitos quânticos.

Abstract

Quantum Computation is a field of research that has been evolving in the last years. Although nowadays quantum algorithms have shown themselves superior to their classical counterparts, quantum decoherence and the need for additional auxiliary qubits for error tolerance routines have been great barriers in the efficient use of quantum algorithms. Because of these restrictions and the shortage in computational resources of current quantum computers, it is necessary to search for ways to minimize the cost of an algorithm. Quantum Circuit Synthesis includes technics to produce a circuit that is capable to do a given task. Many technics can't guarantee the optimality of the resulting circuit, so an optimization process is needed after the synthesis. This work seeks to study the state of art technics for quantum circuit synthesis and implements its own synthesizer using a recent technique called Projective Simulation. The new synthesizer was able to synthesize quantum circuits up to two qubits, and its performance was measured using the Bell states circuits as a benchmark.

Key-words:
Quantum Computation. Quantum Algorithms. Quantum Circuits Synthesis.

Lista de Figuras

Sumário

Introdução

Por mais de 20 anos o desempenho dos computadores cresceu a uma taxa de mais de 50% ao ano [10, figure 1.1]. Esse fenômeno ficou conhecido como Lei de Moore e havia reinado supremo até o início do século XXI, quando a taxa de crescimento de desempenho começou a decrescer. Atualmente, com o fim da Lei de Moore, busca-se maneiras alternativas para se obter mais poder computacional, e uma das alternativas que vem chamando muita atenção é mudar completamente o paradigma de computação para a Computação Quântica.

Embora apenas recentemente os primeiros computadores quânticos estejam sendo construídos, o modelo de computação quântica vem sendo estudado há bastante tempo. Richard Feynman em 1982, na sua palestra "Simulating Physics with Computers" [8], já apontava os limites de um computador clássico para simular sistemas físicos quânticos e sugeria a construção de um computador que operasse diretamente através das leis da mecânica quântica.

O que antes era de interesse apenas dos físicos mudou completamente quando Peter Shor publicou seu algoritmo de fatoração de números primos em tempo polinomial [22]. Esse fato mostrou que existe uma classe de problemas reconhecidamente difíceis no caso clássico que podem ser resolvidos em tempo hábil em um computador quântico. Essa descoberta junto do desenvolvimento de outros algoritmos, como o algoritmo de busca de Grover, demonstrou um grande potencial a ser aproveitado no uso da computação quântica.

Atualmente a Computação Quântica encontra-se em um estágio similar ao da própria Ciência da Computação na década de 70. Muitos computa-

dores quânticos atuais possuem apenas de 10 a 20 qubits de processamento, sendo que os maiores possuem 50 ou 70 [2]. Essa escassez de recursos torna necessária a implementação ótima dos circuitos quânticos a serem testados, visto que as operações básicas da computação quântica não possuem um custo negligenciável [3].

Para garantir a construção de circuitos ótimos busca-se técnicas de síntese e otimização de circuitos. Uma alternativa para a síntese de circuitos quânticos atualmente estudada é sintetizar circuitos booleanos reversíveis e implementá-los em um computador quântico. Entretanto, muitas das técnicas existentes não garantem a síntese de circuitos ótimos, tornando necessária uma etapa de otimização nos circuitos gerados automaticamente.

1. Objetivos

1.1 Objetivo Geral

Esse trabalho de conclusão de curso tem como objetivo geral desenvolver um sintetizador de circuitos quânticos próprio usando-se de técnicas de aprendizado por reforço.

1.2 Objetivos Específicos

1.21 Levantamento de técnicas atuais de síntese de circuitos booleanos reversíveis

1.22 Realizar um estudo mais aprofundado da técnica de *Projective Simulation* usado na geração de experimentos físicos e de sua viabilidade na síntese de circuitos quânticos [15].

1.23 Implementar um sintetizador de circuitos quânticos usando a técnica de *Projective Simulation*.

1.24 Testar a capacidade do novo sintetizador implementado, usando simulações para gerar circuitos quânticos.

1.3 Estruturação do Trabalho

Esse trabalho é estruturado da seguinte maneira: Capítulo 1 introduz os conceitos necessários para se compreender o modelo de computação quântica circuital. As técnicas de síntese de circuitos quânticos são analisadas no Capítulo 2. O Capítulo 3 descreve os detalhes da implementação do sintetizador e o Capítulo 4 mostra os resultados da implementação do sintetizador próprio. Ao fim têm-se as Considerações Finais e Trabalhos futuros.

Capítulo 1

Computação Quântica Circuital

A Ciência da Computação está atualmente em um período de mudança. No início deste século testemunhou-se uma queda na taxa de crescimento da performance dos processadores, culminando no fim da Lei de Moore. Por causa disso busca-se novas formas de aprimorar a performance dos computadores atuais, e uma dessas formas é mudar completamente de paradigma.

Historicamente, a Computação Quântica nasceu na Física, tendo como um de seus objetivos iniciais simular de maneira eficiente sistemas físicos quânticos. Após a criação do Algoritmo de Shor para fatoração de números primos a área vem chamando mais atenção para si e não apenas dos físicos. Shor e seu algoritmo provaram que existem problemas intratáveis de maneira clássica que podem ser resolvidos eficientemente em um computador quântico.

Existem diferentes modelos de computação quântica. O modelo abordado nesse trabalho é a Computação Quântica Circuital. Esse modelo tem algumas propriedades análogas aos circuitos elétricos da computação clássica como, por exemplo, a computação ser efetuada através da aplicação de portas lógicas.

Nesse capítulo serão abordados alguns conceitos básicos para a compreensão do modelo de Computação Quântica Circuital, além de compará-los com os circuitos elétricos clássicos. Serão omitidas algumas provas mais extensas que fogem do escopo desse trabalho. Para um estudo aprofundado e mais didático da Matemática e da Física envolvida na computação quântica é possível ver as referências [17] [16] [7].

1.1 Bits quânticos

Assim como na computação clássica nós temos o conceito de *bit*, na computação quântica e na informação quântica existe o *quantum bit*, ou apenas *qubit*.

O qubit é a unidade básica de informação quântica. Assim como o bit pode assumir os valores 0 e 1, o qubit pode assumir os valores $|0\rangle$ e $|1\rangle$. Entretanto, uma diferença fundamental do qubit para sua contraparte clássica é que o qubit pode assumir valores intermediários entre estados, descritos como combinações lineares entre $|0\rangle$ e $|1\rangle$.

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

Onde os números α e β são números complexos. Esses estados intermediários são chamados de *superposições* e os estados $|0\rangle$ e $|1\rangle$ são chamados de base computacional.

Um desafio enfrentado pela computação quântica é que não é possível descobrir qual é o estado quântico de um qubit. Quando medimos um qubit recebemos ou o valor 0 com probabilidade de $|\alpha|^2$, ou 1 com probabilidade de $|\beta|^2$ sendo que $|\alpha|^2 + |\beta|^2 = 1$. Essa condição é chamada *condição de normalização* e ela garante que a soma das probabilidades dos valores a serem colapsados resulte em 100%. Caso após uma medição tenha-se encontrado o valor 0, toda medição posterior feita nesse mesmo qubit retornará apenas o valor 0. Quando isso acontece dizemos que o estado do qubit *colapsou*. Atualmente ninguém sabe explicar porque esse fato ocorre.

Um modo interessante de visualizar um qubit é como um ponto em uma esfera tridimensional, que chamamos de *esfera de Bloch*. Partindo de $|\alpha|^2 + |\beta|^2 = 1$ então é possível reescrever a equação anterior como:

$$|\psi\rangle = e^{i\gamma}(\cos\frac{\theta}{2}\,|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}\,|1\rangle)$$

Onde θ, γ e φ são números reais. É possível provar que o fator $e^{i\gamma}$ não tem nenhum efeito observável, então a equação pode ser reescrita como:

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}\,|1\rangle$$

Os números θ e φ representam um ponto dentro de uma esfera unitária. Muitas das operações realizadas em um qubit se encaixam perfeitamente

nesse modelo, porém a esfera de Bloch é apenas uma representação limitada pois não existe uma generalização para múltiplos qubits.[1]

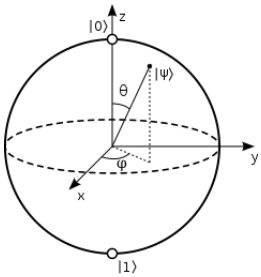


Figura 1.1: Esfera de Bloch.

1.1.1 Múltiplos qubits

Não é possível realizar muitos cálculos com apenas uma unidade de informação, então é preciso maneiras para representar e manipular quantidades maiores. Em um computador clássico usando n bits podemos representar até 2^n estados diferentes. Em um computador quântico, com n qubits nós temos uma base computacional de 2^n estados.

Por exemplo, usando dois bits pode-se criar os estados 00, 01, 10 e 11. De maneira análoga, um sistema com dois qubits possui quatro estados em sua base computacional: $|00\rangle$, $|01\rangle$, $|10\rangle$ e $|11\rangle$. O sistema de 2 qubits também podem formar superposições desses estados, sendo representados por:

$$|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$$

Assim como no caso com um único qubit ao medirmos o estado do nosso sistema ele irá colapsar para algum valor da base computacional. Nesse caso obtemos $|x\rangle$ com a probabilidade $|\alpha_x|^2$, também chamada de amplitude. Assim como no caso com apenas um qubit, a soma de todas as amplitudes precisa ser igual a 1.

É possível medir apenas parte dos qubits que formam o sistema. No exemplo acima, caso o primeiro qubit seja medido ele colapsaria em 0 com a probabilidade de $|\alpha_{00}|^2+|\alpha_{01}|^2$ ou em 1 com a probabilidade $|\alpha_{10}|^2+|\alpha_{11}|^2$. No caso em que o qubit colapse para 0, o estado resultante do sistema seria:

[1]Figura 1.1 Fonte: https://commons.wikimedia.org/wiki/File:Bloch_sphere.svg

$$|\psi\rangle = \frac{\alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}$$

Nota-se que o estado foi renormalizado para que a soma de todas as amplitudes α continue igual.

1.1.2 Emaranhamento e Estados de Bell

Ao se operar sobre múltiplos qubits usa-se implicitamente da operação de produto tensorial. O produto tensorial é uma maneira de se juntar vetores para expandir o espaço vetorial com o qual se está trabalhando. A notação do produto tensorial entre dois vetores ψ e φ é $\psi \otimes \varphi$, porém por economia de notação é possível usar alternativas como $|\psi\varphi\rangle$

$$|\psi\rangle \otimes |\varphi\rangle \equiv |\psi, \varphi\rangle \equiv |\psi\varphi\rangle$$

Entretanto, existem alguns estados entre múltiplos qubits que não podem ser descritos como o produto tensorial de dois ou mais qubits. Esse fenômeno é chamado de *emaranhamento*. Quando dois qubits estão emaranhados o estado interno de um está vinculado ao estado interno do outro, dessa forma ao se medir um deles o outro irá inevitavelmente assumir um outro valor pré-determinado.

O exemplo mais simples de emaranhamento são os chamados *Estados de Bell* ou pares EPR. Os estados de Bell são formados por dois qubits e possuem a forma $|\beta_{xy}\rangle \equiv \frac{|0,y\rangle + (-1)^x |1,\overline{y}\rangle}{\sqrt{2}}$. É possível perceber que não se consegue separar a fórmula de um estado de Bell em um produto tensorial entre dois qubits e por consequência não se pode medir o valor de uma de suas partes sem colapsar todo o sistema.

Por exemplo, caso o primeiro qubit do estado de bell $|\beta_{00}\rangle$ seja medido e o valor obtido seja 0, o valor do segundo qubit também será igual a 0. Isso acontece por causa do efeito de emaranhamento, no qual o estado de um dos qubits está vinculado ao do outro.

A propriedade de emaranhamento permite o desenvolvimento de muitos circuitos interessantes, alguns notáveis sendo o circuito de teletransporte [16, section 1.3.7] e a codificação superdensa [16, section 2.3]. O circuito de teletransporte, apesar do nome, não permite transmissão de informação de forma instantânea mas a partir de suas propriedades estão intimamente conectadas com códigos de correção de erro quântico. A codificação superdensa é um modo de codificar dois bits de informação

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\beta_{00}\rangle$$
$$\frac{|01\rangle - |10\rangle}{\sqrt{2}} = |\beta_{01}\rangle$$
$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\beta_{10}\rangle$$
$$\frac{|01\rangle - |10\rangle}{\sqrt{2}} = |\beta_{11}\rangle$$

Figura 1.2: Estados de Bell

em apenas um qubit, podendo assim economizar o consumo de memória e banda em um computador quântico.

1.2 Circuitos Clássicos

Circuitos elétricos são um modelo de computação universal em que é possível modelar expressões booleanas e a partir disso construir qualquer operação matemática em um computador. Os circuitos são compostos por portas lógicas e fios. As portas lógicas realizam operações booleanas em um ou mais bits que são carregados através do circuito pelos fios.

As portas lógicas podem ser de um único bit ou de múltiplos bits. A única porta não trivial de um único bit é a porta NOT, que inverte o valor de sua entrada. As outras portas são normalmente representadas com dois bits, mas pode-se facilmente estender suas funções para mais entradas. As portas de múltiplos bits mais comuns são a OR e a AND, que realizam as operações lógicas de *ou* e *e* respectivamente. A porta OR retorna o valor 1 caso pelo menos uma das entradas possua o valor 1, senão retorna 0. A porta AND retorna 1 apenas se todas as suas entradas possuem o valor 1, do contrário retorna 0. Existem variantes dessas portas lógicas conhecidas como NAND e NOR, que possuem o mesmo comportamento mas sua saída é negada. Existe também outra variante da porta OR chamada de XOR, ou ou-exclusivo. A porta XOR retorna 1 se apenas uma de suas entradas possui o valor 1.

O comportamento de uma porta lógica pode ser analisado através de sua tabela verdade. Na tabela são listados em cada linha todas as combinações de entrada e o respectivo resultado da operação.

