UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

IMPLEMENTAÇÃO DO ALGORITMO QUÂNTICO DA TELEPORTAÇÃO USANDO JAVA CLOSURES

TRABALHO DE GRADUAÇÃO

Bruno Crestani Calegaro

Santa Maria, RS, Brasil

2010

IMPLEMENTAÇÃO DO ALGORITMO QUÂNTICO DA TELEPORTAÇÃO USANDO JAVA CLOSURES

por

Bruno Crestani Calegaro

Trabalho de Graduação apresentado ao Curso de Ciência da Computação
da Universidade Federal de Santa Maria (UFSM, RS), como requisito
parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Orientador: Profª. Juliana Kaizer Vizzotto (UFSM)

Trabalho de Graduação N. 301

Santa Maria, RS, Brasil

2010

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

IMPLEMENTAÇÃO DO ALGORITMO QUÂNTICO DA TELEPORTAÇÃO USANDO JAVA CLOSURES

elaborado por
Bruno Crestani Calegaro

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Profª. Juliana Kaizer Vizzotto (UFSM)
(Presidente/Orientador)

Profª. Andrea Schwertner Charão (UFSM)

Prof. Giovani Rubert Librelotto(UFSM)

Santa Maria, 07 de Dezembro de 2010.

“Se, a princípio, a idéia não é absurda, então não há esperança para ela.”
— ALBERT EINSTEIN

AGRADECIMENTOS

Gostaria de agradecer primeiramente à minha família, em especial: aos meus pais, João Getúlio e Maria Idanir, que me apoiaram e deram todo o suporte necessário para atingir meus objetivos. Agradeço a confiança de vocês que acreditaram em mim e deixaram eu seguir este caminho com meus próprios pés. Aos meus irmãos, obrigado por fazerem parte da minha vida e serem uma ótima companhia, à Raisa por compartilhar interesses como nenhum outro amigo fez comigo, seja por anime ou por livros; e pro Vitor, por ser meu músico preferido e também por estar sempre me incentivando a ir mais longe na vida. À minha namorada Bruna, que me apoiou durante todo o período de realização da graduação, que me aturou nos momentos de estresse - bem constantes - e que comemorou comigo os momentos de alegria.

Agradeço aos meus eternos amigos, ao Marcus Garcia, por ser sempre uma pessoa alegre que contagia todos em volta. Ao João Gabriel Piccoli por ensinar sua filosifia de vida, que a vida não é quantas vezes se respira, mas sim quantas vezes se perde o fôlego. Ao Vinícius Alves, por mostrar que não se deve parar de sonhar e desistir pois as conquistas vem com muita batalha. Ao Rodrigo Tonin, por ser um grande companheiro na faculdade, no futebol, nas festas e até nos momentos de amargura, posso ter certeza que sempre posso contar com ele.

À minha orientadora Juliana K. Vizzotto, que me motivou e soube sempre dar boas dicas nos momentos em que foram necessárias. Agradeço pela dedicação, paciência e sobretudo pela amizade.

Aos meus colegas da UFSM, que compartilharam muitos momentos difíceis nos 4 anos que estive na UFSM e que me ajudaram a conseguir esta conquista. Ressalvo para as horas de estudo no LaCa antes das provas e também os grupo de estudos na casa do PL antes das matemáticas. Agradeço a todos, e em especial ao Cleber Pedrozo e Felipe Perini que além de colegas foram grandes companheiros nessa jornada.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

IMPLEMENTAÇÃO DO ALGORITMO QUÂNTICO DA TELEPORTAÇÃO USANDO JAVA CLOSURES
Autor: Bruno Crestani Calegaro
Orientador: Profª. Juliana Kaizer Vizzotto (UFSM)
Local e data da defesa: Santa Maria, 07 de Dezembro de 2010.

A computação quântica é uma tecnologia emergente com um grande potencial: enquanto computadores quânticos podem resolver certos problemas computacionais com um ganho exponencial de tempo em relação a sua execução em computadores clássicos, protocolos de criptografia baseados em chaves quânticas já estão sendo usados para fornecer um nível de segurança até então desconhecido. Pesquisadores da área de ciência da computação encontram o desafio de desenvolver novos algoritmos e protocolos quânticos, linguagens de programação quântica e seus modelos semânticos. É preciso investigar novos modelos semânticos para computação quântica, bem como novas soluções de desenvolvimento, análise e simulação de modelos quânticos. Nesse contexto, este trabalho teve como objetivo modelar os estados quânticos da mecânica quântica através da utilização de mônadas e então implementar o algoritmo da teleportação quântica usando tal modelo utilizando o recurso de definição de funções anônimas em Java, Java Closures. A simulação do algoritmo quântico da teleportação no modelo desenvolvido comprovou a eficiência de modelar estados quânticos usando mônadas. Espera-se, que o produto desenvolvido seja usado para implementar outros algoritmos e operações da computação quântica, e também , que sirva como base de estudo para alunos interessados na computação quântica.

Palavras-chave: Computação Quântica, Teleportação Quântica, Java Closures, Mônadas.

ABSTRACT

Trabalho de Graduação
Undergraduate Program in Computer Science
Universidade Federal de Santa Maria

IMPLEMENTATION OF THE ALGORITHM OF QUANTUM TELEPORTATION USING JAVA CLOSURES

Author: Bruno Crestani Calegaro
Advisor: Prof[a]. Juliana Kaizer Vizzotto (UFSM)

Quantum computing is an emerging technology with great potential: while quantum computers can solve certain computational problems with a exponentially gain of time in relation of the classical computers implementations, encryption protocols based on quantum keys are already being used to provide a new level of security until then unknown. Computer science researchers are challenged to develop new quantum algorithms, new quantum protocols, quantum programming languages and their semantic models. We need to investigate new semantic models for the quantum computation, as well as new solutions of development, analysis and quantum model simulations. Therefore, this paper aims to model the quantum states of quantum mechanics through the use of monads using the feature of definition of anonymous functions in Java, Java Closures. And then implement the algorithm of quantum teleportation with that model. The simulation of the quantum teleportation algorithm developed in the model, confirmed the efficiency of modeling quantum states using monads. It is expected that the developed product be used to implement other algorithms and operations of quantum computing, and also, serve as basis of study for students interested in quantum computing.

Keywords: Quantum Computation, Quantum Teleportation, Java Closures, Monads.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Cbit	Bit Clássico
Qbit	Bit Quântico
EPR	Einsten, Podolsky e Rosen
JDK	Java Development Kit
BGGA	Bracha Gafter Gosling Ahé

SUMÁRIO

1 INTRODUÇÃO

A computação quântica é baseada no fato que a concepção clássica do mundo é fundamentalmente diferente da proposta pela física quântica. Feynman (FEYNMAN, 1982) foi o primeiro a sugerir que processos quânticos não podiam ser eficientemente simulados em um computador clássico. Esta observação levou a proposta de um computador quântico como noção fundamental. Notavelmente, Shor mostrou que o problema da fatoração de números pode ser resolvido em tempo polinomial em um computador quântido hipotético (SHOR, 1994), enquanto o melhor algoritmo clássico conhecido precisa de tempo exponencial (POMERANCE, 1996). A definição desse algoritmo quântico representa um desafio para a área de criptografia de chave pública baseada em RSA (RIVEST; SHAMIR; ADELMAN, 1977) e em outros algoritmos. Entretanto, a física quântica também soluciona esse problema utilizando chaves quânticas seguras baseadas no teorema da não-clonagem, o qual enuncia que é impossível contruir uma cópia precisa de um estado quântico.

Tecnicamente, a computação quântica pode ser entendida como processamento de informação realizado fisicamente através de um sistema físico quântico de escala atômica (MERMIM, 2007). Pesquisadores da área de ciência da computação têm trabalhado atualmente no desenvolvimento de linguagens de programação quântica, bem como na definição de seus modelos semânticos para assim fundamentar o desenvolvimento, a análise, a modelagem e a simulação de algoritmos quânticos.

Em (VIZZOTTO; ALTENKIRCH; SABRY, 2006) foi apresentado um modelo semântico para estados quânticos puros utilizando-se o conceito de mônadas (MOGGI, 1989), conceito matemático (da área de teoria das categorias) utilizado para modelar elegantemente efeitos colaterais (estado, por exemplo) no contexto de linguagens funcionais puras. Uma mônada pode ser implementada em Java utilizando-se uma extensão que

suporta funções anônimas e closures(GAFTER, 2009).

