COMPUTAÇÃO QUÂNTICA E TEORIA DE COMPLEXIDADE

CARLOS HENRIQUE CARDONHA

Departamento de Ciência da Computação

Instituto de Matemática e Estatística

Universidade de São Paulo

Resumo. O estudo de computabilidade e complexidade computacional foi sempre inicialmente motivado e impulsionado por questões matemáticas fundamentais. Nos anos 80, foi introduzido o modelo quântico de computação que, na década passada, se mostrou especialmente adequado para resolver eficientemente problemas matemáticos como o da fatoração de inteiros e o do cálculo do logaritmo discreto de um número. Apresentamos o modelo quântico de computação e algumas relações entre as classes de complexidade clássicas **P**, **NP**, **BPP** e outras com as novas classes de complexidade baseadas no modelo quântico.

(Este trabalho está sendo desenvolvido junto com Marcel K. de Carli Silva, sob a orientação da Profa. Cristina Gomes Fernandes (IME-USP). O Marcel também fez uma submissão às Jornadas. A parte do trabalho submetida por ele complementa a nossa, apresentando o resultado algorítmico de maior relevância na área: o algoritmo de Shor para fatoração de inteiros.)

1. Introdução

Em 1900, em uma palestra marcante no Congresso Internacional de Matemáticos realizado em Paris, Hilbert postulou 23 problemas matemáticos, que tratam de temas diversos em matemática e áreas afins. O décimo problema na lista de Hilbert (*determination of the solvability of a diophantine equation*) pergunta se é possível determinar se uma equação diofantina arbitrária tem ou não solução por meio de um "processo finito":

> Given a diophantine equation with any number of unknown quantities and with rational numerical coefficients: to devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.

Date: Setembro de 2004.

Financiado parcialmente pela FAPESP 03/13236-0.

Esse problema pode ser postulado em uma linguagem mais atual como o seguinte: existe um algoritmo que, dada uma equação diofantina, determina se esta tem ou não solução?

Note que a questão postulada por Hilbert precede de décadas a invenção de computadores. Foi apenas nos anos 30 que tais questões foram formuladas e tratadas dentro do que ficou depois conhecido como *teoria da computabilidade*. Esta é a parte da teoria da computação especializada em lidar com esse tipo de questão.

Foi nos anos 30, após um trabalho de Gödel em lógica, que a idéia de algoritmo começou a ser formalizada. Gödel [Göd31] introduziu o conceito de *função primitiva recursiva* como uma formalização dessa idéia. Church [Chu33, Chu36] introduziu o λ-*cálculo* e Kleene [Kle36] definiu o conceito de *funções recursivas parciais* e mostrou a equivalência entre esse e o λ-cálculo. Turing [Tur36, Tur37] por sua vez propôs a sua formalização da idéia de algoritmo: as chamadas *máquinas de Turing*. Nesses trabalhos, Turing mostrou também a equivalência do conceito de máquinas de Turing e de funções recursivas parciais de Church. Vale mencionar que o conceito de máquinas de Turing foi independentemente proposto por Post [Pos36], um professor de colegial de Nova Iorque. Cada uma dessas propostas diferentes do conceito de algoritmo é chamada de *modelo de computação*.

Foi Kleene [Kle52] quem chamou de *tese de Church* a afirmação de que todo modelo de computação *razoável* é equivalente ao da máquina de Turing. A afirmação é propositalmente vaga, pois visa capturar mesmo modelos que ainda venham a ser propostos, e cuja natureza não podemos prever. Por *razoável* entende-se um modelo que seja realista, no sentido de poder (mesmo que de maneira aproximada) ser construído na prática.

A teoria da computabilidade no fundo diferencia os problemas *decidíveis* (para os quais existe um algoritmo) dos *indecidíveis* (para os quais não existe um algoritmo). O surgimento dos computadores nas décadas de 30 e 40 aos poucos evidenciou uma diferença entre os problemas decidíveis: muitos parecem ser bem mais difíceis que outros, no sentido de que se conhece apenas algoritmos extremamente lentos para eles. Com isso, surgiu a necessidade de refinar a teoria de computabilidade para tentar explicar essas diferenças. Foi apenas nos anos 60 que a *teoria de complexidade*, que trata de tais questões, tomou corpo, com a formalização da idéia de "algoritmo eficiente", independentemente introduzida por Cobham [Cob65] e Edmonds [Edm65], e a proposta de reduções eficientes entre problemas [Kar72].

Foi nessa época que surgiram as definições das classes de complexidade **P** e **NP** e do conceito de **NP**-*completude*, que captura de certa maneira a dificuldade de se conseguir algoritmos eficientes para certos problemas. Grosseiramente, um problema em **NP** é dito **NP**-*completo* se qualquer outro problema da classe **NP** pode ser reduzido eficientemente a ele. A mais famosa questão

na área de teoria da computação é se P é ou não igual a **NP**. Se for mostrado que algum problema **NP**-completo está em **P**, então tal questão é resolvida e fica provado que **P** = **NP**.

Um marco na teoria de complexidade é o *teorema de Cook* [Coo71, Lev73], que prova a existência de problemas **NP**-completos. Cook mostrou que o problema conhecido como SAT, de decidir se uma fórmula booleana em forma normal conjuntiva é ou não satisfatível, é **NP**-completo. Após o teorema de Cook e o trabalho de Karp [Kar72], que mostrou que vários outros problemas conhecidos de otimização combinatória eram **NP**-completos, essa teoria se desenvolveu amplamente, tendo estabelecido a dificuldade computacional de problemas das mais diversas áreas [GJ79].