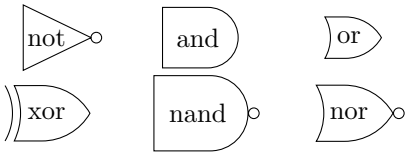


Figura 1.3: Exemplos de portas lógicas clássicas

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Usando de base as operações lógicas booleanas é possível realizar operações aritméticas com os bits. Por exemplo, o circuito somador chamado de *half-adder* é composto por uma porta XOR e uma porta AND. Esse circuito recebe de entrada dois bits x e y e possui de saída os valores S e C. O S representa a soma dos valores dos bits e C representa o *carry-out*, o "vai um" da operação de adição.

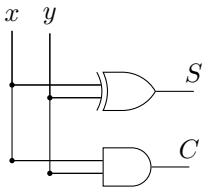


Figura 1.4: Circuito lógico de um half-adder

1.2.1 Universalidade de portas lógicas

Uma propriedade importante dos circuitos elétricos é a universalidade de algumas de suas portas lógicas. Essa propriedade diz que a partir

de uma única porta lógica é possível recriar qualquer operação booleana. Exemplos de portas lógicas universais são a NAND e a NOR.

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

Essa propriedade é muito interessante pois permite a redução de custo na construção de circuitos elétricos. Na tecnologia CMOS uma porta NAND precisa de menos transistores e ocupa uma área menor do que as outras portas lógicas, fazendo com que em muitas situações seja mais interessante construir um circuito usando apenas portas NANDs do que implementando diretamente as portas necessárias.

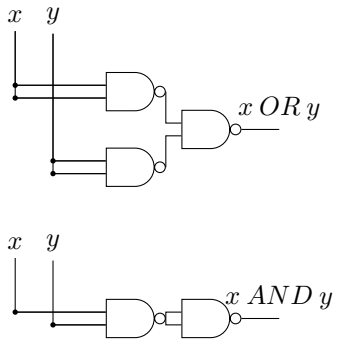


Figura 1.5: Operações lógicas básicas implementadas com portas NAND

1.2.2 Redução de Circuitos Clássicos

Em 1940 Claude Shannon relacionou circuitos elétricos com álgebra booleana. Esse trabalho permitiu não só a construção de qualquer circuito que representasse uma função lógica ou aritmética mas também o uso de álgebra booleana na simplificação de circuitos.

Através do uso de propriedades presentes na álgebra booleana é possível reduzir qualquer circuito construído de uma maneira ingênua para uma forma equivalente e com menos portas lógicas.

Na álgebra booleana a operação de negação pode ser representada por uma barra em cima da variável na qual essa operação é realizada (e.g.

$\overline{A} = NOT\ A$). A operação OR é representada pela adição e a AND pela multiplicação.

Identidade	$\overline{\overline{A}} = A$
Adição	$A + 0 = A$ $A + 1 = 1$ $A + A = A$ $A + \overline{A} = 1$
Multiplicação	$A.0 = 0$ $A.1 = A$ $A.A = A$ $A.\overline{A} = 0$
Comutatividade	$A + B = B + A$ $A.B = B.A$
Associatividade	$A + (B + C) = (A + B) + C$ $A.(B.C) = (A.B).C$
Distributividade	$A + (B.C) = A + B.A + C$ $A.(B + C) = A.B + A.C$
Absorção	$A + (A.B) = A$ $A.(A + B) = A$
De Morgan	$\overline{(A.B)} = \overline{A} + \overline{B}$ $\overline{(A + B)} = \overline{A}.\overline{B}$

Figura 1.6: Propriedades da Álgebra Booleana

O processo de redução ocorre através da manipulação dos termos da função lógica usando as propriedades da álgebra booleana, tentando buscar termos irrelevantes ao resultado e os eliminando. Por exemplo a função $A.B.C + A.\overline{C} + A.\overline{B}$ possui ao todo 3 termos e 8 operações. Porém, após o processo de redução, percebe-se que o único termo relevante é o A, permitindo então cortar todas as operações e os outros termos.

$$
\begin{aligned}
S &= A.B.C + A.\overline{C} + A.\overline{B} \\
&= A.(B.C + \overline{C} + \overline{B})(\text{distributividade}) \\
&= A.(B.C + (\overline{C} + \overline{B}))(\text{associatividade}) \\
&= A.(B.C + \overline{\overline{(\overline{C} + \overline{B})}})(\text{identidade do complemento}) \\
&= A.(B.C + \overline{(C.B)})(\text{De Morgan}) \\
&= A.(B.C + \overline{(B.C)})(\text{comutatividade}) \\
&= A.(1)(\text{identidade da adição } (D + \overline{D} = 1)) \\
&= A
\end{aligned}
$$

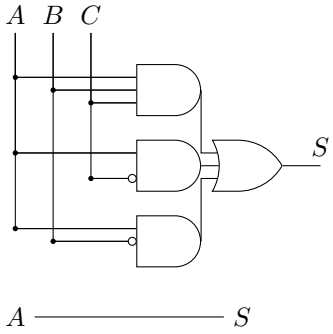


Figura 1.7: Circuito lógico antes e depois da otimização

1.3 Circuitos Quânticos

Assim como podemos representar operações lógicas e matemáticas de um computador clássico usando um circuito elétrico composto por portas lógicas e fios, também podemos descrever os cálculos da computação quântica usando uma notação de circuitos.

Em um circuito quântico cada qubit é representado por uma linha horizontal, com as portas lógicas a serem aplicadas no qubit sendo atravessadas por essa linha. As portas lógicas quânticas são representadas por caixas com o nome delas escrito dentro, com exceção de algumas portas que possuem uma notação mais econômica (e.g. SWAP e CNOT). A computação é efetuada da esquerda para a direita.

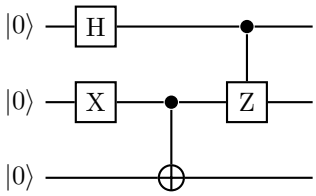


Figura 1.8: Um exemplo de circuito quântico.

O comportamento de uma porta lógica quântica é descrito por uma matriz unitária. Uma matriz U é dita unitária caso $U^\dagger U = I$, sendo que $U^\dagger$ é a matriz adjunta de U e I a matriz identidade. A matriz adjunta é a matriz transposta e complexa-conjugada.

Não existe um equivalente quântico das portas lógicas clássicas mais comuns. Isso se deve principalmente ao fato de as portas lógicas clássicas serem em sua grande maioria irreversíveis, enquanto a computação quântica é naturalmente reversível. Uma operação irreversível é uma operação que dada o valor da saída é impossível descobrir qual o valor das entradas que a originaram. Apesar disso, é possível realizar as operações lógicas básicas usando um circuito quântico reversível.

Uma das maneiras de se realizar operações lógicas básicas em um circuito quântico é usando portas Toffoli. A porta Toffoli é uma porta lógica reversível que recebe três entradas e possui três saídas. Chamamos as duas primeiras entradas de controle e a terceira de alvo. Caso as duas primeiras entradas sejam iguais a 1 então o valor da terceira entrada é negado. A porta Toffoli originalmente é uma porta lógica clássica reversível, entretanto existe uma matriz unitária capaz de recriar esse comportamento, fazendo com que exista uma porta Toffoli quântica.

A operação da porta Toffoli pode ser escrita como:

$$(C_1, C_2, T) \rightarrow (C_1, C_2, T \oplus (C_1.C_2))$$

Sendo $\oplus$ a operação de XOR na álgebra booleana. A partir dessa definição e fixando o valor de T em 1 então o resultado da aplicação da porta lógica será

$$\begin{aligned}(C_1, C_2, 1) &\rightarrow (C_1, C_2, 1 \oplus (C_1.C_2)) \\ &\rightarrow (C_1, C_2, \overline{C_1.C_2})\end{aligned}$$

Como foi dito anteriormente a operação NAND é universal e a partir dela é possível se obter todas as outras operações lógicas. Isso é suficiente para

Entradas			Saídas		
C_1	**C_2**	**T**	**C_1'**	**C_2'**	**T'**
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figura 1.9: Tabela verdade da porta de Toffoli e sua representação em matriz unitária

dizer que é possível recriar o comportamento de qualquer circuito clássico usando um circuito quântico.

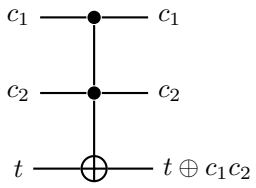


Figura 1.10: Representação em circuito de uma porta Toffoli

1.3.1 Portas lógicas de um único qubit

Como dito anteriormente, os qubits são unidades de informação na forma:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

Para manipular um qubit usa-se sua notação em vetor coluna, onde a primeira entrada é a amplitude de $|0\rangle$ e a segunda entrada a amplitude de $|1\rangle$.

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Partindo disso representamos as portas lógicas quânticas de um único qubit como matrizes dois por dois. Essa representação segue diretamente

do fato das portas quânticas agirem linearmente nos qubits [16, p. 18].

Foi dito na sessão anterior que as portas quânticas são representadas por matrizes unitárias. Essa restrição garante que a condição de normalização do qubit $|\alpha|^2 + |\beta|^2 = 1$ seja mantida mesmo após a ação da porta sobre o qubit. Essa é a única condição para uma porta lógica quântica, portanto qualquer matriz unitária representa uma porta quântica válida.

Ao comentar sobre qubits foi mencionado sua representação como um ponto em uma esfera unitária que chamamos de Esfera de Bloch. Essa representação ajuda a entender os efeitos das portas lógicas quânticas de um único qubit, pois elas podem ser interpretadas como rotações do qubit dentro dessa esfera. Algumas das operações mais conhecidas são as matrizes de Pauli, representadas por X, Y e Z, que representam respectivamente uma rotação de 180° do qubit na esfera de Bloch sobre os eixos x, y e z.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Figura 1.11: Representação matricial das portas X, Y e Z.

As matrizes de Pauli servem como base dos *operadores de rotação*, que são gerados a partir da exponenciação das matrizes. Os operadores são da forma $R_{\hat{n}}(\theta)$, onde $\hat{n} \in \{x, y, z\}$, e representam uma rotação de θ do qubit na esfera de Bloch sobre o eixo $\hat{n}$.

$$R_x(\theta) \equiv e^{-i\theta X/2} = \cos\frac{\theta}{2} I - i \sin\frac{\theta}{2} X = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$

$$R_y(\theta) \equiv e^{-i\theta Y/2} = \cos\frac{\theta}{2} I - i \sin\frac{\theta}{2} Y = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$

$$R_z(\theta) \equiv e^{-i\theta Z/2} = \cos\frac{\theta}{2} I - i \sin\frac{\theta}{2} Z = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{-i\frac{\theta}{2}} \end{bmatrix}$$

Esses operadores de rotação possuem um papel muito importante na computação quântica porque a partir deles é possível recriar qualquer porta lógica quântica de um único qubit. Para recriar o comportamento de um operador unitário U qualquer decompõe-se ele em três operações:

$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta)$$

Onde α, β, γ, δ, são valores reais e o fator $e^{i\alpha}$ representa uma mudança de fase global. Essa aproximação é chamada de *decomposição Z-Y*. Para

a construção de qualquer operador unitário não é preciso criar as portas quânticas referentes a qualquer valor de α, β, γ e δ, basta apenas alguns valores fixos específicos [16, p. 20]. Dessa forma é possível construir uma porta quântica arbitrária usando um conjunto finito de portas quânticas. Essa propriedade em conjunto com as portas lógicas de múltiplos qubits permitem a geração de um conjunto universal de portas lógicas quânticas.

Mas qual é o significado por trás de cada rotação? Algumas portas representam efeitos notáveis nos qubits. Por exemplo, a matriz de Pauli X é o equivalente quântico da operação NOT clássica, e muitas vezes é referenciada pelo mesmo nome, e funciona trocando as amplitudes α e β do qubit entre si. A porta Z mantém o valor de $|0\rangle$ enquanto inverte o sinal de $|1\rangle$. Outras portas notáveis são a porta S, ou porta de fase, a porta T e a porta de Hadamard. A porta de Hadamard serve para trocar a base computacional em que são efetuadas as operações, e quando aplicada nos qubits $|0\rangle$ ou $|1\rangle$ ela deixa o qubit "no meio do caminho" entre $|0\rangle$ e $|1\rangle$. As portas S e T tem funções importantes na implementação de alguns algoritmos quânticos.

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \qquad S = \begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix} \qquad T = \begin{bmatrix}1 & 0\\ 0 & e^{i\frac{\pi}{4}}\end{bmatrix}$$

Figura 1.12: Representação matricial das portas Hadamard, S e T.

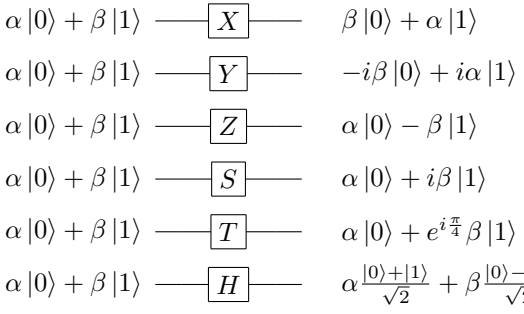


Figura 1.13: Portas lógicas quânticas de um qubit

1.3.2 Portas lógicas de múltiplos qubits

Ao programar é muito comum o uso de estruturas que chamamos de condicionais, representada pelos *if-else* inclusos em todas as linguagens de

programação. A capacidade de se realizar uma operação dado que as condições necessárias sejam encontradas é muito poderosa. É possível obter um efeito similar na computação quântica com o uso de portas controladas.

Uma porta controlada é uma operação que usa pelo menos dois qubits, separados em *controle* e *alvo*. Caso o qubit de controle esteja em $|1\rangle$ então a operação é efetuada no qubit alvo. Por exemplo, a porta controlled-not, ou *CNOT*, possui apenas um qubit de controle e um qubit alvo e caso o qubit de controle possua o valor $|1\rangle$ então o qubit alvo é negado. Existem também portas de múltiplos qubits que não são controladas. Por exemplo a porta SWAP, que troca dois qubits adjacentes de posição.

Ao se representar uma porta controlada o qubit de controle é demarcado por um ponto preto conectado à porta e os qubits alvo atravessam a porta a ser aplicada. Uma porta controlada pode ter qualquer número de qubits alvo e de controle. As portas SWAP e CNOT, e suas versões controladas, possuem uma notação alternativa. A porta SWAP pode ser escrita com dois $\times$ conectados e a porta NOT pode ser escrita usando apenas o símbolo $\oplus$.