Neste contexto, este trabalho compreende os seguintes objetivos: (i) implementar um modelo monádico de estados quânticos utilizando a extensão de Java, *Java Closures* e (ii) implementar o *algoritmo da teleportação quântica*(BENNETT et al., 1933) neste modelo de computação quântica em Java. Espera-se essencialmente que a implementação do algoritmo ajude estudantes na compreensão das características da computação quântica.

No restante deste texto, são apresentados os conceitos e teoremas utilizados no desenvolvimento deste trabalho. Na sequência, apresenta-se o modelo monâdico criado para representar os estados quânticos e a análise do comportamento do algoritmo da teleportação neste modelo.

2 CONCEITOS E TECNOLOGIAS

Neste capítulo são apresentados os conceitos e as tecnologias utilizados para o desenvolvimento deste trabalho.

2.1 Computação Quântica

A computação quântica é uma tecnologia nova com grande potencial que pode vir a complementar ou até substituir a que temos atualmente. Essa tecnologia faz o uso de propriedades da mecânica quântica, tais como as teorias de superposição e interferência(ALMEIDA, 2009), executando os cálculos com um grande desempenho em relação a computação clássica. É importante ressaltar que *computação quântica* abrange mais que uma multiplicação na velocidade dos microprocessadores. Ela cria um novo tipo de computação com novos algoritmos qualitativamente diferentes dos clássicos.

O uso da *computação quântica* permite que algoritmos com ordem exponencial de operações em computadores tradicionais sejam executados em tempo polinomial por computadores quânticos. Um exemplo clássico de desempenho é o algoritmo de Shor(SHOR, 1994), algoritmo quântico para fatorar inteiros em números primos, capaz de quebrar muitos dos sistemas criptográficos em uso atualmente, sendo que o melhor algoritmo da computação clássica(POMERANCE, 1996) demoraria anos. Outro trabalho de sucesso é o protocolo de criptografia baseado em chaves quânticas(BENNETT et al., 1993), que já está sendo usado para fornecer um nível de segurança até então desconhecido.

2.1.1 Bits Quânticos

O sistema básico de informação da computação clássica é o bit tradicional ou *Cbit*, e pode assumir os valores 0 ou 1. Um *Cbit* pode ser representado como um escalar que

pertence a {1,0}:

$$\textit{Cbit} \in \{1, 0\}$$

Na computação quântica as informações são representadas por estados quânticos e não podem ser eficientemente simuladas usando os *Cbits* da computação clássica (FEYNMAN, 1982). Dessa forma, para representer esses estados surge o conceito do bit quântico, o *Qbit*. O *Qbit* pode assumir além dos dois estados tradicionais de um *Cbit*, O ou 1, todos os valores entre eles formando um estado de *superposição* coerente de ambos(MERMIM, 2007).

A característica de *superposição* dos estados quânticos, permite operar sobre todos os estados qûanticos de uma vez(NICOLIELLO, 2009). Para entender melhor, considere uma função booleana *f(X)*, onde "X"é um único bit. Na computação clássica, para testarmos todos os resultados possíveis da função, teriamos que executar ela 2 vezes, uma para X = 0 e outra para X = 1. Já na computação quântica bastaria uma única execução, porque a função pode ser aplicada a uma *superposição* de $|0\rangle$ e $|1\rangle$. Esse recurso é chamado de *paralelismo quântico* e é usado imediatamente no algoritmo de Deutsch, algoritmo quântico que verifica se uma função é constante ou balanceada(DEUTSCH, 1985).

Considerando os estados especiais $|0\rangle$ e $|1\rangle$, também chamados de *estados computacionais básicos*, capazes de formar duas bases ortogonais num espaço vetorial, podemos definir o estado quântico como uma *combinação linear* dessas bases:

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

também representado por um vetor

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Os numéros complexos α e β, representam as probabilidades $|\alpha|^2$ do estado quântico estar no estado 0, e β^2 no estado 1. Estes dois coeficientes precisam estar *normalizados*, por isso eles obedecem a regra da soma das suas probabilidades ser sempre 1:

$$|\alpha|^2 + |\beta|^2 = 1$$

Geometricamente, como as duas bases $|0\rangle$ e $|1\rangle$ são normalizadas, o estado quântico é um vetor unitário dentro de um espaço vetorial complexo de duas dimensões(NIELSEN; CHUANG, 2000). Esse espaço vetorial que armazena os estados quânticos é conhecido na fisica quântica como *espaço de Hilbert*.

Observando a figura 2.1, nota-se o espaço vetorial é representado por uma esfera que é formada pelos estados computacionais básicos $|0\rangle$ e $|1\rangle$. O estado $|\psi\rangle$ é formado pelos coeficientes de suas bases, o que resulta num vetor com o mesmo tamanho do raio da esfera. Segundo os conceitos de *álgebra linear*, isso significa que o estado $|\psi\rangle$ pode assumir infinitos valores, por isso os coeficientes de suas bases são formados por números complexos.

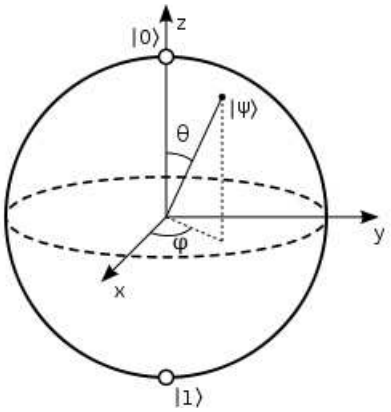

Figura 2.1: Esfera de Bloch representando um qubit

2.1.2 Evolução Unitária dos Bits Quânticos

Existem somente dois tipos de operações sobre bits quânticos, *transformações unitárias* e medidas (observação) sobre um estado. Uma propriedade das *transformações unitárias* é que elas são operações reversíveis, permitindo que as informações não sejam perdidas, ao contrário da operação de medida (ver Seção 2.1.5). Logo, o único modo de alterar o conteúdo de um *qubit* é fazendo o uso de *transformações unitárias*. Consequentemente, a computação quântica se dedica a implementar os seus algoritmos através das *transformações unitárias* nos estados quânticos(PALAO; KOSLOFF, 2002).

Supondo um estado $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, podemos aplicar uma transformação T através de uma matriz 2 x 2:

$$T|\phi\rangle = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Generalizando, uma *transformação unitária* pode ser representada por uma matriz unitária, o que significa que para um estado de *n* qubits uma matriz unitária de tamanho 2^n x 2^n tem que ser usada.

As matrizes unitárias são frequentemente chamadas de *quantum gates* e são usadas na

elaboração dos *quantum circuits*. Importantes *quantum gates* são as:

$$\textit{NOT} \text{ ou Pauli-X } = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \text{ Hadamard ou } H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$\text{Pauli-Z ou } Z = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\textit{CNOT} \text{ ou Not Controlado} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

A aplicação da porta NOT, troca os coeficientes de $\alpha|0\rangle + \beta|1\rangle$ para $\beta|0\rangle + \alpha|1\rangle$. A diferença para a CNOT é que ela aplica a porta NOT somente se o primeiro qubit for 1, senão ela não faz nada. A Handmard é muito utilizada, ela transforma um estado quântico em estado de *superposição coerente* mapeando os estados bases com peso igual. A porta Z representa uma mudança de fase complexa. A figura 2.1.2 mostra as representações de algumas portas em um circuito quântico.