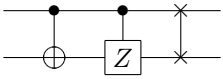


Figura 1.14: Representação em circuito das portas CNOT, Controlled-Z e SWAP

Algumas das portas de múltiplos qubits mais notáveis são a Toffoli, a Fredkin e a CNOT. A Toffoli também é conhecida como CCNOT ou Controlled-CNOT e a Fredkin é conhecida como a Controlled-SWAP. Com qualquer uma delas é possível recriar o comportamento de circuitos clássicos em um computador quântico. A porta CNOT serve de base para a criação de circuitos universais quânticos, e usando ela junto de outras portas de um único qubit é possível recriar o comportamento de qualquer porta de múltiplos qubits.

1.3.3 Universalidade em Circuitos Quânticos

Foi demonstrado que na computação clássica existem portas lógicas consideradas universais, isto é, a partir delas é possível reconstruir qualquer circuito lógico. Foi demonstrado também que com o uso da porta Toffoli é possível recriar qualquer circuito clássico usando computação quântica. Mas existe dentro da computação quântica alguma porta lógica universal?

A universalidade em circuitos pode ter duas formas, a estrita e a ampla. A universalidade estrita diz que com um conjunto finito de portas lógicas é possível implementar perfeitamente qualquer operação e a universalidade ampla diz que é possível aproximar qualquer circuito permitindo-se um erro ϵ arbitrário.

Na computação quântica, o conjunto de todas as portas quânticas de um qubit mais a porta CNOT formam um conjunto universal estrito. [7, p.118-124]. Entretanto esse é um conjunto absurdamente grande e não se sabe um método para se implementar todas essas portas de uma maneira tolerante a erros. Para contornar esse problema busca-se um conjunto universal no sentido amplo.

Como o conjunto de todas as portas lógicas quânticas é contínuo é impossível recriar perfeitamente um operador unitário qualquer usando um conjunto discreto de portas. Contudo é possível recriar de maneira aproximada um circuito quântico usando um conjunto finito de portas. Nesse caso considerando dois operadores unitários U e V, sendo U o operador real e V uma aproximação, o erro ao se usar V em vez de U é definido por:

$$E(U,V) \equiv \|(U-V)\,|\psi\rangle\|$$

Caso $E(U,V)$ seja pequeno então qualquer medida efetuada em $V\,|\psi\rangle$ resultará aproximadamente nos mesmos resultados de $U\,|\psi\rangle$. Isso se demonstra extremamente interessante pois assim pode-se focar em um conjunto pequeno de portas de forma que a computação seja tolerante a erros.

Existem alguns conjuntos que cumprem esses critérios. O conjunto padrão de portas lógicas quânticas é composto pelas portas CNOT, Hadamard, S e T e sua prova de universalidade pode ser feita de maneira simples. [16, p. 194–197] Outros exemplos de conjuntos são Hadamard, S, CNOT e Toffoli [16, p. 195] e X, Y, Z, H, T, S e CNOT [17, p. 89]. Nota-se que nem todos esse conjuntos são mínimos pois pode-se obter algumas dessas portas a partir de outras. (e.g. $T^2 = S$, $S^2 = Z$).

1.3.4 Identidades de Circuitos Quânticos

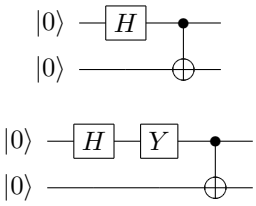
Interpretando um qubit como um ponto na esfera de Bloch onde ele se posicionaria após rotacioná-lo 90° no eixo y, 180° no eixo x, 180° no eixo z e novamente 90° no eixo y e 180° no eixo x? O resultado dessas rotações é equivalente a uma rotação em 180° no eixo x. Essas ações correspondem exatamente com a sequência de portas lógicas HZH e sua equivalência com a porta X.

Essa observação nos leva a uma importante ferramenta na construção de circuitos quânticos, que é encontrar sequências de portas lógicas com efeitos idênticos. Essas *identidades* permitem construir circuitos quânticos que usam menos recursos. A lista de todas as identidades de circuitos com até 3 portas lógicas é extensa [13] e a prova delas pode ser construída a partir de cálculo direto [17, p. 77–87].

Algumas identidades são triviais de serem encontradas. Por exemplo, as matrizes de Pauli e a porta de Hadamard são suas próprias inversas. Então sabe-se que $XX = YY = ZZ = HH = I$. Outras identidades podem ser deduzidas a partir de propriedades dos próprios operadores, como $XY = iZ$, $YZ = iX$, $XZ = iY$.

Identidades também são úteis para quando se está trabalhando com uma base computacional diferente. Foi dito anteriormente que a função da porta de Hadamard era levar o qubit para o "meio do caminho" entre $|0\rangle$ e $|1\rangle$. Matematicamente o que ocorre é que a porta leva o valor de $|0\rangle$ para $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ e $|1\rangle$ para $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, que respectivamente são representados por $|+\rangle$ e $|-\rangle$. Quando se trabalha com a base $|+\rangle$ e $|-\rangle$ muitas portas mudam de comportamento. Por exemplo, a porta X não é mais equivalente à operação NOT, pois nessa nova base uma rotação no eixo X não leva de um valor da base para o outro. Ao mudar de base as matrizes de Pauli seguem a seguinte relação: $HXH = Z$, $HYH = -Y$ e $HZH = X$. É possível aproveitar-se dessas identidades para remover parte do custo de um circuito relacionado a conversão de base em algum passo intermediário.

Além da equivalência entre portas lógicas também é possível pensar em equivalência entre estados quânticos. Por exemplo, os seguintes circuitos quânticos:



Apesar de possuírem uma sequência de portas diferentes resultam no mesmo estado quântico. Isso ocorre porque os estados finais de ambos os circuitos se diferem apenas por um fator global de $-i$. Para entender esse fenômeno pode-se voltar a abstração da esfera de Bloch. Rotacionar a esfera por um fator global pode ser interpretado como rodar a esfera inteira por um determinado ângulo, portanto o estado do qubit interno

não é alterado pois o ângulo relativo entre o vetor do qubit e os eixos da esfera não foi alterado já que todos sofreram a mesma mudança.

Capítulo 2

Síntese de Circuitos Quânticos

A medida que os computadores quânticos vão se tornando mais capazes e sofisticados, o interesse por algoritmos quânticos capazes de resolver instâncias reais de problemas aumenta. Embora em alguns algoritmos conhecidos o circuito tenha sido montado de forma manual [9] [22], a capacidade de se gerar automaticamente instâncias arbitrárias de computação é de grande interesse para aplicações práticas.

As ténicas iniciais de geração de circuitos quânticos baseavam-se na decomposição de portas lógicas de múltiplos qubits em portas de qubits únicos e CNOTs [6] [20]. Isso se deve principalmente pela dificuldade de se implementar diretamente portas de múltiplos qubits. Isso gerou também a medida de custo de um circuito quântico baseado em sua quantidade de portas CNOTs, visto que atualmente é a porta de maior custo e maior propensão a erro.

Existem muitas técnicas de síntese sendo estudadas, entretanto não existe um consenso da melhor técnica atual. É possível dividir as técnicas existentes em duas categorias, dependendo do seu nível de escalabilidade e do tipo de algoritmo usado.

1. **Métodos de Síntese Ótima:**

   Existem algumas técnicas focadas em gerar diretamente o circuito mínimo de um algoritmo. Entretanto, essas técnicas possuem pouca escalabilidade e não conseguem gerar resultados ótimos para instâncias maiores de problemas [25]. Essas limitações se devem principalmente pelo rápido acréscimo da complexidade de tempo e espaço

após se atingir um determinado limite.

2. **Métodos de Síntese Sub-ótima:**

   Técnicas de síntese dessa categoria produzem circuitos sub-ótimos, sendo necessário uma etapa de otimização após a síntese do circuito. A vantagem que esses métodos trazem é sua escalabilidade, sendo possível realizar instâncias arbitrárias de problemas. Técnicas como árvores binárias de decisão (BDD) [26], buscas em grafo [11], soma exclusiva de produtos (ESOP) [5], decomposição de matrizes [21], além de *and-inverter graphs* e *majority-inverter graphs* [4] são exemplos dessa classe.

2.1 Técnicas Atuais

Uma solução atual muito estudada para a síntese de circuitos quânticos é inicialmente construir um circuito booleano reversível e depois mapear sua solução para um circuito quântico equivalente. Isso é possível principalmente pelo comportamento naturalmente reversível de um circuito quântico.

A computação reversível se caracteriza pela não perda de informação e é possível apenas se o número de entradas e saídas de uma dada função F é igual e F é injetora [11]. Por exemplo, uma porta lógica clássica NOT é reversível e dado o valor da saída é possível recuperar a entrada do circuito aplicando-se a mesma operação. Ao contrário de uma porta lógica XOR clássica, onde dada duas entradas é gerado apenas uma saída. Nesse caso é dito que houve perda de informação no processo e não é possível recuperar as entradas dada apenas a saída do circuito.

Os métodos para geração de circuitos reversíveis são divididos em métodos ótimos e métodos heurísticos [18].

Métodos ótimos tem por objetivo encontrar o melhor circuito que implementa uma determinada operação. Uma vantagem desse método é que não é preciso de uma etapa de otimização sobre o circuito, entretanto, por oferecer a garantia de encontrar sempre a melhor solução, esse tipo de método não é escalável, sendo muitas das técnicas existentes possíveis apenas para circuitos de 3 ou 4 qubits [25].

Para mostrar a velocidade com que o tempo e o espaço necessário para esses métodos cresce, ao construir um circuito reversível de n qubits, onde sua implementação precisa de h portas de uma biblioteca L, um método enumerativo pode abrir até h caminhos em cada porta de L. Por exemplo,

assumindo que existem apenas portas Toffoli em uma biblioteca, uma busca exaustiva examina $(n \times 2^{n-1})^h$ portas para encontrar um circuito ótimo [18].

Métodos heurísticos usam heurísticas para tentar formar o melhor circuito possível e não podem garantir a otimalidade de sua solução. O maior ponto positivo desses métodos é sua escalabilidade, sendo possível trabalhar com instâncias arbitrárias de computação. Não existe um padrão de solução para essa classe de métodos e atualmente muitos grupos de heurísticas já foram usados. Alguns exemplos de heurísticas possíveis são:

- **Métodos baseados em transformação:** Esses métodos iterativamente selecionam uma porta de modo a fazer a tabela verdade de sua função mais similar com a função identidade. Esses métodos são majoritariamente eficientes para funções cujas saídas seguem um padrão regular.

- **Métodos baseados em busca:** Esses métodos atravessam uma árvore de busca para encontrar um circuito razoavelmente bom. A eficiência desses métodos depende muito do número de linhas e portas lógicas do circuito final.

- **Métodos baseados em ciclo:** Esses métodos decompõem uma dada função em um conjunto de ciclos disjuntos e sintetizam individualmente cada ciclo. Comparado aos outros métodos, esses métodos são majoritariamente eficientes para funções sem padrões regulares e funções reversíveis que deixam muitas combinações de entrada intactas.

- **Métodos baseados em BDD:** Esses métodos usam diagramas binários de decisão para aprimorar o partilhamento entre os controles de um circuito reversível. Essas técnicas escalam melhor do que as outras. Entretanto elas requerem um número maior de qubits auxiliares, o que é um recurso ainda escasso nos computadores quânticos atuais.

Muitas outras heurísticas podem usar métodos diferentes que não se encaixam nas classes listadas acima. Algumas reusam algoritmos desenvolvidos para síntese de circuitos lógicos convencionais. Outras usam uma sequência de portas ESOP junto de qubits auxiliares. Outras ainda usam teoria dos grupos para sintetizar circuitos reversíveis. Independente da

heurística selecionada essas técnicas tem demonstrado potencial para a síntese de circuitos quânticos.

Mesmo com o amplo estudo de técnicas voltadas para circuitos reversíveis, existem técnicas focadas em gerar diretamente o circuito quântico. Entre elas chamam atenção a programação genética [19] e *projective simulation* [14]. A programação genética se mostra interessante por sua possibilidade de gerar soluções melhores do que outros métodos, visto que seu processo de síntese envolve escolher circuitos que se adequem melhor a uma dada métrica de otimalidade. O modelo de *projective simulation* nunca foi usado para síntese de circuitos quânticos, entretanto ele já demonstrou resultados na criação de novos experimentos físicos [15] onde foi apontado seu pontencial para a síntese de circuitos quânticos.

2.2 Projective Simulation

Projective Simulation (PS) [14] é um modelo de aprendizado baseado no processamento estocástico de uma memória episódica que pode ser naturalmente aplicado em instâncias de problemas de aprendizado por reforço.

O ponto central desse modelo é um tipo específico de memória denotada por *episodic and compositional memory* (ECM). A ECM é representada por um grafo dirigido e com peso, onde cada nodo do grafo é referenciado como um clipe. Os clipes são as unidades básicas de memória e correspondem a experiências episódicas curtas. Os clipes da memória representam lembranças de percepções (*percepts*), ações ou uma combinação de ambos. Um clipe pode ser estimulado a partir de uma percepção do ambiente, e esse estímulo é então passado para um clipe adjacente com uma probabilidade relacionada ao peso da aresta conectando os dois clipes. Dessa forma cada percepção vinda do ambiente inicia um passeio aleatório pela memória. O passeio aleatório termina quando um clipe representando uma ação é atingido e gera uma ação real correspondente do agente para o ambiente.

O aprendizado em PS é alcançado a partir de modificações na ECM, tanto adicionando ou deletando clipes quanto alterando o peso de suas arestas. No início da simulação o agente se encontra em um estado de *tabula rasa*, i.e. não possui nenhuma preferência para um determinado tipo de ação. A medida que o agente é recompensado pelo ambiente a sua ECM é alterada de acordo com um conjunto pré-estabelecido de regras. As alterações na memória do agente podem resultar em uma nova tomada de decisão, recebendo então uma nova recompensa do ambiente.

O agente de PS é uma entidade situada em um ambiente parcialmente desconhecido, que recebe percepções através de sensores e pode realizar diferentes ações. As ações do agente são recompensadas pelo ambiente, o que causa uma mudança na estrutura interna de sua memória. O agente de PS não possui um modelo de ambiente que prevê o próximo estado ou a recompensa do ambiente, sendo assim considerado *model free*. O agente pode ser descrito através da probabilidade $P^{(t)}(a|s)$ de realizar uma ação a dada a percepção s. No entanto, uma descrição completa do agente conecta a probabilidade $P^{(t)}(a|s)$ com o estado interno de sua memória no tempo t, e especifica como a memória é modificada a medida que o agente interage com o ambiente.

2.2.1 Formalismo

- **Episodic and Compositional Memory (ECM):** é o elemento central do PS. É definida por um grafo direcionado e com pesos. Cada nodo do grafo é chamado de clipe.

- **Clipe:** Representa fragmentos de uma experiência episódica, que são definidas por L-tuplas $c = (c_1, c_2, ..., c_L)$. Cada c_i é a representação de uma percepção (S), ou uma ação (A).

- **Percepções (*Percepts*):** são definidas como N-tuplas $s = (s_1, s_2, ..., s_N) \in S \equiv S_1 \times S_2 \times ... \times S_N$, $s_i \in 1, ..., \|S\|$, onde o número de percepções possíveis é dado por $S \equiv \|S\| \equiv \|S_1\| ... \|S_N\|$.

- **Ações:** são definidas como M-tuplas $a = (a_1, a_2, ..., a_M) \in A \equiv A_1 \times A_2 \times ... \times A_M$, $a_i \in 1, ..., \|A\|$, onde o número de ações possíveis é dado por $A \equiv \|A\| \equiv \|A_1\| ... \|A_M\|$.

- **Arestas:** Cada aresta conectando o clipe c_i ao clipe c_j, possui o peso dinâmico $h^{(t)}(c_i, c_j)$, que se altera ao longo do tempo e é denotado como o valor-h da aresta.

- **Hopping Probability:** A probabilidade de um estímulo pular de um clipe c_i para um clipe c_j é dada por $p^{(t)}(c_i, c_j) = \frac{h^{(t)}(c_i, c_j)}{\sum_k h^{(t)}(c_i, c_k)}$ que representa a soma de todos os pesos dos clipes c_k conectados a c_i.

- **Emotion Tag:** Essas tags podem ser alocadas nas arestas que conectam percepções e ações para indicar se a transição correspondente

foi recompensada na última vez que ela foi escolhida. As tags podem ser usadas para memorizar a recompensa mais recente em uma dada transição, dessa forma permitindo a detecção de mudanças a curto prazo no ambiente.

- **Reflexão:** É o mecanismo que explora as emotion tags. Antes que uma ação a seja atingida a partir de um estímulo de um clipe de percepção s, a tag $e(s, a)$ é verificada. Se for positiva a ação é realizada, se for negativa o passeio aleatório é feito novamente. Isso permite o agente "reconsiderar" suas escolhas. O número máximo de passeios aleatórios por decisão é limitado por um parâmetro R que define o "tempo de reflexão". Por padrão $R = 1$, que significa sem tempo de reflexão.

- **Interface:** A interface entre o agente PS e o ambiente externo é feita através de sensores e atuadores e suas conexões com a memória. Uma percepção externa estimula um clipe-percepção c de acordo com a função de probabilidade $\mathcal{I}(c|s)$. Similarmente um clipe-ação é conectado para realizar uma ação de acordo com a função de probabilidade $\mathcal{O}(a|c)$. A função de conexão relaciona o passeio aleatório da memória, descrito pelas probabilidades $p^{(t)}(c_j|c_i)$, com o comportamento externo do agente, descrito por $P^{(t)}(a|s)$.

- **Parâmetro de Amortecimento:** O modelo de PS admite um parâmetro opcional γ chamado de parâmetro de amortecimento. Esse parâmetro pode ser interpretado como um nível de "esquecimento" do agente ao longo do tempo. Essa medida permite o enfraquecimento das conexões entre os clipes, tornando possível o agente se adaptar a ambientes mutáveis.

O processo por trás do modelo PS é estocástico. Cada passo t começa com uma percepção vinda do ambiente e estimulando um clipe de memória c_i dentro da memória de acordo com a função $\mathcal{I}(c|s)$. Depois, o estímulo pula do clipe c_i para um de seus clipes adjacentes, c_j, com a probabilidade $p^{(t)}(c_j|c_i)$. Esse processo continua em um passeio aleatório, permitindo que o estímulo se propague através da rede de clipes. O processo de pulo então chega ao fim quando um clipe de ação é alcançado e gera uma ação equivalente no "mundo-real" de acordo com a função $\mathcal{O}(a|c)$.

Quando uma ação é realizada, o agente recebe uma recompensa $\lambda \geq 0$ do ambiente. Como resultado, o valor-h de todas as arestas são atualizados de acordo com as seguintes regras:

(i) Os valores-h de todas as arestas ativadas, i.e. as arestas que foram atravessadas durante o último passeio aleatório, são atualizadas de acordo com a regra:

$$h^{(t+1)}(c_i, c_j) = h^{(t)}(c_i, c_j) - \gamma(h^{(t)}(c_i, c_j) - 1) + \lambda$$

onde t é o passo atual, γ $(0 \leq \gamma \leq 1)$ é o parâmetro de amortecimento e λ é a recompensa dada pelo ambiente.

(ii) O valor-h de todas as outras arestas da rede são amortecidas, como descrito pela regra:

$$h^{(t+1)}(c_i, c_j) = h^{(t)}(c_i, c_j) - \gamma(h^{(t)}(c_i, c_j) - 1)$$

Com essas atualizações, a ECM conclui um único passo de tempo.

A cada passo todos os valores-h são reduzidos por um fator $\gamma(h-1)$ e apenas as arestas que foram atravessadas durante a etapa anterior recebem a recompensa λ

Quando o agente recebe uma recompensa positiva, as arestas que foram atravessadas duranto o passeio aleatório que levou até a ação correta são fortalecidas, fazendo com que a probabilidade que elas sejam selecionadas novamente no futuro aumente. Por outro lado, se uma ação errada é tomada e nenhuma recompensa é obtida, i.e. $\lambda = 0$, todas as arestas são apenas amortecidas, incluindo aquelas que foram selecionadas durante o último passeio, diminuindo sua probabilidade de serem escolhidas novamente no futuro.

2.2.2 Correlação Temporal

Em alguns problemas de aprendizado por reforço a recompensa que o agente recebe pode não estar relacionada apenas a sua última ação realizada no presente, mas também nas ações feitas anteriormentes. Esse tipo de caso é referido como correlação temporal.

Para ser permitido o uso de correlação temporal no modelo de PS é definido um novo parâmetro na função de estímulo. Esse parâmetro é referido como *edge glow* e ele representa o nível do estímulo atual da aresta. Por exemplo, toda vez que uma aresta é escolhida seu *edge glow* assume o valor máximo, e para cada passo em que ela não é escolhida novamente o *glow* vai decaindo. Dessa forma uma aresta que não foi escolhida durante o último passeio aleatório na memória ainda poderá ser (parcialmente) recompensada. Esse efeito é referido como *afterglow*.

Formalmente o *afterglow* é implementado acrescentando um novo parâmetro g chamado de *glow*, o qual é relacionado com as arestas da ECM. Inicialmente $g = 0$ para todas as arestas. Então, uma vez que uma aresta é visitada durante o passeio aleatório seu parâmetro *glow* recebe o valor $g = 1$. A cada passo subsequente g decai até o valor 0 de acordo com a regra:

$$g^{(t+1)} = g^{(t)} - \eta g^{(t)}, 0 \leq \eta \leq 1$$

E a função de recompensa é alterada para:

$$h^{(t+1)}(c_1, c_2) = h^{(t)}(c_1, c_2) - \gamma(h^{(t)}(c_1, c_2) - 1) + \lambda g^{(t)}(c_1, c_2)$$

onde $g^{(t)}(c_1, c_2)$ representa o valor g da aresta conectando os clipes c_1 e c_2 no tempo t. Como antes, apenas as arestas estimuladas são fortalecidas quando a recompensa λ é entregue. A diferença agora é que as arestas podem estar parcialmente excitadas, permitindo recompensas parciais.

2.2.3 Memória Associativa

Dentro da *Projective Simulation* a ideia de uma memória associativa é definida pela introdução de novas arestas entre clipes-percepção que são considerados "similares". Esse é um processo dinâmico, onde as arestas são criadas durante o processo de aprendizado, a cada passo do tempo.

Dois clipes são considerados similares se eles se diferem em exatamente um único componente. De forma a evitar "associação prolífica", i.e. a situação onde arestas associativas são criadas entre clipes que compartilham propriedades irrelevantes, o agente é providenciado com uma "máscara de similaridade", que indica para cada componente dentro do espaço de percepção se ele é uma propriedade relevante para a associação ou não.

É possível aumentar a performance da memória associativa introduzindo uma modificação simples na regra de atualização da rede ECM, denotada por *clip glow*.

A ideia por trás do *clip-glow* é em vez de fortalecer as arestas que foram visitadas durante o último passeio aleatório, fortalecer as arestas que conectam clipes estimulados. Isso é atingido definindo um parâmetro *glow* $g \geq 0$ para cada clipe, em vez das arestas. Ao iniciar o parâmetro g é igual a zero, e toda vez que o clipe é encontrado seu parâmetro g é modificado para $g = 1$. A regra de atualização correspondente é dada por:

$$h^{(t+1)}(c_1, c_2) = h^{(t)}(c_1, c_2) - \gamma(h^{(t)}(c_1, c_2) - 1) + \lambda g^{(t)}(c_1) g^{(t)}(c_2)$$

onde $g^{(t)}(c_1)$ e $g^{(t)}(c_2)$ são os parâmetros *glow* referentes aos clipes c_1 e c_2, respectivamente. Isso significa que uma aresta só é recompensada se e somente se ela conecta dois clipes estimulados, i.e. clipes que possuem o parâmetro g positivo.

2.2.4 Composição

O aspecto composicional da ECM é um processo dinâmico que permite mudanças estruturais nas suas conexões internas. Em particular, ele permite a criação espontânea de novos clipes dentro da ECM, a partir da combinação ou variação de clipes existentes. Os novos clipes podem representar episódios fictícios que nunca foram experimentados antes, dessa forma extendendo a variedade de eventos concebíveis e ações que existem dentro da ECM. Como resultado a memória está menos ligada ao passado real do agente. Isso efetivamente permite que o agente gere opções alternativas àquelas que ele havia encontrado anteriormente, tornando-o mais capaz e flexível.

Existem muitas possibilidades para a junção e variação de clipes. Em [14] é definido um mecanismo de composição para clipes de ação com M dimensões.

- Dois clipes de ação $c_a = (a_1, a_2, ..., a_M)$ e $c_b = (b_1, b_2, ..., b_M)$ são compostos em um novo clipe de ação se e somente se: (a) ambas as ações correspondentes foram suficientemente recompensadas através da mesma percepção; e (b) os clipes de ação c_a e c_b se diferenciam exatamente em dois componentes.

- Quando os clipes de ação c_a e c_b se diferenciam em seus i-ésimo e j-ésimo componentes, a composição resulta em dois clipes compostos: $c_1^{new} = (a_1, a_2, ..., b_i, ..., a_j, ..., a_M)$ e $c_2^{new} = (a_1, a_2, ..., a_i, ..., b_j, ..., a_M)$.

- Um novo clipe é criado apenas se ele ainda não existe dentro da ECM.

- Os novos clipes de ação são conectados ao clipe de percepção correspondente c_0 com valor-h dado pela soma dos valores-h dos clipes de ação originais. $h(c_0, c_1^{new}) = h(c_0, c_2^{new}) = h(c_0, c_a) + h(c_0, c_b)$. Além

disso, os novos clipes de ação serão conectados a todos os outros clipes de percepção com valor-h inicial de 1.

Capítulo 3

Proposta

O objetivo geral deste trabalho é desenvolver um sintetizador de circuitos quânticos. Após analisar as técnicas levantadas no capítulo anterior foi escolhido o modelo de *Projective Simulation* para se gerar os circuitos. Essa técnica foi escolhida principalmente por ser uma técnica recente e ainda pouco explorada em outros trabalhos, e também por já ter mostrado resultados em outros campos [15].

O modelo de PS, conforme apresentado anteriormente, possui um agente, definido principalmente por sua *Episodic Compositional Memory* (ECM), que interage com um ambiente parcialmente desconhecido. O agente recebe percepções e retorna ações a serem feitas. As ações são recompensadas pelo ambiente e, a partir dessas recompensas, o agente vai alterando sua ECM de forma a reforçar as ações que foram bem-sucedidas.

O sintetizador implementado possui uma estrutura simples, não realizando as funções de reflexão, composição de ações e nem uma estrutura de memória associativa, embora sejam permitidas pelo modelo. Essas decisões foram tomadas principalmente pela falta de tempo no desenvolvimento do trabalho, já que, se baseando em trabalhos anteriores [15], seria possível obter resultados melhores usando composição de ações durante o aprendizado do agente.

A implementação foi feita usando a linguagem de programação Python e possui três partes principais, *PS_agent.py*, que representa o agente que irá aprender a sintetizar um circuito quântico, *quantum_circuit.py*, que representa o ambiente com o qual o agente irá interagir e *simulation.py*, que é um arquivo auxiliar que junta os dados obtidos em cada iteração e

monta os gráficos para análise. Existe também o arquivo auxiliar *run.py* que serve para instanciar todas as classes e rodar a simulação.

3.1 PS_agent.py

O arquivo PS_agent.py descreve duas classes, a PS_Agent e a ECM. A ECM descreve a Episodic Compositional Memory do agente e a PS_agent descreve o agente do modelo de *Projective Simulation*.

3.1.1 PS_agent

O PS_agent possui três métodos, o __init__(), o act() e o learn(). O método __init__() inicia os valores internos do agente, nesse caso a lista de ações que ele pode tomar, a lista de percepções iniciais, os parâmetros γ e η e por fim a própria ECM.

O método act() representa a tomada de decisão do agente. A partir de uma percepção passada como parâmetro o agente cria um nodo dentro da ECM para guardar aquela percepção, caso ela já não esteja na memória. O agente então inicia o passeio aleatório pela ECM e retorna uma ação a ser tomada por ele, escolhida de acordo com o processo interno de sua memória.