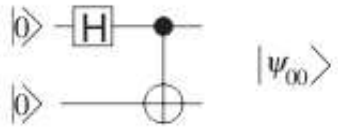

Figura 2.2: Circuito quântico do algoritmo de criptografia (MERMIM, 2007)

2.1.3 Composição de Bits Quânticos

Individualmente cada qubit é descrito por um vetor em um espaço de Hilbert H. Assim se temos um sistema de dois qubits, temos dois espaços de Hilbert, que formalmente estariam perto um do outro. A única maneira de descrever esse sistema composto é construir um espaço a partir dos espaços dos estados de seus sistemas componentes através do *produto tensorial* entre eles(MERMIM, 2007). O *produto tensorial* é uma forma de compor espaços vetorias para formar espaços vetoriais de dimensões maiores. Dessa forma, um estado de *N* qubits pode ser expresso como um vetor num espaço dimensional 2^N, sendo esse espaço dimensional oriundo do *produto tensorial* dos espaços de Hilbert de cada qubit em questão. O produto tensorial pode ser visto como a maneira formal de aninhar esses dois espaços de Hilbert.

$$|\psi\rangle \in H_1$$

$$|\phi\rangle \in H_2$$

$$|\psi\rangle \otimes |\phi\rangle \in H_1 \otimes H_2$$

Supondo um estado com dois qubits, cada um com bases $\{|0\rangle, |1\rangle\}$, podemos construir os *sistemas computacionais básicos* desse sistema pelo produto tensorial das duas bases ortonormais: $\{|0\rangle.|1\rangle\} \otimes \{|0\rangle.|1\rangle\} = \{|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle\}$, também representados como $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. O estado quântico de dois qubits é a *combinação linear* dos quatro estados basicos encontrados:

$$|\psi\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle = \sum_{i,j\in\{0,1\}} a_{ij}|i,j\rangle$$

onde a_{00}, a_{01}, a_{10} e a_{11} são coeficientes complexos que representam a probabilidade a_{00}^2 do estado $|\psi\rangle$ estar em $|00\rangle$, a_{01}^2 em $|01\rangle$, a_{10}^2 em $|10\rangle$ e a_{11}^2 do estado estar em $|11\rangle$. Lembrando que os coeficientes devem estar *normalizados*, logo:

$$|a_{00}|^2 + |a_{00}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1$$

Pode-se concluir que um estado de *N* qubits é um vetor não nulo, num espaço de Hilbert, que pode ser representado formalmente pelas combinações lineares:

$$\sum_{b1,\ldots,bn\in\{0,1\}} a_{b1,\ldots,bn}|b1\ldots bn\rangle, com \sum_{b1,\ldots,bn\in\{0,1\}} |a_{b1\ldots bn}|^2$$

2.1.4 Estados Emaranhados

Um estado de dois qubits importante é o estado de de *Bell* ou par EPR:

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Esse par tem a propriedade de que uma medida no primeiro qubit sempre dará o mesmo resultado que uma medida no segundo qubit e vice-versa. Para entender melhor, observe que se medirmos o primeiro qubit, obtemos dois possíveis resultados: um com uma probabilidade 1/2, deixando o estado após a medida como $|00\rangle$, outro com uma probabilidade de 1/2 deixando o estado como $|11\rangle$. Assim, como uma medida no primeiro qubit interfere no segundo qubit, dizemos que as operações de medidas desse estado estão *correlacionadas*(NIELSEN; CHUANG, 2000).

Estados desse tipo são chamados de estados *emaranhados* e não podem ser descritos como um produto tensorial de seus componentes, consequentemente não podem ser separados facilmente. Por exemplo, considere novamente o par EPR acima. Não podemos

representar o estado como:

$$|\psi\rangle = (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle)$$

$$|\psi\rangle = (ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle$$

pois isso implica que $ad = 0$ e portanto ou $a = 0$ ou $b = 0$ o que anularia o estado $|00\rangle$ ou $|11\rangle$ de existir.

Estados emaranhados são muito importantes na computação quântica. São eles que tornam possível o fenômeno da *teleportação quântica* (descrito na seção 2.1.6).

2.1.5 Operações de Medidas

Na computação clássica, para sabermos o valor de um Cbit, apenas o observamos e obtemos como resposta o valor 0 ou 1, não alterando seu contéudo. Contudo, devido as teorias de interferência da mecânica quântica ao medirmos o valor de um Qbit o seu estado entra em *colapso* e se altera. Infelizmente, segundo o teorema da não-clonagem, não é possível copiar o estado de um Qbit para salvar as informações, antes de efetuar uma medição. Dessa forma, a medida não permite recuperar o estado anterior à medição, o que a torna a única operação *irreversível* da computação quântica.

Figura 2.3: Representação de uma operação de medida em um circuito quântico

2.1.6 Algoritmo Quântico da Teleportação

A *Teleportação Quântica* é uma técnica para mover mover estados quânticos, mesmo na ausência de um canal de comunicação quântico entre a origem e o destino do estado quântico. Em essência, a *teleportação quântica* permite uma conexão de um estado quântico desconhecido através de um par EPR compartilhado anteriormente (BENNETT et al., 1933).

Suponha que Alice e Bob se encontraram há muito tempo, mas agora vivem longe. Enquanto estavam juntos eles geraram um par EPR (ver Tabela 2.1.6) e cada um pegou um qubit do par. Anos depois, a missão de Alice é entregar um qubit, $|\psi\rangle$, a Bob. Ela não sabe o estado do qubit, e somente pode enviar informações clássicas para Bob.

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

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

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

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

Tabela 2.1: Pares EPR

Alice não conhece o estado $|\psi\rangle$ do qubit que ela precisa enviar para Bob, e as leis da mecânica quântica não permitem determinar o estado quando ela tem uma única cópia de $|\psi\rangle$. O que é pior, mesmo que ela conhecesse o estado $|\psi\rangle$, descrevê-lo, leva uma quantidade infinita de informação clássica. Felizmente para Alice, *teleportação quântica* é uma maneira de utilizar o par EPR para enviar $|\psi\rangle$ para Bob, com um pequeno custo de comunicação clássica.

Os passos da solução proposta pelo algoritmo são os seguintes: Alice interage seu qubit $|\psi\rangle$ com sua metade do par EPR, operação CNOT, e então observa os dois qubits em sua possessão, obtendo um dos quatro possíveis resultados clássicos, $00,\ 01,\ 10,\ 11$. Ela envia essa informação para Bob.

Dependendo da mensagem clássica de Alice, Bob executa uma das quatro operações na sua metade do par EPR. Supreendentemente, fazendo isso, ele pode reconstituir o estado original $|\psi\rangle$.

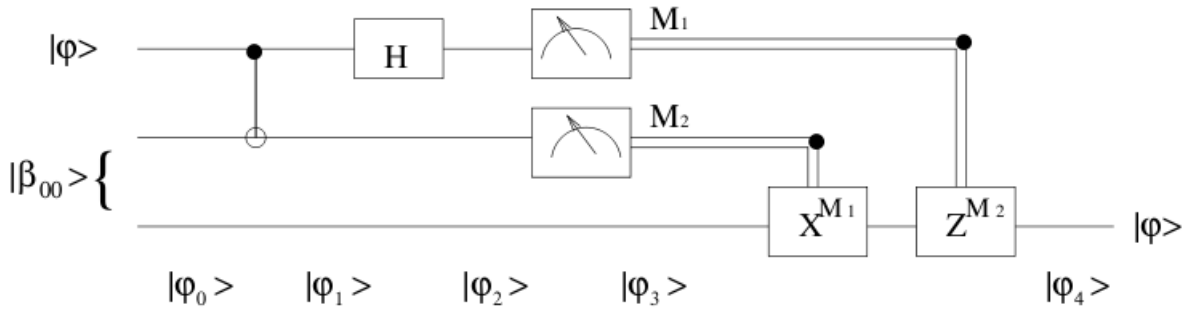

Figura 2.4: Circuito quântico do algoritmo da teleportação

Suponha que o estado a ser teleportado é $|\psi\rangle = \alpha|0\rangle+\beta|1\rangle$, onde α e β são amplitudes deconhecidas. A Figura 2.1.6, mostra um circuito quântico para teleportar um qubit. As

duas linhas de cima representam o sistema da Alice, enquanto que a linha de baixo é o sistema de Bob. As linhas duplas denotam a transmissão de bits clássicos e as linhas simples representam qubits.

O estado de entrada do circuito é

$$\begin{aligned} |\psi_0\rangle &= |\psi\rangle|\beta_{00}\rangle \\ &= \tfrac{1}{\sqrt{2}}[\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|00\rangle + |11\rangle)]. \end{aligned}$$

onde utilizamos a convenção que os dois primeiros qubits (da esquerda) pertencem à Alice e o terceiro quibit a Bob. Como vimos, o segundo qubit de Alice e o qubit de Bob começam em um estado EPR. Alice, envia seus qubits através de uma porta CNOT, obtendo

$$|\psi_1\rangle = \tfrac{1}{\sqrt{2}}[\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|10\rangle + |01\rangle)].$$

Então, ela aplica a porta Hadamard no seu primeiro qubit:

$$|\psi_2\rangle = \tfrac{1}{2}[\alpha(|0\rangle + |1\rangle)(|00\rangle + |11\rangle) + \beta(|0\rangle - |1\rangle)(|10\rangle + |01\rangle)].$$

Simplesmente, reagrupando os termos, esse estado pode ser reescrito da seguinte maneira:

$$\begin{aligned} |\psi_2\rangle &= \tfrac{1}{2}[(\alpha|0\rangle + \alpha|1\rangle)(|00\rangle + |11\rangle) + (\beta|0\rangle - \beta|1\rangle)(|10\rangle + |01\rangle)] \\ &= \tfrac{1}{2}[(\alpha|000\rangle + \alpha|011\rangle) + (\alpha|100\rangle + \alpha|111\rangle) \\ &\quad +(\beta|010\rangle + \beta|001\rangle) - (\beta|110\rangle + \beta|101\rangle)] \\ &= \tfrac{1}{2}[|00\rangle(\alpha|0\rangle + \beta|1\rangle) + |01\rangle(\alpha|1\rangle + \beta|0\rangle) \\ &\quad +|10\rangle(\alpha|0\rangle - \beta|1\rangle) + |11\rangle(\alpha|1\rangle - \beta|0\rangle)] \end{aligned}$$

Essa expressão pode ser quebrada em quatro termos. O primeiro termo tem os qubits de Alice no estado $|00\rangle$, e o qubit de Bob no estado $\alpha|0\rangle + \beta|1\rangle$ - que é o estado original $|\psi\rangle$. Se Alice executar uma observação e obtiver o resultado 00, então o sistema de Bob estará no estado $|\psi\rangle$. Similarmente, a partir da expressão acima, podemos ver o estado de Bob após a observação:

$$\begin{aligned} 00 &\mapsto |\psi_3(00)\rangle \equiv [\alpha|0\rangle + \beta|1\rangle] \\ 01 &\mapsto |\psi_3(01)\rangle \equiv [\alpha|1\rangle + \beta|0\rangle] \\ 10 &\mapsto |\psi_3(10)\rangle \equiv [\alpha|0\rangle - \beta|1\rangle] \\ 11 &\mapsto |\psi_3(11)\rangle \equiv [\alpha|1\rangle - \beta|0\rangle] \end{aligned}$$

Dependendo da saída da observação de Alice, Bob estará em um desses quatro estados possíveis. Para Bob saber em qual resultado está, ele precisa saber o resultado da

observação de Alice (esse é o fato que previne que a teleportação seja utilizada para transmitir informação mais rápida que a luz). Uma vez que Bob sabe a saída da observação de Alice, ele pode utilizar seu estado para recuperar $|\psi\rangle$, aplicando a porta lógica apropriada. Por exemplo, no caso onde a observação produz $|00\rangle$, Bob não precisa fazer nada. Se a observação é $|01\rangle$ então Bob pode transformar seu estado aplicando a porta X em $\alpha|1\rangle + \beta|0\rangle$:

$$\begin{bmatrix} 0 & \beta \\ \alpha & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \beta & 0 \\ 0 & \alpha \end{bmatrix}$$

obtendo, dessa maneira, o estado $|\psi\rangle$.

Existem diversas características importantes na teleportação. Primeiramente, podemos notar que teleportação quântica poderia possibilitar uma comunicação mais rápida que a luz, senão fosse o fato de ele precisar usar um *canal de comunicação clássico* para sua execução (BENNETT et al., 1933). Uma segunda observação sobre a teleportação é que parece que criamos uma cópia do estado quântico que está sendo teleportado, violando, assim, o teorema da não-clonagem. Essa violação é somente ilusória, por que depois que o processo de teleportação está completo, somente o qubit destino é deixado no estado $|\psi\rangle$. O qubit original termina em uma das bases computacionais $|0\rangle$ ou $|1\rangle$.

2.2 Java

Java é uma linguagem de programação orientada a objeto desenvolvida na década de 90 por uma equipe de programadores chefiada por James Gosling, na empresa Sun Microsystems. Diferentemente das linguagens convencionais, que são compiladas para código nativo, a linguagem Java é compilada para um bytecode que é executado por uma máquina virtual. Essa linguagem foi escolhida para este trabalho, por causa da sua portabilidade e capacidade de processamento paralelo (multi-thread).

2.2.1 Java Closures

Closure é um tipo de *função anônima* tipicamente usado em linguagens funcionais. A palavra *closure* origina-se do termo *closed expression*(expressão fechada), que se refere a uma expressão *lambda* onde suas variáveis livres foram fechadas por um ambiente léxico.(LANDIN, 1964). Em essência, um *closure* é um trecho de código que pode ser passado como argumento para uma chamada de função. Dessa forma, o uso de closures

permite a criar funções dinamicamente, ou seja, em tempo de execução.

Um *closure* é uma função criada dinamicamente em tempo de execução do programa, que captura as variáveis livres automaticamente (LANDIN, 1964). Um *closure* pode usar variáveis que estão dentro de um escopo, mesmo que esse escopo não esteja ativo no momento de sua chamada, se ele é passado como um argumento para um método, ele vai continuar usando as variáveis do escopo onde ele foi criado (ECHEVARRIA, 2010).

Algumas linguagens de programação não-funcionais já estão incorporando *closures* na sua implementação (Python e Ruby por exemplo), Java não fica para trás e pretende incorporar essa funcionalidade com o lançamento da JDK 7. Felizmente, é possível incorporar um protótipo que dê suporte a closures e funções anônimas em Java fazendo o uso de uma extensão chamada *BGGA Closures* (GAFTER, 2009).

Usando BGGA, uma função anônima pode ser definida usando a sintaxe: {parâmetros formais => expressão}, onde tanto os *parâmetros formais*, quanto a *expressão*, são opcionais. Por exemplo:

```
{int x => x * 2}
```

A função definida acima, é uma função que recebe um inteiro e retorna seu valor elevado ao quadrado. Para chamar uma função anônima usamos o método *invoke*:

```
int answer = { => 42 }.invoke();
System.out.println(answer); //Imprime 42
```

Figura 2.5: Exemplo da chamada de função *invoke*

Uma função anônima também pode ser referenciada por variáveis. Uma variável pode possuir um tipo que é uma função, sendo a sintaxe `{parâmetros formais => tipo de retorno}`. Por exemplo:

```
{int, String => String } func = { int x, String s =>
      "Nome:" + s + " Idade:" + String.valueOf(x) };
String pessoa = (func.invoke(28,"Carlos"));
```

Figura 2.6: Exemplo de definição de um *closure* com diferentes argumentos

A variável `func` possui o tipo `{ int, String => String}` o que significa que é uma função com dois parâmetros, um `int` e uma `String`, e retorna uma `String`. O resultado da chamada *invoke* acima é : "Nome:Carlos Idade:28". Com a extensão BGGA, podemos também passar funções como argumentos para métodos Java:

```
public static print( {int => int} func){
   for (int i = 0; i < 4; i++)
      System.out.println(func.invoke(i));
}

public static void main(String args[]) {
   int x = 1;
   {int => int} func = {int y => x+y};
   print(func);    //Imprime 1,2,3,4
   x++;
   print(func);    //Imprime 2,3,4,5
}
```

Figura 2.7: Exemplo de passagem de um *closure* como argumento

A variável `func` possui o tipo `{ int => int}` e é passada como argumento para o método *print*, que recebe uma função e imprime os resultados da sua execução. Observe também, que o *closure* engloba a variável `x`, pois está dentro de seu contexto.

2.3 Mônadas

Uma mônada é uma maneira de descrever computações, em termos de valores e sequências de computações, que podem ser combinadas gerando novas computações (NEWBERN, 2009). Por esta razão, mônadas são principalmente usadas em linguagens funcionais puras, onde existe o conceito de função pura. Elas podem ser vistas como um conceito matemático utilizado para modelar efeitos colaterias tais como entradas, saídas e até estados. Existem diferente tipos de mônadas e estratégias para combinar sua computação. Algumas já são partes da biblioteca padrão da linguagem *Haskell*.

Uma mônada pode ser implementada como um tipo abstrato que representa um container para uma computação. Essas computações podem ser criadas e combinadas usando a operação *bind*. Uma mônada é definida por:

1. Um construtor de tipo T
2. Uma função *unit*:

   ```
   unit ::  a -> T a
   ```

3. Uma função *bind*:

   ```
   bind ::  T a -> ( a -> T b) -> T b
   ```

2.3.1 Mônada Quântica

Uma mônada para representar estados quânticos, pode ser modelada levando em conta que, os coeficientes das bases computacionais quânticas são números complexos, e podem ser mapeados por um tipo `A`. Considere o estado quântico $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ por exemplo, um tipo A que mapeie α e β, pode representado por um tipo *booleano*.