O método learn() representa o aprendizado do agente e chama a função interna da ECM que atualiza os valores-h das arestas de acordo com a recompensa recebida do ambiente pelo agente. Esse método também tem a função de limpar a ECM ao final de cada experimento que não foi bem sucedido. Como a cada construção de um novo circuito o agente cria um novo nodo para cada percepção nova recebida, para prevenir o uso excessivo de memória todos os nodos criados durante a geração de um circuito que não foi recompensado são deletados. Esse processo pode ser visto como "limpar a mesa de experimentos".

3.1.2 ECM

A classe ECM representa a estrutura da *Episodic and Compositional Memory* do agente. Sua estrutura principal é representado por um objeto Graph implementado pelo módulo de Python *graph-tool*. Seus métodos principais são o random_walk() e o update(), que são chamados pelo PS_-Agent para, respectivamente, iniciar um passeio aleatório pela memória e para atualizar os valores-h das arestas.

3.2 quantum_circuit.py

O quantum_circuit.py descreve os ambientes com os quais os agentes irão interagir. A classe QuantumCircuitEnv2Qubit representa um circuito com apenas 2 qubits. O ambiente possui um espaço de ações onde estão listadas todas as portas lógicas possíveis seguindo o padrão de nome "Pn" sendo "P" o nome da porta e "n" o número do qubit em que a porta será aplicada. Por exemplo, a ação H1 representa a aplicação de uma porta de hadamard no qubit número 1. A única porta de dois qubits disponível é a CNOT e sua nomeclatura é "CNOTca", sendo c o número do qubit de controle e a o número do qubit alvo.

As portas lógicas disponíveis foram selecionadas baseando-se no ambiente Quantum Experience da IBM [1].

3.3 simulation.py

O simulation.py chama o agente e o ambiente e roda a simulação um determinado número de vezes, passado por parâmetro. Ao final de todos os episódios ele recolhe os dados referentes às recompensas, ao número de portas e a profundidade do circuito e gera os gráficos referentes a esses dados.

3.4 run.py

Um arquivo auxiliar que instancia o agente, o ambiente, a simulação e roda o programa.

Capítulo 4

Resultados

Foram realizadas no total 4 simulações, de 200 iterações cada, usando como parâmetros $\gamma = 0.1$ e $\eta = 0.1$ e aceitando como profundidade máxima do circuito 4 portas. Os parâmetros de amortecimento γ e η foram escolhidos aleatoriamente, entretanto, os valores ótimos para esses parâmetros podem ser aprendidos pelo modelo PS com um custo de tempo de aprendizado maior. O ambiente de simulação foi um Intel Core i7-8550U 1.8GHz e uma RAM com 8Gb de capacidade.

A função de recompensa escolhida inicialmente foi,

$$\lambda = 100 - \sum e_g * \frac{d_{min}}{d_i}$$

onde e_g é o erro de cada porta do circuito, d_{min} é a profundidade mínima entre os circuitos encontrados pelo agente e d_i é a profundidade do circuito sendo avaliado. Essa heurística foi escolhida com objetivo de fazer o agente priorizar circuitos que acumulassem a menor quantidade de erro, seja ele decorrente tanto das próprias portas quanto pelo tempo de decoerência dos qubits. Nos primeiros testes porém, era comum o agente encontrar uma única solução e ficar repetindo ela até o fim da simulação. Isso pode ter ocorrido por causa da diferença entre *exploration* e *exploitation* do agente. Com essa função de recompensa, a pontuação final era muito grande, o que causava um reforço muito acentuado na estrutura da ECM, o que diminuía a capacidade de exploração (*exploration*) do agente. Por causa disso, a função de recompensa final foi alterada para,

$$\lambda = 100 - \sum e_g * \frac{d_{min}}{d_i^2}$$

Foi decidido substituir d_i por d_i^2 na função para diminuir a taxa de exploração (*exploitation*) do agente. Os resultados obtidos com a nova função de recompensa foram muito mais diversos.

Cada simulação tinha como objetivo encontrar um circuito gerador para um estado de Bell específico. O circuito básico para se gerar um estado de Bell possui apenas 2 portas lógicas, uma Hadamard e uma CNOT. A partir desse circuito é possível gerar todos os estados de Bell apenas alterando as entradas. Os estados de Bell são estados emaranhados formados por dois qubits, e foram escolhidos para testar o sintetizador por sua relevância na área de Computação Quântica e pela simplicidade de seu circuito gerador.

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\beta_{00}\rangle$$
$$\frac{|01\rangle - |10\rangle}{\sqrt{2}} = |\beta_{01}\rangle$$
$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\beta_{10}\rangle$$
$$\frac{|01\rangle - |10\rangle}{\sqrt{2}} = |\beta_{11}\rangle$$

Figura 1.2: Estados de Bell (Repetida da página 7)

Pelo circuito alvo ser muito pequeno e bem conhecido a ideia de buscar circuitos alternativos para a gerações dos estados pode parecer estranha, mas o objetivo final do sintetizador seria gerar circuitos não triviais. Por causa disso foca-se na descoberta de muitas opções (*exploration*), que podem ser então otimizadas e assim encontrada a sua versão mínima, em vez de encontrar uma solução e repetí-la indefinidamente (*exploitation*).

A relação entre os qubits foi definida baseando-se na plataforma Quantum Experience da IBM [1]. Essa plataforma online dispõe de 5 qubits e dois *layouts* possíveis para seus qubits. Os experimentos foram realizados levando em consideração o *layout* IBM Q 5 Tenerife, por apresentar uma menor taxa de erros. Pelo circuito alvo precisar de 2 qubits foram consideradas apenas as relações entre os qubits 0 e 1 da plataforma. Por causa disso a única direção possível para a operação de CNOT com o controle sendo o qubit 1 e o alvo sendo o qubit 0. Por motivos de simplicidade foram consideradas apenas as portas X, Y, Z, H e CNOT_{10} dentre todas as opções da plataforma.

Para testar a capacidade do sintetizador as entradas foram mantidas fixas como $|00\rangle$. Assim para se gerar os estados de Bell diferentes de β_{00} seria necessária a aplicação de algumas portas extras para se alterar a base antes de se aplicar o circuito básico composto pela Hadamard e CNOT.

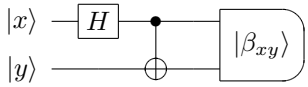


Figura 4.2: Circuito básico para criar um estado de Bell

Foram encontrados cerca de 10 circuitos geradores para cada estado de Bell. Como esperado, todos os circuitos apresentaram como subrotina o circuito básico formado pelas portas Hadamard e CNOT. Pela simplicidade do circuito alvo é possível ver claramente sequências de portas lógicas que não contribuem para o resultado final. Por exemplo, alguns circuitos apresentaram duas portas X seguidas. Um processo de otimização poderia facilmente reconhecer que como a porta X é sua própria inversa essa sequência de ações resulta na operação de identidade e logo poderia ser descartada do circuito sem nenhuma consequência. Esses resultados eram esperados já que o modelo de síntese não garantia a otimalidade de seus resultados. Nota-se que os circuitos que apresentaram esse comportamento não ótimo pontuaram muito menos do que os circuitos que apresentaram um número menor de portas.

Estado alvo	**Circuitos encontrados**
$\|\beta_{00}\rangle$	10
$\|\beta_{01}\rangle$	8
$\|\beta_{10}\rangle$	11
$\|\beta_{11}\rangle$	9

Todas as simulações encontraram a versão mínima do circuito capaz de gerar o estado de Bell alvo, demonstrado pelo primeiro circuito de cada amostra dos resultados a seguir.

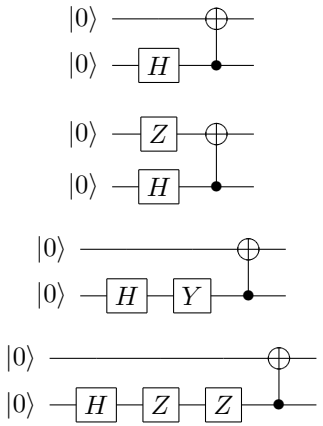


Figura 4.3: Circuitos encontrados pelo sintetizador para gerar o estado $|\beta_{00}\rangle$

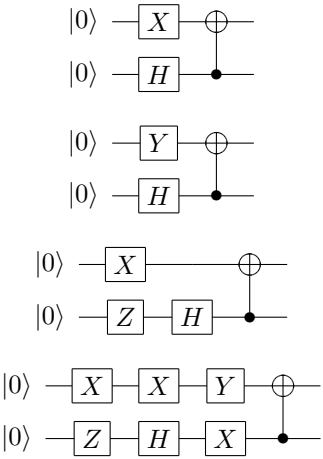


Figura 4.4: Circuitos encontrados pelo sintetizador para gerar o estado $|\beta_{01}\rangle$

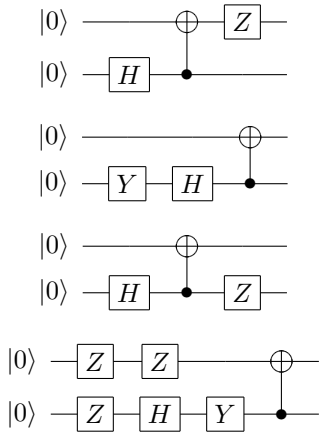


Figura 4.5: Circuitos encontrados pelo sintetizador para gerar o estado $|\beta_{10}\rangle$

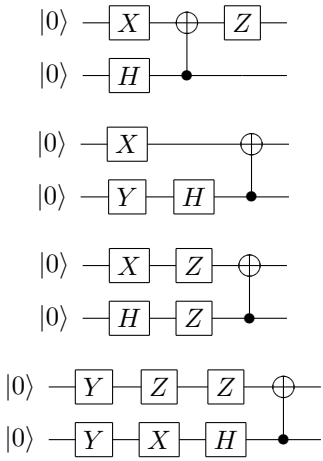


Figura 4.6: Circuitos encontrados pelo sintetizador para gerar o estado $|\beta_{11}\rangle$

Analisando o aprendizado do agente nota-se rapidamente que a pontuação máxima dos circuitos geradores do estado $|\beta_{00}\rangle$ e $|\beta_{01}\rangle$ foram superiores em relação aos demais circuitos. Isso ocorreu por causa da heurística escolhida na função de recompensa. Os estados $|\beta_{00}\rangle$ e $|\beta_{01}\rangle$ podiam ser gerados por um circuito de profundidade 2, já que a entrada $|00\rangle$ permitia isso, então a pontuação que eles eram capaz de atingir era maior do que a dos outros estados.

Observa-se que para os circuitos geradores dos estados de Bell diferentes de $|\beta_{00}\rangle$ o número de iterações bem-sucedidas da simulação foi consideravelmente menor. A causa desse comportamento foi possivelmente a pontuação média dos circuitos não ser muito alta, isso fez com que a possibilidade de explorar (*exploit*) essas soluções diminuísse, já que a quantidade de pontos perdidos pelo fator de amortecimento γ pode ter sido superior à pontuação recebida. Mesmo assim, esse fato não garantiu que o sintetizador descobrisse mais circuitos para $|\beta_{00}\rangle$, apenas indicou que ele explorou (*exploit*) mais as suas opções já encontradas.

Tendo em vista os resultados obtidos, a técnica de *Projective Simulation* demonstrou-se capaz de gerar todos os circuitos desejados. Entretanto, nada pode-se afirmar sobre sua escalabilidade, já que as simulações feitas foram com uma quantidade muito pequena de qubits. Para isso pode-se modificar o ambiente com o qual o agente interage, descrito em *quantum_circuit.py*, para descrever ambientes maiores e testar os limites do simulador.

Esse trabalho também pode ser expandido para aceitar mais tipos de portas lógicas, por exemplo as portas S e T suas respectivas inversas, além também de poder usar outras ferramentas disponíveis no modelo PS, como a composição de ações usada por [15] em sua pesquisa de geração de experimentos físicos. Com esses incrementos seria possível gerar um conjunto maior de circuitos.

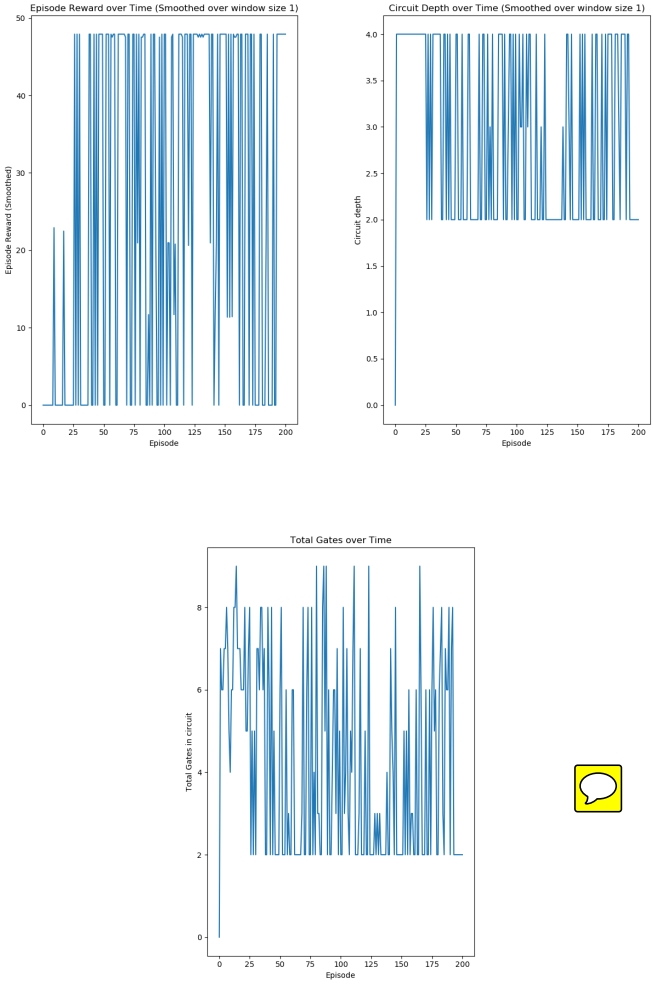


Figura 4.7: Resultados do sintetizador para o β_{00} com $\eta = 0.1$ e $\gamma = 0.1$