A função *unit* da mônada deve receber como parâmetro um tipo A, e retornar um vetor que mapeie o valor de A para um, e o resto para zero. Essa função serve para criar qualquer vetor quântico básico. A função *bind* deve combinar transformações e aplicar em um vetor, essa função aplicará qualquer tipo de transformação em estado quântico. Essas tranformações são unitárias e vão ser sequências de computações que podem gerar novas computações.

Uma implementação genérica usando a linguagem *Haskell* (VIZZOTTO; ALTENKIRCH; SABRY, 2006), é descrita como:

```
data Vec a = a -> ℂ
unit a1 a2 = if ( a1 == a2) then 1.0 else 0.0
bind va f = λ b -> sum[va a * f a b]
```

Devido a alta capacidade de modelagem e flexibilidade das mônadas, elas apresentam uma solução ótima a respeito da modelagem dos estados quânticos da computação quântica.

3 IMPLEMENTAÇÃO DO MODELO MONÂDICO QUÂNTICO COM JAVA CLOSURES

Para implementar as mônadas descritas na seção 2.3, foi desenvolvido em linguagem Java uma classe chamada Monad.java. Além disso, foi desenvolvido um conjunto de classes auxiliares para ajudar na manipulação dos dados como pode ser visto no diagrama da figura 3.1. As próximas seções apresentam detalhes dessas implementações.

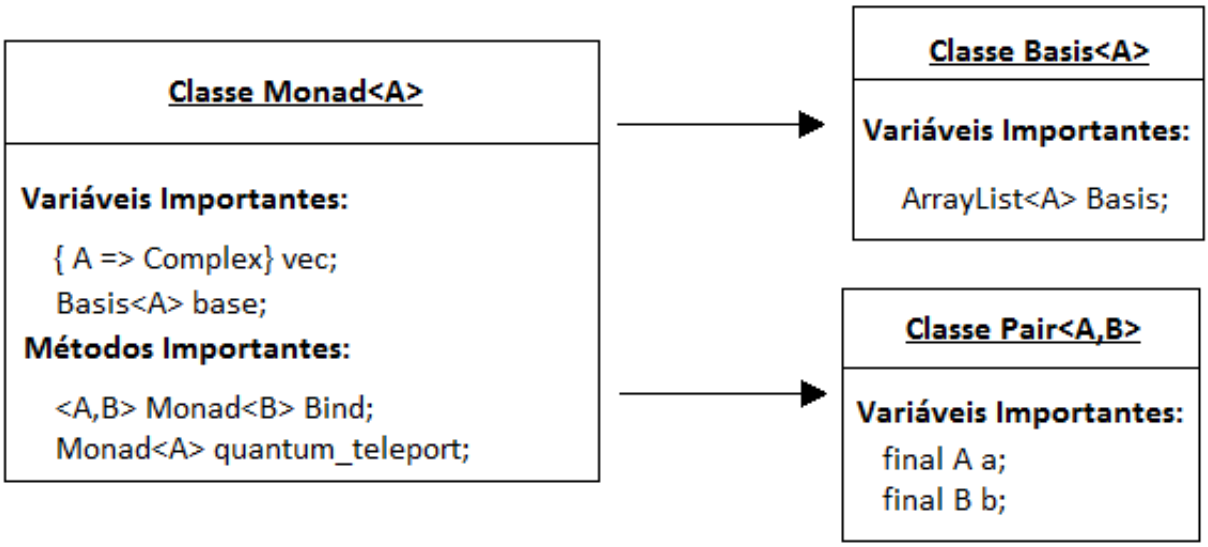

Figura 3.1: Diagrama de Classes

3.1 Classe Monad

A mônada quântica é implementada através da classe Monad<A>, que é usada para representar estados quânticos e possui como atributo uma função `vec` que mapeia A para *double*, pois Java não tem suporte nativo a números complexos. Essa função representa os coeficientes do estado quântico. Outro atributo é uma instância da classe Basis<A>,

que contém uma lista dos possíveis valores das bases do tipo genérico A (ver seção 3.2.1).

```
public class Monad<A> {
   { A => double } vec;    //Funcao que mapeia um tipo A para complexo
   Basis<A> base;          //Lista dos bases do vetor vec
   //Construtor
   Monad(Object tipo, {A => double} vec){
      this.vec = vec;
      this.base = new Basis<A>(tipo);
   }
   //Construi um vetor basico (unit)
   Monad(Object b){
      this.base = new Basis<A>(b);    //Lista das bases do vetor vec
      this.vec = { A bool =>
         float resultado = 0 ;
            if (bool.toString().equals(b.toString()))
               resultado = 1;  //Retorna um somente na posicao passada
         resultado    //retorna o resultado
      };
   }
```

Figura 3.2: Classe Monad<A>

A função `unit` das mônadas é o construtor de tipos básicos e é implementada pelo construtor da classe (Figura 3.3). Para construir a base $|0\rangle = 1*|0\rangle + 0*|1\rangle$, basta chamar um construtor com o argumento *false*. Quando chamado o método *invoke(A)* do vetor, ele retorna o número complexo referente ao coeficiente da base A passada.

```
   Monad<Boolean> base0 = new Monad<Boolean>(false);
   System.out.println(base0.vec.invoke(false)); //Imprime 1 + 0i
   System.out.println(base0.vec.invoke(true));  //Imprime 0 + 0i
```

Figura 3.3: Base $|0\rangle$ criada com o método `unit`

O método *bind* serve para combinar transformações. Ele aplica uma transformação `f` no vetor `v`, sendo que f é uma função que representa uma matrix unitária e v o estado quântico. O método *bind* é usado para aplicar transformações do tipo NOT, Handmard, entre outras.

É possível combinar qualquer transformação unitária representada for uma função `{ A => Monad<B> v}`, sobre um estado quântico. Algumas transformações comumente usadas, como a NOT, CNOT e Hadamard, estão implementadas com métodos que geram essas transformações dinamicamente levando em conta o tamanho do tipo genérico do vetor a ser transformado.

Outro método auxiliar criado é o método que soma e subtrai dois vetores. Esses mé-

```
//Bind generico
//Aplica a transformacao F no vetor V
public static <A,B> Monad<B> bind(Monad<A> v, {A => Monad<B>} f){
   return new Monad<B> ( v.base.get(0),
      {B bool =>
         Basis<A> ba = v.base;              //Lista de bases do vetor v
         Basis<B> bb = new Basis<B>(bool); //Lista de bases do vetor vb
         double resultado = 0;
         for(int i = 0; i < bb.size(); i++){
            //Encontra a posicao no vetor
            //que contem a transformacao a ser aplicada
            //na posicao bool
            if ( bool.toString().equals(bb.get(i).toString())){
               //O vetor Vtemp e' o vetor com os valores que
               //que seram aplicadados
               //nos coeficientes do vetor V
               Monad<B> vtemp = f.invoke(ba.get(i)); //Pega o vetor
               //Percorre os vetores e executa a transformacao
               for(int j = 0; j < ba.size(); j++){
                  resultado += vtemp.vec.invoke(bb.get(j)) *
                     v.vec.invoke(ba.get(j));
               }
               break;
            }
         }
         resultado //Retorna o novo valor gerado pela transformacao f
      }
   );
}
```

Figura 3.4: Método *bind*

todos são usados na criação da tranformação Hadamard, e no caso da soma ela também é usada para gerar um estado de superposição dos dois vetores somados. Como os coeficientes devem estar normalizados, a função normaliza recebe como parâmetro o vetor a ser normalizado.

A operação de medida é simulada com um gerador de números aleatórios entre zero e um. O método medida percorre os coeficientes do vetor passado como argumento, em busca do intervalo em que se encontra o número gerado. Quando encontrado, ele retorna o valor do estado quântico, que é o valor da posição encontrada. Esse método serve para simular a medida (observação) de estado e causa um *colapso* no estado, setando ele como uma base computacional igual ao resultado da medida. Ou seja, se a medida retorna false, então o estado é uma base computacional $|0\rangle$, com probabilidade um de estar no estado $|0\rangle$, e zero no estado $|1\rangle$.

```
//Cria uma transformacao NOT dinamicamente
//Retorna um vetor com o tamanho do tipo de base
public static <A> {A => Monad<A> } make_not(Monad<A> v){
    return { A bool =>
        Basis<A> ba = v.base;
        Monad<A> re = null;
        for (int i = 0; i < ba.size(); i++){
            if (bool.toString().equals(ba.get(i).toString())){
                re = new Monad<A>(ba.get(ba.size()-1-i));
                break;
            }
        }
        re
    };
}
```

Figura 3.5: Método que cria a transformação unitária NOT

3.2 Classe auxiliares

3.2.1 Classe Basis

A classe Basis<A> serve para gerar uma lista que contém todos valores possíveis do tipo genérico. Serve para auxiliar na execução do método *bind*, que percorre essa lista em busca de todas posições possíveis.