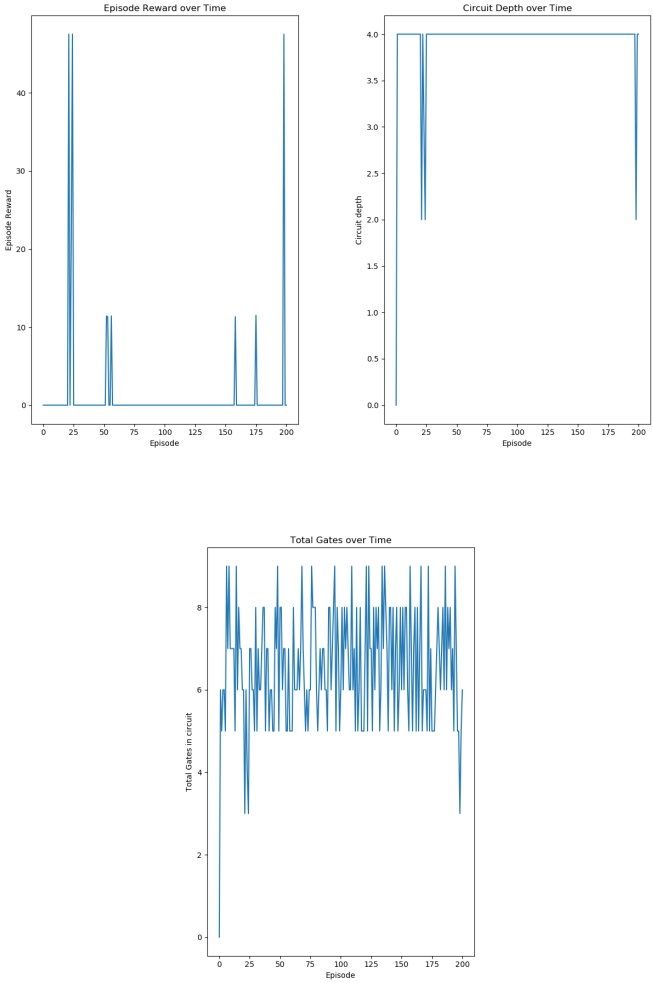


Figura 4.8: Resultados do sintetizador para o β_{01} com $\eta = 0.1$ e $\gamma = 0.1$

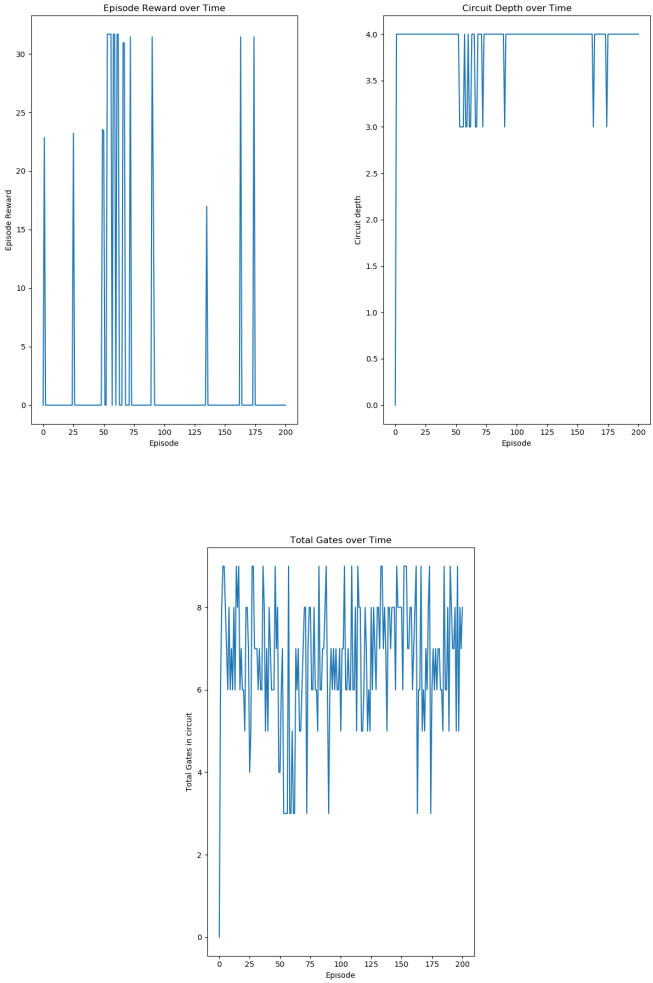


Figura 4.9: Resultados do sintetizador para o β_{10} com $\eta = 0.1$ e $\gamma = 0.1$

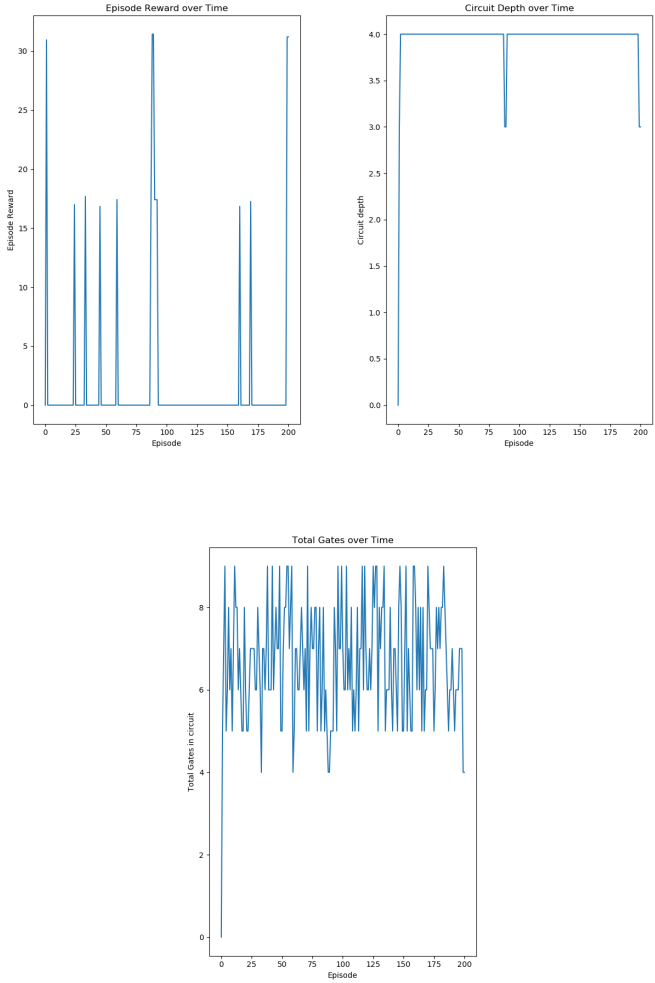


Figura 4.10: Resultados do sintetizador para o β_{11} com $\eta = 0.1$ e $\gamma = 0.1$

Conclusão

Considerações Finais

Esse trabalho trouxe uma análise do modelo Computação Quântica Circuital, fez um levantamento das técnicas atuais na área de Síntese de Circuitos Quânticos e implementou um sintetizador de circuitos quânticos usando a técnica de *Projective Simulation*. A escolha da técnica se deu por ela ser uma técnica recente e ainda pouco explorada na área, procurando-se então produzir uma análise de seu potencial. Os resultados se mostraram promissores, mas o escopo do trabalho foi muito pequeno para saber se é possível escalar essa técnica.

~~Trabalhos Futuros~~

Esse trabalho pode ser continuado expandindo a capacidade do sintetizador de forma a aceitar mais qubits e possuir mais possibilidades de portas lógicas. É possível também implementar a função de Composição de Ações, como prevista no modelo de PS, para tentar explorar mais opções de circuitos em problemas maiores. Outra maneira de expandir esse trabalho é desenvolver um otimizador a ser usado em conjunto com o sintetizador. Dessa forma seria possível garantir a otimalidade dos resultados obtidos pelo programa e usar diretamente os circuitos obtidos em problemas reais.

Espera-se que esse trabalho possa servir de inspiração para futuros trabalhos na área de Computação Quântica e principalmente em Síntese e Otimização de Circuitos Quânticos.

Bibliografia

[1] **IBM Quantum Experience** `https://quantumexperience.ng.bluemix.net` acesso em maio 2019.

[2] **Quantum Computing Report, Qubit Count** `https://quantumcomputingreport.com/scorecards/qubit-count/`, acesso em Abril 2019.

[3] **Quantum Computing Report, Qubit Quality Table II** `https://quantumcomputingreport.com/scorecards/qubit-quality/`, acesso em Abril 2019.

[4] C. Bandyopadhyay, R. Das, A. Chattopadhyay, and H. Rahaman. Design and synthesis of improved reversible circuits using aig- and mig-based graph data structures. *IET Computers Digital Techniques*, 13(1):38–48, 2019.

[5] C. Bandyopadhyay, S. Parekh, and H. Rahaman. Improved circuit synthesis approach for exclusive-sum-of-product-based reversible circuits. *IET Computers Digital Techniques*, 12(4):167–175, 2018.

[6] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, Nov 1995.

[7] Giuliano Benenti, Giulio Casati, and Giuliano Strini. *PRINCIPLES OF QUANTUM COMPUTATION AND INFORMATION Volume I: Basic Concepts.* World Scientific, 2004.

[8] Richard P. Feynman. Feynman and computation. chapter Simulating Physics with Computers, pages 133–153. Perseus Books, Cambridge, MA, USA, 1999.

[9] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219, 1996.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[11] D. K. Kole, H. Rahaman, D. K. Das, and B. B. Bhattacharya. Optimal reversible logic circuit synthesis based on a hybrid dfs-bfs technique. In *2010 International Symposium on Electronic System Design*, pages 208–212, Dec 2010.

[12] Mario Krenn, Mehul Malik, Robert Fickler, Radek Lapkiewicz, and Anton Zeilinger. Automated search for new quantum experiments. *Phys. Rev. Lett.*, 116:090405, Mar 2016.

[13] Chris Lomont. Quantum circuit identities. page 6, Jul 2003.

[14] Julian Mautner, Adi Makmal, Daniel Manzano, Markus Tiersch, and Hans J. Briegel. Projective simulation for classical learning agents: A comprehensive investigation. *New Generation Computing*, 33(1):69–114, Jan 2015.

[15] Alexey A. Melnikov, Hendrik Poulsen Nautrup, Mario Krenn, Vedran Dunjko, Markus Tiersch, Anton Zeilinger, and Hans J. Briegel. Active learning machine learns to create new quantum experiments. *Proceedings of the National Academy of Sciences*, 115(6):1221–1226, 2018.

[16] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, Cambridge, 2010.

[17] Giovani G. Pollachini. Computação quântica: Uma abordagem para estudantes de graduação em ciências exatas, 2018.

[18] Mehdi Saeedi and Igor L. Markov. Synthesis and optimization of reversible circuits—a survey. *ACM Comput. Surv.*, 45(2):21:1–21:34, March 2013.

[19] Moein Sarvaghad-Moghaddam, Philipp Niemann, and Rolf Drechsler. *Multi-objective Synthesis of Quantum Circuits Using Genetic Programming: 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, pages 220–227. 01 2018.

[20] M. Sedlák and M. Plesch. Towards optimization of quantum circuits. *Open Physics*, 6:128–134, Dec 2008.

[21] V. V. Shende, S. S. Bullock, and I. L. Markov. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1000–1010, June 2006.

[22] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.

[23] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, Jan 2016.

[24] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, Oct 2018.

[25] Marek Szyprowski and Paweł Kerntopf. Optimal 4-bit reversible mixed-polarity toffoli circuits. In Robert Glück and Tetsuo Yokoyama, editors, *Reversible Computation*, pages 138–151, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] R. Wille and R. Drechsler. Bdd-based synthesis of reversible logic for large functions. In *2009 46th ACM/IEEE Design Automation Conference*, pages 270–275, July 2009.

Apêndice A

Aprendizado por Reforço

A.1 Introdução

Aprendizado por Reforço, ou Reinforcement Learning, é um tipo de aprendizado de máquina onde se define um *agente* e um *ambiente*. O ambiente gera um estado para o agente que em retorno toma uma ação baseada nesse estado. O ambiente processa essa ação e gera um novo estado e uma recompensa para o agente. O objetivo do agente é descobrir a série de ações que resulte na maior recompensa total.

O aprendizado por reforço se difere dos outros aprendizados de máquina pelo seu método distinto de aprendizado. Ao contrário do aprendizado supervisionado, que recebe anteriormente um conjunto de dados com "a resposta correta", o aprendizado por reforço recebe ao final de uma iteração uma nota indicando sua performance. Esse método é ás vezes chamado de "aprendizado com um crítico" em oposição ao "aprendizado com um professor" do aprendizado supervisionado.

Pelo único indicador de desempenho que o agente possui ser a nota dada pelo crítico ao final de uma iteração, o agente se baseia fortemente em tentativa-e-erro para chegar até o seu objetivo.

A.2 Exploration x Exploitation

Suponha que existam três portas. Ao abrir a primeira o agente recebe 10 pontos, ao abrir a segunda o agente recebe 5 pontos e a terceira ainda não foi aberta. Qual é a melhor ação a se tomar nessa situação? Pode-se

escolher a primeira porta, porque ela atualmente é a porta que garante mais pontos, ou pode-se escolher a terceira porta, apostando que atrás dela existe uma recompensa maior do que a primeira.

Esse conflito característico do aprendizado por reforço é conhecido como *Exploration/Exploitation Dilemma*. *Exploration* se refere a explorar novas alternativas de forma a buscar uma solução ótima ainda não encontrada, no exemplo acima seria abrir a porta número 3. *Exploitation* se refere a tomar uma ação já conhecida que retorne o maior valor, no exemplo seria abrir a porta número 2. Um algoritmo guloso é um exemplo de algoritmo que foca totalmente em *Exploitation* e nada em *Exploration*.

Um bom algoritmo de aprendizado por reforço deve ser capaz de balancear ambas as decisões de forma a retornar o maior resultado possível. Um algoritmo que explore (*Explore*) demais pode acabar perdendo pontos por tomar muitas decisões subótimas enquanto confirma a melhor solução e um algoritmo que explore (*Exploit*) demais pode acabar preso em um ótimo local e não conseguir a pontuação máxima.