A lista é gerada levando em conta o tamanho do argumento passado ao construtor. A partir disso, são anexadas a lista, todos valores (posições) que o tipo genérico pode assumir.

O motivo de criar essa classe, é que a mônada usa um tipo genérico e precisa percorrer todas as posições do vetor genérico, logo é preciso saber de antemão quais são todas posições possíveis do vetor. Foi usada uma lista para bases e não um simples array, pois ficou mais prático representar o tipo genérico A com a classe ArrayList<A>.

3.2.2 Classe Pair

Devido a representação genérica das mônadas, a representação das bases de um estado de 2 ou mais qubits, deveria ser feita através de tuplas<Boolean,Boolean,...>. Contudo, na linguagem Java não tem suporte a *tuplas* infinitas, por isso a solução encontrada foi implementar a classe Pair<A,B>, que representa uma tupla de dois elementos genéricos.

Com esta classe, é possível criar pares de tipos genéricos diferentes. Um estado de dois qubits pode ser criado com uma tupla de dois booleanos. Além do construtor, essa classe apresenta as funções getA() e getB(), que retornam o objeto A ou B respectiva-

```
//Soma dois vetores ba'sicos
//Retorna um vetor com os coeficentes referente a soma
//dos coeficientes de cada base normalizados
public static <A> Monad<A> soma (Monad<A> v1, Monad<A> v2){
   return new Monad<A>(v2.base.get(0), { A bool =>
      double re = 0;
      re = (v1.vec.invoke(bool) + v2.vec.invoke(bool));
      re
    });
}
//Subtrai dois vetores ba'sicos
//Retorna um vetor com os coeficentes referente a soma
//dos coeficientes de cada base normalizados
public static <A> Monad<A> sub (Monad<A> v1, Monad<A> v2){
   return new Monad<A>(v2.base.get(0), { A bool =>
      double re;
      re = (v1.vec.invoke(bool) - v2.vec.invoke(bool));
      re
    });
}
//Normaliza o valor do coeficiente
//deve ser usada apos uma soma ou subtracao
public static <A> Monad<A> normaliza (Monad<A> v, int i){
   return new Monad<A>(v.base.get(0), { A bool =>
      double re;
      re = v.vec.invoke(bool);
      re = re / Math.sqrt(i);
      re
    });
}
```

Figura 3.6: Método auxliaries para construção de vetores

mente.

Após a implementação dessa classe, foi possível criar estados quânticos maiores e principalemente estados emarranhados. O uso destas tuplas facilitou o desenvolvimento deixando mais eficiente a utilização da classe Monad.

3.3 Algoritmo Quântico da Teleportação

Com a definição da mônada, o algoritmo foi implementado em cima de um estado quântico com três qubits. A criação do par EPR, pode ser feita criando um estado básico $|00\rangle$, aplicando no primeiro qubit a transformação Hadamard e na sequência um not controlado.

Com as operações acima, o estado *bell_state* criado, representa o estado emarranhado descrito como:

$$|bell\_state\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

```
    //Realiza a medida no vetor passado
    public static <A> A medida(Monad<A> v){
        Random generator = new Random();
        double temp = generator.nextDouble();
        Basis<A> ba = v.base;
        double probabilidade;
        double comp = 0;
        double comp2 = 0;
        for (int i = 0; i < ba.size(); i++){
            //Pega a probabilidade do estado estar na posição i
            probabilidade = Math.pow(v.vec.invoke(v.base.get(i)),2);
            comp2 += probabilidade;
            //Compara se o número gerado encontra-se no intervalo
            if (comp < temp && temp < comp2 )
                break;
            else
                comp += probabilidade;
        }
        //Simula o colapso da medida
        //fazendo o estado virar uma base básica
        //com o valor da medida
        v =  new Monad<A>(ba.get(i));
        return ba.get(i);
    }
```

Figura 3.7: Método que simula a medida da computação quântica

```
public class Basis<A>{
    //Lista das bases
    ArrayList<A> basis;

    ...
}
```

Figura 3.8: Classe Basis

Para aplicar o algortimo da teleportação, esse par EPR deve ser compartilhado por Alice e Bob. Para representar os estados de Alice e Bob, podemos usar uma instância da classe Monad, que utilize como bases Boolean,Pair<Boolean,Boolean>.

O algoritmo aplica em Alice a transformação Cnot em seu par EPR. Na sequência aplica-se Hadamard somente no primeiro qubit de Alice. Essas transformações deixam o estado de Alice em quatro possíveis soluções.

$$
\begin{aligned}
00 &\mapsto |\psi_3(00)\rangle \equiv [\alpha|0\rangle + \beta|1\rangle] \\
01 &\mapsto |\psi_3(01)\rangle \equiv [\alpha|1\rangle + \beta|0\rangle] \\
10 &\mapsto |\psi_3(10)\rangle \equiv [\alpha|0\rangle - \beta|1\rangle] \\
11 &\mapsto |\psi_3(11)\rangle \equiv [\alpha|1\rangle - \beta|0\rangle]
\end{aligned}
$$

O próximo passo é realizar a medida e enviar essa informação a Bob. Se o resultado da observação de Alice for $|00\rangle$, Bob não precisa fazer nada. Senão, Bob precisa aplicar

```
public class Pair<A,B> {
    public final A a;
    public final B b;
    ...
}
```

Figura 3.9: Classe Pair

```
    Monad<Pair> bell_state = new Monad<Pair>(new Pair(false,false));
    bell_state = bind(bell_state, make_H());
    bell_state = bind(bell_state, make_Cnot(bell_state));
```

Figura 3.10: Construção de um par EPR

uma transformação apropriada. Por exemplo, se a observação é $|01\rangle$ então Bob pode transformar seu estado aplicando a porta NOT em $\alpha|1\rangle + \beta|0\rangle$:

$$\begin{bmatrix} 0 & \beta \\ \alpha & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \beta & 0 \\ 0 & \alpha \end{bmatrix}$$

O método que implementa o algoritmo da teleportação, recebe dois vetores Alice e Bob, e retorna o vetor de Bob com o estado que era de Alice. Alice ao realizar a medida, perde suas informações.

```
public static <A> Monad<A> quantum_teleport(Monad<A> Alice,
                                            Monad<A> Bob){
    //Aplica a transformações Cnot
    Alice = bind(Alice, make_Cnot2(Alice));
    //Aplica Hadamard apenas no primeiro qubit
    Alice = bind(Alice, make_Hadamard1(Alice));
    //Realiza a medida nos dois primeiros qubits
    A medida = medida2(Alice);
    //Procura o resultado da medida
    if (A.toString().equals("false,false"){
        //Não faz nada
    }
    else if (A.toString().equals("false,true"){
        //Aplica a porta X
        Bob = bind(Bob, make_not(Bob));
    }
    else if (A.toString().equals("true,false"){
        //Aplica a porta Z
        Bob = bind(Bob, make_pauli(Bob));
    }
    else
        //Aplica a porta Z, e depois X
        Bob = bind(Bob, make_pauli(Bob));
        Bob = bind(Bob, make_not(Bob));
    }
    return Bob;
}
```

Figura 3.11: Implementação do Algoritmo Quântico

4 CONCLUSÕES

A computação quântica é uma tecnologia nova com grande potencial e pode vir a complementar ou até substituir a que temos atualmente. Contudo, ela abrange mais que apenas uma multiplicação na velocidade dos microprocessadores, ela introduz uma nova abordagem com novos algoritmos qualitativamente diferentes dos clássicos. Um do desafios dessa área é desenvolver linguagens de programação quântica e seus modelos semânticos.

Dessa forma, este trabalho apresentou um modelo monádico para representar estados quânticos, bem como, os conceitos básicos da computação quântica. Espera-se, que o produto desenvolvido seja usado para implementar outros algoritmos e operações da computação quântica, e também , que sirva como base de estudo para alunos interessados na computação quântica.

Foram explicados nesse trabalho, diversos conceitos básicos sobre a computação quântica necessários para a criação de um modelo para representar os estados quânticos, principalmente o conceito que a informação quântica não pode ser representada corretamente por um bit clássico. Além disso, é importante ressaltar que a única forma de alterar um estado quântico é através de transformações unitárias.

A modelagem criada com o conceito de mônadas, se resume em combinar transformações em um dado vetor, e assim gerar novos vetores. Dessa forma, o estado quântico pode ser visto como um efeito colateral da mônada criada, combinando transformações unitárias gerando novos estados. O método *bind* implementado, é capaz de combinar qualquer tipo de transformação a um vetor desde que eles sejam do mesmo tipo genérico. Outro aspecto do modelo, é que a medida foi simulada através de sorteio de números aleatórios, o que não é o esperado de um estado quântico mas serviu para as simulações serem executadas.

Também foram definidas algumas portas quânticas tais como: *Not*, *CNot*, *Pauli-Z* e *Hadamard*. A implementação do algoritmo quântico da teleportação mostrou-se ser bem simples. Todavia, o algoritmo está funcional para teleportar apenas um qubit. Pretende-se continuar este trabalho, e assim implementar o algoritmo genericamente.

A implementação do algoritmo foi desenvolvida em linguagem Java e deve ser de fácil interpretação. Espera-se que este trabalho, ajude alunos da ciência da computação na compreensão das características da computação quântica e que desperte interesse pela área.

O trabalho seguirá em desenvolvimento e tem como proposta de trabalhos futuros:

- Implementar o produto tensorial ao modelo
- Implementar o Algoritmo Quântico da Teleportação genericamente
- Criar uma classe que represente números complexos
- Publicação de resultados

REFERÊNCIAS

ALMEIDA, R. R. de. **Computação Quântica**: um melhor entendimento do mundo quântico. Disponível em `http://www.webartigos.com/articles/31294/1`. Acesso em: Agosto de 2010.

BENNETT, C. H.; BRASSARD, G.; CRÉPEAU, C.; JOZSA, R.; PERES, A.; ; WOOTTERS, W. K. **Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels**. 1993.

BENNETT, C. H.; BRASSARD, G.; CREPEAU, C.; JOZSA, R.; PERES, A. W. W. Teleportation an unknown quantum state via dual classical EPR channels. In: 1933. **Anais...** [S.l.: s.n.], 1933.

DEUTSCH, D. Quantum Theory, the Church-Turing Principle, and the Universal Quantum Computer. In: 1985. **Anais...** [S.l.: s.n.], 1985. v.47, p.97–117.

ECHEVARRIA, M. G. **Uma linguadem de Domínio Específico para Programação de Memórias Transacionais em Java**. 2010.

FEYNMAN, R. Simulating physics with computers. **International Journal of Theoretical Physics**, [S.l.], v.21, p.467–488, 1982.

GAFTER, N. **Java Closures**. Disponível em `http://www.javac.info/`. Acesso em: Agosto de 2010.

LANDIN, J. P. The mechanical evaluation of expressions. In: 1964. **Anais...** [S.l.: s.n.], 1964.

MERMIM, N. D. **Quantum Computer Science**. [S.l.]: Cambridge University Press, 2007.

MOGGI, E. Computational lambda-calculus and monads. In: FOURTH ANNUAL SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, 1989. **Proceedings...** IEEE Press, 1989. p.14–23.

NEWBERN, J. **All About Monads**. Disponível em `http://www.haskell.org/all_about_monads/html/index.html`. Acesso em: Agosto de 2010.

NICOLIELLO, H. **Introdução a Computação Quântica**. 2009.

NIELSEN, M. A.; CHUANG, I. L. Quantum Computation and Quantum Information. In: IEEE PRESS, 2000. **Anais...** [S.l.: s.n.], 2000.

PALAO, J. P.; KOSLOFF, R. Quantum Computing by an Optimal Control Algorithm for Unitary Transformations. In: 2002. **Anais...** [S.l.: s.n.], 2002. v.89.

POMERANCE, C. A Tale of Two Sieves. **Notices Amer. Math. Soc.**, [S.l.], v.43, p.1473–1485, 1996.

RIVEST, R. L.; SHAMIR, A.; ADELMAN, L. M. **A Method for Obtaining Digital Signatures and Public-Key Cryptosystems**. [S.l.]: MIT, 1977. (MIT/LCS/TM-82).

SHOR, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In: IN PROC. IEEE SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 1994. **Anais...** [S.l.: s.n.], 1994. p.124–134.

VIZZOTTO, J. K.; ALTENKIRCH, T.; SABRY, A. Structuring Quantum Effects: superoperators as arrows. **Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages**, [S.l.], v.16, p.453–468, 2006.