A.3 Aplicações

A.3.1 Inteligência Artificial

Em 2016 o campeão mundial de Go foi derrotado por uma inteligência artificial chamada de AlphaGo. Esse programa foi desenvolvido por um grupo de pesquisadores e usava redes neurais treinadas usando aprendizado supervisionado, usando como base de dados jogadas de jogadores profissionais, e aprendizado por reforço, jogando contra si mesma [23]. Essa vitória foi um marco para os pesquisadores de aprendizado de máquina porque foi a primeira vez que uma inteligência artificial derrotou um humano em um jogo tão complexo como Go.

Após a vitória, os desenvolvedores do AlphaGo começaram a trabalhar em seu sucessor, o AlphaGoZero [24]. Diferente do programa original, o AlphaGoZero foi treinado usando apenas Aprendizado por reforço e jogando contra si mesmo. Essa escolha de treinamento é notável porque diz que em nenhum momento do processo a IA recebeu alguma informação pronta de um humano. Como resultado a nova inteligência artificial conseguiu derrotar o AlphaGo original em 100 partidas sem perder nenhuma.

Um dos pontos mais impressionantes do experimento do AlphaGoZero foi a capacidade da inteligência artificial começar sem nenhuma informação prévia, ou *tabula rasa*, e sem nenhuma interferência humana ser capaz de

atingir níveis superiores aos humanos chegando a desenvolver jogadas próprias. Caso essa técnica seja aprimorada, seria possível transferir o agente de um jogo de Go para qualquer outro domínio, obtendo uma inteligência artifical genérica capaz de aprender qualquer habilidade em qualquer contexto.

A.3.2 Mecânica Quântica

Compreender a mecânica quântica tem se mostrado um dos maiores desafios para a ciência moderna. Devido a sua natureza contraintuitiva um dos problemas encontrados pelos físicos é desenvolver novos experimentos capazes de levar a novas descobertas. Pensando nisso, um grupo de físicos da Universidade de Vienna desenvolveu um algoritmo chamado Melvin, capaz de desenvolver novos experimentos quânticos, além de criar e manipular estados quânticos complexos [12]. O algoritmo Melvin tem se mostrado muito útil na criação de novos experimentos quânticos por causa de sua capacidade de "pensar fora da caixa", usando técnicas não familiares aos físicos mas perfeitamente concebíveis.

Mas qual é o limite do uso de máquinas para a ciência? Inspirados pelo sucesso do Melvin, o mesmo grupo de físicos buscou criar uma inteligência artificial para desenvolver novos experimentos quânticos [15]. O objetivo dessa inteligência artificial é contribuir para avançar novas pesquisas, o que vai além do poder do Melvin de identificar rapidamente soluções para problemas específicos.

Usando um framework de aprendizado por reforço, eles foram capazes de criar um modelo capaz de desenvolver experimentos complexos sem a necessidade de conhecimento prévio ou de intuição humana. O sistema também se mostrou capaz de descobrir técnicas de experimentação não-triviais, mostrando então que o uso de máquinas é capaz de oferecer avanços impressionantes em como experimentos são gerados.

A.4 Elementos do aprendizado por reforço

A.4.1 Agente

É a entidade que toma as decisões no modelo de aprendizado por reforço. Pode ser representado por um robô procurando a saída de um labirinto ou uma entidade abstrata que aprende um jogo, como xadrez ou go. O agente aprende a atingir seu objetivo ao interagir com o ambiente

por meio de seu conjunto de ações.

A.4.2 Ambiente

Abrange tudo o que não é o agente no modelo de aprendizado. Pode ser o tabuleiro e as peças do jogo a ser aprendido ou um percurso com obstáculos.

O ambiente aceita as ações enviadas pelo agente e em retorno produz uma recompensa e um novo estado.

A.4.3 Passo

Configura o tempo da escolha da ação e do retorno do estado/recompensa. Pode ser literalmente um passo do agente no percurso ou uma abstração de passagem de tempo. É usualmente denotado por t.

A.4.4 Estado

Representa a situação do ambiente na qual o agente baseia a sua ação. Num jogo de xadrez é representado pela configuração das peças no tabuleiro, em um percurso com obstáculos é representado pela posição do agente dentro do trajeto.

A.4.5 Recompensa

Um escalar retornado pelo ambiente quando o agente seleciona uma ação. Normalmente é representada por r_t indicando a recompensa r no passo t.

O valor da recompensa é determinado pelo desenvolvedor do problema e deve representar o objetivo a ser atingido pelo agente.

A.4.6 Ação

Pode ser parte de um conjunto discreto de ações, como por exemplo um robô que pode se deslocar em alguma das direções cardinais, ou parte de um conjunto contínuo, como por exemplo em um carro onde o volante pode assumir qualquer ângulo em um determinado intervalo.

A.4.7 Política

Um mapeamento de cada estado para a ação que deve se tomar naquele estado. Possui como notação π. Uma política pode ser determinística ou estocástica.

A.4.8 Função Valor

Mede o valor de um estado calculado pelo agente. Representa o acúmulo de recompensa a longo prazo partido de um estado s e seguindo uma política π.

Existem dois tipos de Função valor, a $V^{\pi}(s)$ e a $Q^{\pi}(s, a)$. A primeira representa o valor apenas do estado atingido seguindo a política π, a segunda, também chamada de Função valor-estado, representa o valor de um estado s ao se tomar a ação a seguindo uma política π.

Apêndice B

Código

B.1 PS_agent.py

```
1  import numpy as np
2  from graph_tool.all import *
3
4  class ECM():
5
6      def __init__(self, actions, percepts):
7          self.ECM = Graph()          # Network of clips that represents the
           ↪  memory
8          self.p_clips = []           # List of active percept clips
9          self.new_p_clips = []       # List of new percept clips that might
           ↪  be
10                                     # removed
11         self.a_clips = []           # List of action clips
12
13         # Initializing action properties
14         action = self.ECM.new_vertex_property("object")
15
16         # Initializing percept properties
17         percept = self.ECM.new_vertex_property("object")
18
19         # Creating percepts-clips
20         for p in percepts:
21             percept_clip = self.ECM.add_vertex()
22             percept[percept_clip] = p
```

```
23              self.p_clips.append(percept_clip)
24
25          # Creating action-clips
26          for a in actions:
27              action_clip = self.ECM.add_vertex()
28              action[action_clip] = a
29              self.a_clips.append(action_clip)
30
31          # Creating edges between percepts-clips and action-clips
32          for p in self.p_clips:
33              for a in self.a_clips:
34                  self.ECM.add_edge(p,a)
35
36          # Initializing edge properties h_value and glow
37          h_value = self.ECM.new_edge_property("double")
38          glow = self.ECM.new_edge_property("double")
39
40          # Setting initial values for h_value and glow
41          edges = self.ECM.get_edges()
42          for e in edges:
43              h_value[e] = 1
44              glow[e] = 0
45
46          # Adding properties to Graph
47          self.ECM.vertex_properties["action"] = action
48          self.ECM.vertex_properties["percept"] = percept
49
50          self.ECM.edge_properties["h_value"] = h_value
51          self.ECM.edge_properties["glow"] = glow
52
53      def random_walk(self, percept):
54          # Finding clip that matches percept
55          for v in self.ECM.vertices():
56              if np.array_equal(self.ECM.vp.percept[v], percept):
57                  hopping_clip = v
58                  break
59
60          # Random Walk until action-clip is found
61          while self.ECM.vp.action[hopping_clip] == None:
62              # Retrieving out edges from clip
63              out_edges_list = self.ECM.get_out_edges(hopping_clip)
64              out_edges = hopping_clip.out_edges()
65
66              # Setting probabilitie of hop
67              h_values = [self.ECM.ep.h_value[e] for e in out_edges]
```

```
68              sum_h_values = sum(h_values)
69              probabilities = [h/sum_h_values for h in h_values]
70
71              # Hopping
72              selected_edge = out_edges_list[np.random.choice(
73
                                                    ↪  len(out_edges_list),
74                                                  1,
75                                                  p=probabilities)]
76
77              # Setting glow parameter to 1
78              self.ECM.ep.glow[self.ECM.edge(
79                                      selected_edge[0][0],
80                                      selected_edge[0][1])] = 1
81
82              # Setting clip for next iteration
83              hopping_clip = self.ECM.vertex(selected_edge[0][1])
84
85          return self.ECM.vp.action[hopping_clip]
86
87      def update(self, reward, gamma, eta):
88          # Update h-value and glow of each edge
89          for e in self.ECM.edges():
90              self.ECM.ep.h_value[e] = max(1, self.ECM.ep.h_value[e] - gamma
                ↪  * (self.ECM.ep.h_value[e] - 1) + (reward *
                ↪  self.ECM.ep.glow[e]))
91              self.ECM.ep.glow[e] = max(0, self.ECM.ep.glow[e] - (eta *
                ↪  self.ECM.ep.glow[e]))
92
93  # TODO Implement composition function
94      def composition(self):
95          pass
96
97      def add_percept(self):
98          # If the last sequence of actions resulted in a reward, add the
99          # corresponding percept clips to the permanent list
100         self.p_clips.extend(self.new_p_clips)
101         self.new_p_clips = []
102
103     def clip_deletion_percept(self):
104         # If the last sequence of actions didn't result in a reward, delete
            ↪  the
105         # percepts created during the episode
106         for n_p in reversed(sorted(self.new_p_clips)):
107             self.ECM.remove_vertex(n_p)
```

```
108         self.update_clip_list()
109         self.new_p_clips = []
110 
111 #TODO Implement later after composition function
112     def clip_deletion_action(self):
113         pass
114 
115     def update_clip_list(self):
116         # Reevaluate clips indexes after a deletion action
117         self.a_clips = []
118         self.p_clips = []
119 
120         for v in self.ECM.vertices():
121             if self.ECM.vp.action[v] is None:
122                 self.p_clips.append(v)
123             if self.ECM.vp.percept[v] is None:
124                 self.a_clips.append(v)
125 
126 class PS_agent:
127 
128     def __init__(self, actions, percepts, eta, gamma):
129         self.actions = actions                          # List of possible
            ↪  actions
130         self.num_actions = len(actions)                 # Total number of
            ↪  actions
131         self.eta = eta                                  # Glow damping
            ↪  parameter
132         self.gamma = gamma                              # H-value damping
            ↪  parameter
133         self.memory = ECM(actions, percepts)            # Agent memory
134 
135     def act(self, percept):
136         # If percept is already in memory look for corresponding clip and
            ↪  start
137         # a random walk
138         for p in self.memory.p_clips:
139             if np.array_equal(self.memory.ECM.vp.percept[p], percept):
140                 return self.memory.random_walk(percept)
141 
142         # If percept is a new percept but was already added in a temporary
143         # percept-clip, find corresponding clip and start random-walk
144         for new_p in self.memory.new_p_clips:
145             if np.array_equal(self.memory.ECM.vp.percept[new_p], percept):
146                 return self.memory.random_walk(percept)
147         else:
```

```
148             # If percept not in memory, create corresponding clip and star
149             # random walk
150             new_percept = self.memory.ECM.add_vertex()
151             self.memory.ECM.vp.percept[new_percept] = percept
152             self.memory.new_p_clips.append(new_percept)
153             for a in self.memory.a_clips:
154                 e = self.memory.ECM.add_edge(new_percept,a)
155                 self.memory.ECM.ep.h_value[e] = 1
156             return self.memory.random_walk(percept)
157 
158     def learn(self, reward, done):
159         # Update h-values from edges
160         self.memory.update(reward, self.gamma, self.eta)
161 
162         # If last episode was succesful, add new percepts to the permanent
163         # list. Else, clear recently created percept clips
164         if done:
165             if reward > 0:
166                 self.memory.add_percept()
167             else:
168                 self.memory.clip_deletion_percept()
```