APÊNDICE A CÓDIGOS FONTE

A.1 Classe Principal

Listing A.1: Classe Monad<A>

```
import java.util.Random;

//Classe que representa a monada quantica
public class Monad<A> {
   //Funcao que mapeia um tipo A para complexo
   { A => double } vec;

   //Lista dos bases do vetor vec
   //depende do A: Boolean ou Boolean,Boolean, por exemplo
   Basis<A> base;

   //Construtor
   Monad(Object tipo, {A => double} vec){
      this.vec = vec;
      //Cria a lista das bases do vetor vec
      this.base = new Basis<A>(tipo);
   }

   //Construtor de tipos basicos
   //simula o metodo unit da monada
   Monad(Object arg){
      //Cria a lista das bases do vetor vec
      this.base = new Basis<A>(arg);
      //Define o vetor sendo uma funcao
      //que somente tera valor 1 (100%)
      //para o argumento passado
      this.vec = { A bool =>
         float re = 0 ;
            if (bool.toString().equals(arg.toString()))
               re = 1;
         //Retorna a variavel re
         re
      };
   }

   //Bind generico, combina computacoes
   //Aplica a transformacao F no vetor V
   public static <A,B> Monad<B> bind(Monad<A> v, {A => Monad<B>} f){
    return new Monad<B> ( v.base.get(0),
         {B bool =>
            //Pega a lista das bases do vetor v
            Basis<A> ba = v.base;
            //Pega a lista das bases do vetor temporario vtemp
            Basis<B> bb = new Basis<B>(bool);
            double re = 0;
            //Ira percorrer a lista de bases do vetor vb
            //ate achar o argumento passado para essa funcao (variavel bool)
            for(int i = 0; i < bb.size(); i++){
               //Se encontrou a posicao desejado calcula e para o laco
               if ( bool.toString().equals(bb.get(i).toString())){
                  //A monada Vtemp contem o vetor que representa a transformacao
                  //da posicao desejada (variavel i)
                  //Logo, ele que sera aplicadado nos coeficientes do vetor V
                  Monad<B> vtemp = f.invoke(ba.get(i));
                  //Percorre os dois vetor e executa a transformacao
                  for(int j = 0; j < ba.size(); j++){
                     re += vtemp.vec.invoke(bb.get(j)) * v.vec.invoke(ba.get(j));
                  }
                  break;
               }
            }
            //Retorna a variavel re
            re
         }
     );
   }

   //Metodo que implementa o algoritmo quantico
   public static <A> Monad<A> quantum_teleport(Monad<A> Alice, Monad<A> Bob){
```

```
    //Aplica a transforma'c~oes Cnot
    Alice = bind(Alice, make_Cnot(Alice));
    //Aplica Hadamard apenas no primeiro qubit
    Alice = bind(Alice, make_Hadamard(Alice));
    //Realiza a medida nos dois primeiros qubits
    A medida = medida(Alice);
    //Procura o resultado da medida
    if (A.toString().equals("false,false"){
        //NÃ£o faz nada
    }
    else if (A.toString().equals("false,true"){
        //Aplica a porta X
        Bob = bind(Bob, make_not(Bob));
    }
    esle if (A.toString().equals("true,false"){
        //Aplica a porta Z
        Bob = bind(Bob, make_pauli(Bob));
    }
    else
        //Aplica a porta Z, e depois X
        Bob = bind(Bob, make_pauli(Bob));
        Bob = bind(Bob, make_not(Bob));
    }
    return Bob;
}

//Soma dois vetores ba'sicos
//Retorna um vetor com os coeficentes referente a soma
//dos coeficientes de cada base normalizados
public static <A> Monad<A> soma (Monad<A> v1, Monad<A> v2){
    return new Monad<A>(v2.base.get(0), { A bool =>
        double re = 0;
        re = (v1.vec.invoke(bool) + v2.vec.invoke(bool));
        //Retorna a variavel re
        re
     });
}

//Subtrai dois vetores ba'sicos
//Retorna um vetor com os coeficentes referente a soma
//dos coeficientes de cada base normalizados
public static <A> Monad<A> sub (Monad<A> v1, Monad<A> v2){
    return new Monad<A>(v2.base.get(0), { A bool =>
        double re;
        re = (v1.vec.invoke(bool) - v2.vec.invoke(bool));
        re
     });
}

//Normaliza o valor do coeficiente
//deve ser usada apos uma soma ou subtracao
public static <A> Monad<A> normaliza (Monad<A> v, int i){
    return new Monad<A>(v.base.get(0), { A bool =>
        double re;
        re = v.vec.invoke(bool);
        re = re / Math.sqrt(i);
        //Retorna a variavel re
        re
     });
}

//Cria uma transformacao NOT dinamicamente
//Retorna um vetor com o tamanho do tipo de base
public static <A> {A => Monad<A> } make_not(Monad<A> v){
    return { A bool =>
        Basis<A> ba = v.base;
        Monad<A> re = null;
        for (int i = 0; i < ba.size(); i++){
            if (bool.toString().equals(ba.get(i).toString())){
                re = new Monad<A>(ba.get(ba.size()-1-i));
                break;
            }
        }
        //Retorna a variavel re
        re
    };
}

//Cria uma transformacao CNOT dinamicamente
//Retorna um vetor com o tamanho do tipo de base
public static <A> {A => Monad<A>} make_Cnot(Monad<A> v){
    return {A bool =>
        Basis<A> ba = v.base;
        Monad<A> re = null;
        for (int i = ba.size()/2; i < ba.size(); i++){
            if (bool.toString().equals(ba.get(i).toString())){
                re = new Monad<A>(ba.get(ba.size()-(i-1)));
                break;
            }
        }
        if (re == null)
            re = new Monad<A>(bool);
        //Retorna a variavel re
        re
    };
}

public static { Pair => Monad<Pair>} make_Hadamard2(){
    return { Pair bb =>
```

```
        Monad<Pair> re = null;
        Monad<Pair> base00 = new Monad<Pair>(new Pair(false,false));
        Monad<Pair> base01 = new Monad<Pair>(new Pair(false,true));
        Monad<Pair> base10 = new Monad<Pair>(new Pair(true,false));
        Monad<Pair> base11 = new Monad<Pair>(new Pair(true,true));

        if (bb.getA().toString().equals("false") && bb.getB().toString().equals("false"))
           re = soma(soma(base00,base01),soma(base10,base11));
        else if (bb.getA().toString().equals("false") && bb.getB().toString().equals("true"))
           re = sub( soma(base00,base10), soma(base01,base11) );
        else if (bb.getA().toString().equals("true") && bb.getB().toString().equals("false"))
           re = sub( soma(base00,base01), soma(base10,base11) );
        else
           re = sub( soma(base00,base11), soma(base01,base10) );
        normaliza(re,re.base.size())
     };
  }


  //Realiza a medida no vetor passado
  //fazendo o estado entrar em colapso
  //apo's a medida
  public static <A> A medida(Monad<A> v){
     Random generator = new Random();
     int i;
     //Pega um numero aleatorio
     double temp = generator.nextDouble();
     double probabilidade;
     Basis<A> ba = v.base;
     double comp = 0;
     double comp2 = 0;
     //Procura um intervalo que contenha o numero sorteado
     //com os valores do vetor quantico
     for (i = 0; i < ba.size(); i++){
        probabilidade = Math.pow(v.vec.invoke(v.base.get(i)),2);
        comp2 += probabilidade;
        if (comp < temp && temp < comp2 )
           break;
        else
           comp += probabilidade;
     }
     //Retorna um valor referente a um bit classico
     return ba.get(i);
  }


  public static void main(String[] args) {

     /*-------------------------------------------
     |       Exemplo de uso de um estado de 1 qbits
      -------------------------------------------*/

     Monad<Boolean> soma = soma(new Monad<Boolean>(false),new Monad<Boolean>(true));
     soma = normaliza(soma,soma.base.size());
     System.out.print("Soma das da superposicao das Base 0 e 1: ");
     System.out.println((Math.pow(soma.vec.invoke(false),2) + Math.pow(soma.vec.invoke(true),2)));

     System.out.println( medida(soma));

     Basis<Boolean> a = new Basis<Boolean>(0);
     Monad<Boolean> not = bind(base1, make_not(base1));

     System.out.println("Not na base 1");
     for (int i=0; i<a.size(); i++)
     {
         System.out.println(not.vec.invoke(a.get(i)));
     }

     /*-------------------------------------------
     |       Exemplo de uso de um estado de 2 qbits
     -------------------------------------------*/
     Basis<Pair> b = new Basis<Pair>(new Pair());

     Monad<Pair> base00 = new Monad<Pair>(new Pair(false,false));
     Monad<Pair> base01 = new Monad<Pair>(new Pair(false,true));
     Monad<Pair> base11 = new Monad<Pair>(new Pair(true,true));
     Monad<Pair> base10 = new Monad<Pair>(new Pair(true,false));

     //Cria um estado emarranhado de dois qubits
     Monad<Pair> bell = bind(base00, make_H3());
     bell = bind(bell,make_Cnot(bell));
     System.out.println(medida(bell));
  }
}
```

A.2 Classes Auxiliares

Listing A.2: Classe Basis<A>

```
import java.util.ArrayList;

//Class que manipula a lista de bases do vetor
```

```
public class Basis<A>{
   //Lista das bases
   ArrayList<A> basis;

   Basis(Object base){
      basis = new ArrayList<A>();
      //Cria as listas dinamicamente
      //verificando o tamanho do argumento passado
      String[] split = base.toString().split(",");
      //Verifica se Ã© um tipo basico
      if (split.length == 1){
         //Tipo basico, logo existem apenas dois tipos de base
         basis.add((A) Boolean.FALSE);
         basis.add((A) Boolean.TRUE);
      }
      //Verifica se Ã© uma tupla<Boolean,Boolean>
      else if (split.length == 2){
         //Tipo Boolean,Boolean
         //quatro possiveis valores
         Pair p1 = new Pair(false,false);
         Pair p2 = new Pair(false,true);
         Pair p3 = new Pair(true,false);
         Pair p4 = new Pair(true,true);
         basis.add( (A) p1);
         basis.add( (A) p2);
         basis.add( (A) p3);
         basis.add( (A) p4);
      }
      //Verifica se Ã© uma tupla<Boolean,Pair<Boolean,Boolean>>
      else if (split.length == 3){
         //Tipo Boolean,Boolean,Boolean
         //oito possiveis valores
         Pair p1 = new Pair(false,new Pair(false,false));
         Pair p2 = new Pair(false,new Pair(false,true));
         Pair p3 = new Pair(false,new Pair(true,false));
         Pair p4 = new Pair(false,new Pair(true,true));
         Pair p5 = new Pair(true,new Pair(false,false));
         Pair p6 = new Pair(true,new Pair(false,true));
         Pair p7 = new Pair(true,new Pair(true,false));
         Pair p8 = new Pair(true,new Pair(true,true));
         basis.add( (A) p1);
         basis.add( (A) p2);
         basis.add( (A) p3);
         basis.add( (A) p4);
         basis.add( (A) p5);
         basis.add( (A) p6);
         basis.add( (A) p7);
         basis.add( (A) p8);
      }
   }

   //Metodo auxiliar que retorna a posicao da lista
   public A get(int i){
      return basis.get(i);
   }

   //Metodo auxiliar que retorna o tamanho da lista
   public int size(){
      return basis.size();
   }
}
```

Listing A.3: Classe Pair

```
//Class que auxilia a criacao de tuplas de um tipo generico
public class Pair<A,B> {
   public  final A a;
   public  final  B b;

   //Construtor defeault
   public Pair(){
      this.a = null;
      this.b = null;
   }

   //Construtor com passagem de argumento
   public Pair(final A a, final B b){
      this.a = a;
      this.b = b;
   }

   //Pega o valor de A
   public A getA(){
      return this.a;
   }

   //Pega o valor de B
   public B getB(){
      return this.b;
   }

   //Metodo que converte o conteudo da classe em um string padrao
   //permite que sejam comparados instancias da classe Pair
   //mesmo sendo de tipo genericos diferentes
   @Override
   public String toString(){
```

```
        return a + "," + b;
    }
}
```