B.2 quantum_circuit.py

```
import numpy as np
from copy import deepcopy

### QuantumCircuitEnv environment

class QuantumCircuitEnv2Qubits:

    def __init__(self, max_circuit_depth, goal_state, tolerance):
        # State and action space
        # self.S = Column vector generated from tensor product of qubits
        # self.action_space = list of the name of the possible gates to use in
        #       the circuit. Every action follows the naming rule "Gn" where G
        #       is the name of the gate and n is the number of the qubit
        #       affected by the gate. CNOT gates follow the rule "CNOTct" where
        #       c is the number of the control qubit and t is the number of the
        #       target qubit.

        # Reward structure
        # self.trace_distance()

        # Transitions
        # self.operate()

        self.action_space = ['X0', 'Y0', 'Z0', 'H0', 'X1', 'Y1', 'Z1', 'H1', 'CNOT10']
        self.max_circuit_depth = max_circuit_depth # Maximum depth accepted by
                                                   # the circuit
        self.min_circuit_depth = max_circuit_depth # Minimum depth found while
                                                   # generating circuits
        self.goal_state = goal_state               # Quantum state to be
                                                   # generated by the circuit
        self.num_qubits = 2                        # Number of qubits in circuit
        self.tolerance = tolerance                 # Tolerance of trace distance
```

```
33
34          # Matrices representation of the possible gates
35          self.gate_matrices = {
36                  'X0':(np.matrix([[0,0,1,0],[0,0,0,1], [1,0,0,0],
                    ↪   [0,1,0,0]]), 0.77),
37                  'Y0':(np.matrix([[0,0,-1j,0],[0,0,0,-1j], [1j,0,0,0],
                    ↪   [0,1j,0,0]]), 0.77),
38                  'Z0':(np.matrix([[1,0,0,0],[0,1,0,0], [0,0,-1,0],
                    ↪   [0,0,0,-1]]), 0.77),
39                  'H0':(1/np.sqrt(2) * np.matrix([[1,0,1,0],[0,1,0,1],
                    ↪   [1,0,-1,0], [0,1,0,-1]]), 0.77),
40                  'X1':(np.matrix([[0,1,0,0],[1,0,0,0], [0,0,0,1],
                    ↪   [0,0,1,0]]), 1.46),
41                  'Y1':(np.matrix([[0,-1j,0,0],[-1j,0,0,0], [0,0,0,1j],
                    ↪   [0,0,1j,0]]), 1.46),
42                  'Z1':(np.matrix([[1,0,0,0],[0,-1,0,0], [0,0,1,0],
                    ↪   [0,0,0,-1]]), 1.46),
43                  'H1':(1/np.sqrt(2) * np.matrix([[1,1,0,0],[1,-1,0,0],
                    ↪   [0,0,1,1], [0,0,1,-1]]), 1.46),
44                  'CNOT10':(np.matrix([[1,0,0,0],[0,0,0,1], [0,0,1,0],
                    ↪   [0,1,0,0]]), 2.7),
45                  }
46
47          # Number of gates applied to each qubit
48          self.circuit_depths = np.zeros(self.num_qubits)
49          # Gates applied to each qubit
50          self.circuit_gates = [[] for q in range(self.num_qubits)]
51          # Sum of the gate errors in the circuit
52          self.sum_error = 0
53
54      # Reset the internal state of the circuit and return all qubits in the
55      # computational basis
56      def reset(self):
57          self.s = self.init_comp_basis()
58          self.circuit_depths = np.zeros(self.num_qubits)
59          self.circuit_gates = [[] for q in range(self.num_qubits)]
60          self.sum_error = 0
61          self.is_reset = True
62
63          return self.s
64
65      # Return corresponding matrix of given gate
66      def action2matrix(self, action):
67          return self.gate_matrices[action]
68
```

```
69      # Update circuit depth
70      def calculate_circuit_depth(self, a):
71          if a == 'X0' or a == 'Y0' or a == 'Z0' or a == 'H0':
72              self.circuit_depths[0] += 1
73              if max(self.circuit_depths) > self.max_circuit_depth:
74                  return self.max_circuit_depth
75              else:
76                  self.circuit_gates[0].append(a)
77                  return max(self.circuit_depths)
78          elif a == 'X1' or a == 'Y1' or a == 'Z1' or a == 'H1':
79              self.circuit_depths[1] += 1
80              if max(self.circuit_depths) > self.max_circuit_depth:
81                  return self.max_circuit_depth
82              else:
83                  self.circuit_gates[1].append(a)
84                  return max(self.circuit_depths)
85          elif a == 'CNOT10':
86              self.circuit_depths[1] = max(self.circuit_depths[0],
                ↪  self.circuit_depths[1])
87              self.circuit_depths[0] = max(self.circuit_depths[0],
                ↪  self.circuit_depths[1])
88              self.circuit_depths[1] += 1
89              self.circuit_depths[0] += 1
90              if max(self.circuit_depths) > self.max_circuit_depth:
91                  return self.max_circuit_depth
92              else:
93                  self.circuit_gates[1].append(a)
94                  self.circuit_gates[0].append(a)
95                  return max(self.circuit_depths)
96
97      # Calculate trace distance between current state and goal state
98      def trace_distance(self, s):
99          density_s = self.density_matrix(s)
100         density_goal = self.density_matrix(self.goal_state)
101         trace = sum(abs(np.linalg.eigvals(density_s - density_goal)))/2
102
103         if trace < self.tolerance:
104             return 100
105         else:
106             return 0
107
108     # Return density matrix of given qubit
109     def density_matrix(self,s):
110         new_s = deepcopy(s)
111         new_s.shape = (4,1)
```

```
112         return new_s*np.conj(new_s).T
113
114     # Multiply gate matrix and qubit vector
115     def operate(self, s, a):
116         return np.dot(s,a)
117
118     # Given an action, update internal state and return reward. If a final
119     # state is reached, reset environment
120     def step(self,action):
121         s_prev = self.s
122         a, e = self.action2matrix(action)
123         self.sum_error += e
124         self.s = self.operate(self.s, a)
125         reward = self.trace_distance(self.s)
126         depth = self.calculate_circuit_depth(action)
127         self.is_reset = False
128
129         # If current circuit was rewarded or maximum circuit depth reached,
130         # print result and reset environment
131         if reward > 0 or max(self.circuit_depths) > self.max_circuit_depth:
132             if self.min_circuit_depth > depth:
133                 self.min_circuit_depth = depth
134             output = open('output.out', 'a')
135             print("Gates:\n", file = output)
136             print("qubit 0: ", self.circuit_gates[0], file = output)
137             print("qubit 1: ", self.circuit_gates[1], file = output)
138             print("min circuit depth: ", self.min_circuit_depth, file =
                ↪  output)
139             if reward > 0:
140                 reward = (reward-self.sum_error) *
                    ↪  (self.min_circuit_depth/(depth*depth))
141                 print("Right circuit", reward, file = output)
142             print("\n", file = output)
143             output.close()
144
145             self.reset()
146
147         return (self.s, reward, self.is_reset, depth)
148
149     # Return the computational basis |00>
150     def init_comp_basis(self):
151         basis = np.array([1,0,0,0])
152         return basis
```

B.3 simulation.py

```
import numpy as np
import sys
import pandas as pd
from collections import namedtuple
from matplotlib import pyplot as plt
from matplotlib import pylab
import matplotlib.gridspec as gridspec

EpisodeStats = namedtuple("Stats",["episode_lengths", "episode_rewards", "episode_depths", "episode_running_variance"])

class Simulation:
    def __init__(self, env, agent):

        self.env = env
        self.agent = agent

        self.episode_length = np.array([0])
        self.episode_reward = np.array([0])

        self.fig = pylab.figure(figsize=(10, 5))
        gs = gridspec.GridSpec(2, 2)
        self.ax = pylab.subplot(gs[:, 0])
        self.ax.xaxis.set_visible(False)
        self.ax.yaxis.set_visible(False)

        self.ax1 = pylab.subplot(gs[0, 1])
        self.ax1.yaxis.set_label_position("right")
        self.ax1.set_ylabel('Length')
        self.ax1.set_xlim(0, max(10, len(self.episode_length)+1))
        self.ax1.set_ylim(0, 51)

        self.ax2 = pylab.subplot(gs[1, 1])
        self.ax2.set_xlabel('Episode')
        self.ax2.yaxis.set_label_position("right")
        self.ax2.set_ylabel('Reward')
        self.ax2.set_xlim(0, max(10, len(self.episode_reward)+1))
        self.ax2.set_ylim(0, 2)

        self.line, = self.ax1.plot(range(len(self.episode_length)),self.episode_length)
```

```
40          self.line2, =
            ↪  self.ax2.plot(range(len(self.episode_reward)),self.episode_reward)
41
42      def plot_episode_stats(self, stats, smoothing_window=10,
            ↪  hideplot=False):
43          # Plot the episode length over time
44          fig1 = plt.figure(figsize=(10,5))
45          plt.plot(stats.episode_lengths)
46          plt.xlabel("Episode")
47          plt.ylabel("Total Gates in circuit")
48          plt.title("Total Gates over Time")
49          if hideplot:
50              plt.close(fig1)
51          else:
52              plt.show(fig1)
53
54          #Plot the episode circuit depth over time
55          fig2 = plt.figure(figsize=(10,5))
56          plt.plot(stats.episode_depths)
57          plt.xlabel("Episode")
58          plt.ylabel("Circuit depth")
59          plt.title("Circuit Depth over Time")
60          if hideplot:
61              plt.close(fig2)
62          else:
63              plt.show(fig2)
64
65          # Plot the episode reward over time
66          fig3 = plt.figure(figsize=(10,5))
67          rewards_smoothed =
            ↪  pd.Series(stats.episode_rewards).rolling(smoothing_window,
            ↪  min_periods=smoothing_window).mean()
68          plt.plot(rewards_smoothed)
69          plt.xlabel("Episode")
70          plt.ylabel("Episode Reward")
71          plt.title("Episode Reward over Time")
72          if hideplot:
73              plt.close(fig3)
74          else:
75              plt.show(fig3)
76
77          return fig1, fig2, fig3
78
79      def run_ps(self, max_number_of_episodes=100, display_frequency=1):
80          circuit_depth = np.array([0])
```

```
81
82          # repeat for each episode
83          for episode_number in range(max_number_of_episodes):
84
85              # initialize state
86              percept = self.env.reset()
87
88              done = False # used to indicate terminal state
89              R = 0 # used to display accumulated rewards for an episode
90              t = 0 # used to display accumulated steps for an episode i.e
                 ↪  episode length
91
92              # repeat for each step of episode, until state is terminal
93              while not done:
94
95                  t += 1 # increase step counter - for display
96
97                  # Return action from PS Agent memory
98                  action = self.agent.act(percept)
99
100                 # take action, observe reward and next percept
101                 next_percept, reward, done, depth = self.env.step(action)
102
103                 # agent learn (ECM update)
104                 self.agent.learn(reward, done)
105
106                 # state <- next state
107                 percept = next_percept
108
109                 R += reward # accumulate reward - for display
110
111             self.episode_length = np.append(self.episode_length,t) # keep
                 ↪  episode length - for display
112             self.episode_reward = np.append(self.episode_reward,R) # keep
                 ↪  episode reward - for display
113             circuit_depth = np.append(circuit_depth, depth)        # keep
                 ↪  episode depth - for display
114
115         # Make Graphics and save them in file
116         self.fig.clf()
117         stats = EpisodeStats(
118             episode_lengths=self.episode_length,
119             episode_rewards=self.episode_reward,
120             episode_depths=circuit_depth,
121             episode_running_variance=np.zeros(max_number_of_episodes))
```

```
122         lenght_plot, depth_plot, reward_plot =
            ↪  self.plot_episode_stats(stats, display_frequency)
123         lenght_plot.savefig("total_gates.png")
124         depth_plot.savefig("depth.png")
125         reward_plot.savefig("reward.png")
```

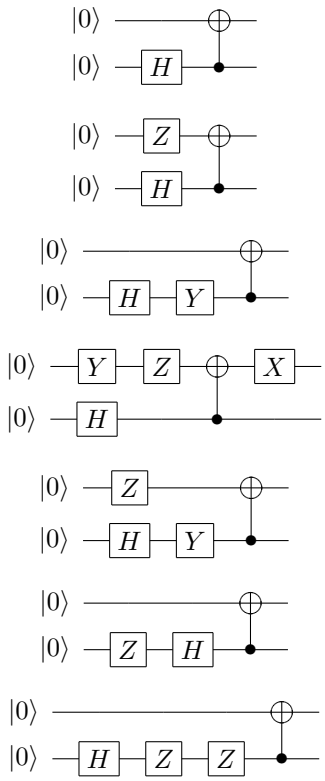
B.4 run.py

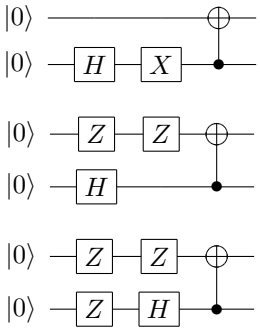
```
import numpy as np
import sys

if "../" not in sys.path:
    sys.path.append("../")

from PS_agent import PS_agent
from envs.quantum_circuit import QuantumCircuitEnv2Qubits
from lib.simulation import Simulation

interactive = False

# Computational basis |00>, |01>, |10>, |11>
zero  = np.array([1,0,0,0])
one   = np.array([0,0,1,0])
two   = np.array([0,1,0,0])
three = np.array([0,0,0,1])

# Bell State to be reached
goal_state = 1/np.sqrt(2) * (one-two)

# Enviroments instantiation
env = QuantumCircuitEnv2Qubits(4,goal_state, 1e-13)

# Agents instantiation
agent = PS_agent(env.action_space, [env.reset()], eta=0.1, gamma=0.1)

# Simulation instantiation
experiment = Simulation(env, agent)

# Run simulation 200 times
experiment.run_ps(200)
```

Apêndice C

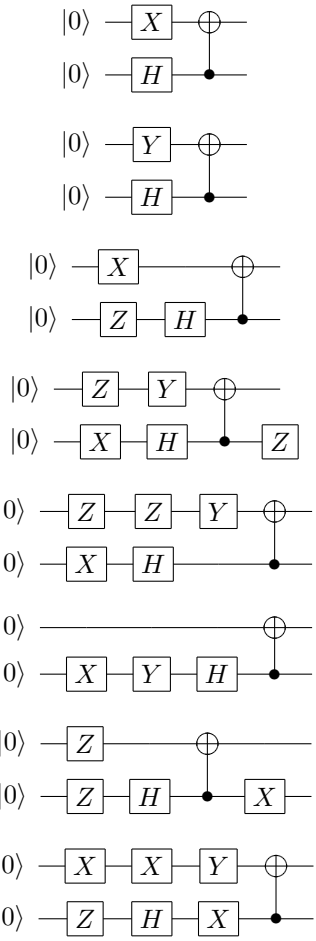
Resultados Completos

C.1 Estado de Bell $|\beta_{00}\rangle$

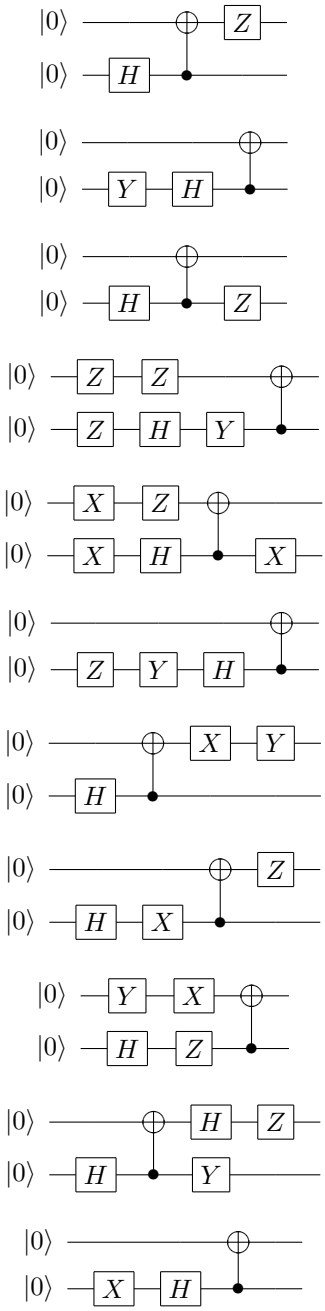




C.2 Estado de Bell $|\beta_{01}\rangle$



C.3 Estado de Bell $|\beta_{10}\rangle$



C.4 Estado de Bell $|\beta_{11}\rangle$