Um problema muito famoso cuja complexidade continua em aberto, mesmo após várias décadas de esforço da comunidade no sentido de resolvê-lo, é o problema da *fatoração de inteiros*: dado um inteiro, determinar a sua fatoração em números primos. Recentemente, o seu parente próximo, o problema de decidir se um número inteiro é primo ou não, chamado de *problema da primalidade*, teve sua complexidade totalmente definida, com o algoritmo de Agrawal, Kayal e Saxena (AKS) [AKS02], que é aliás o assunto de um dos mini-cursos nesse colóquio. O algoritmo de AKS mostra que o problema da primalidade está na classe **P**, resolvendo com isso uma questão em aberto há anos. Não se sabe até hoje, no entanto, se há um algoritmo eficiente para resolver o problema da fatoração de inteiros!

Na verdade, a dificuldade computacional do problema da fatoração de inteiros tem sido usada de maneira crucial em alguns sistemas criptográficos bem-conhecidos. Se for descoberto um algoritmo eficiente para resolver o problema da fatoração, vários sistemas criptográficos importantes seriam quebrados, incluindo o famoso sistema RSA de chave pública.

O assunto de nossa iniciação científica — Computação Quântica — trata de um novo modelo de computação, o modelo quântico, que vem levantando questões intrigantes dentro da teoria de complexidade, e pode ter impactos práticos dramáticos no mínimo na área de criptologia. O modelo quântico de computação não infringe a validade da tese de Church, porém questiona a validade de uma versão mais moderna dessa, a chamada *tese de Church estendida*, que diz que todo modelo de computação razoável pode ser simulado *eficientemente* por uma máquina de Turing.

Pode-se dizer que a teoria de computação quântica iniciou-se nos anos 80, quando Feynman [Fey82] observou que um sistema quântico de partículas, ao contrário de um sistema clássico, parece não poder ser simulado eficientemente em um computador clássico e sugeriu um computador que explorasse efeitos da física quântica para contornar o problema. Desde então, até 1994, a teoria de computação quântica desenvolveu-se discretamente, com várias contribuições de

Deutsch [Deu85, Deu89], Bernstein e Vazirani [BV97], entre outros, que colaboraram fundamentalmente para a formalização de um modelo computacional quântico.

Foi apenas em 1994 que a teoria recebeu um forte impulso e uma enorme divulgação. Isso deveu-se ao algoritmo de Shor [Sho94, Sho97], um algoritmo quântico eficiente para o problema da fatoração de inteiros, considerado o primeiro algoritmo quântico combinando relevância prática e eficiência. O algoritmo de Shor é uma evidência de que o modelo computacional quântico proposto pode superar de fato o modelo clássico, derivado das máquinas de Turing. O resultado de Shor impulsionou tanto a pesquisa prática, objetivando a construção de um computador segundo o modelo quântico, quanto a busca por algoritmos criptográficos alternativos e algoritmos quânticos eficientes para outros problemas difíceis. Essas e várias outras questões, relacionadas tanto com a viabilidade do modelo quântico quanto com as suas limitações, têm sido objeto de intensa pesquisa científica.

Do ponto de vista prático, busca-se descobrir se é ou não viável construir um computador segundo o modelo quântico que seja capaz de manipular números suficientemente grandes. Tal viabilidade esbarra em uma série de questões técnicas e barreiras físicas e tecnológicas. Já se tem notícia de computadores construídos segundo o modelo quântico, mas todos ainda de pequeno porte. Em 2001, por exemplo, foi construído um computador quântico com 7 *qubits* (o correspondente aos bits dos computadores tradicionais). Nesse computador, foi implementado o algoritmo de Shor que, nele, fatorou o número 15. Uma parte dos cientistas da computação acredita que a construção de computadores quânticos de maior porte será possível, enquanto outra parte não acredita nisso.

Do ponto de vista de teoria de complexidade, busca-se estabelecer a relação entre as classes de complexidade derivadas do modelo quântico e as classes de complexidade tradicionais. Também busca-se, claro, estabelecer a complexidade no modelo quântico de problemas bem-conhecidos, ou seja, busca-se por algoritmos quânticos eficientes para outros problemas relevantes.

Nesse trabalho, apresentamos os modelos mais tradicionais de computação e o modelo quântico, definimos as principais classes de complexidade clássicas e quânticas, e apresentamos alguns resultados de complexidade envolvendo tais classes. A parte do nosso trabalho de iniciação científica submetida por Marcel K. de Carli Silva mostra o resultado algorítmico mais relevante na área de computação quântica: o algoritmo de Shor.

Esperamos que estes dois trabalhos dêem uma visão do que é esta área nova e intrigante, e das suas potencialidades e dificuldades. O tema é multidisciplinar, no sentido de que depende de uma série de conceitos da mecânica quântica, e empresta a notação usada nessa área, o que dificulta um pouco a

apresentação da área para pessoas de outras áreas, como computação e matemática. Um texto mais completo que estamos preparando dentro dessa iniciação científica, com esses resultados e outros, pode ser encontrado no endereço `http://www.ime.usp.br/~magal/quantum/`.

2. Máquinas de Turing

Nesse seção, apresentamos o mais tradicional modelo de computação — a máquina de Turing (MT). Uma máquina de Turing nada mais é que um computador bastante rudimentar com memória infinita. Vamos apresentar três variantes desse modelo: a máquina de Turing determinística, a não-determinística e a probabilística. Depois disso, apresentamos o modelo quântico de computação através da máquina de Turing quântica, fazendo um paralelo com o modelo clássico de computação.

2.1. **Máquina de Turing determinística.**

Uma máquina de Turing determinística (MTD) é composta por uma central de controle, uma cabeça de leitura e por uma fita dividida em células. Essa fita tem um final à esquerda e contém infinitas células à direita.

Em cada uma das células da fita é armazenado um símbolo pertencente a um conjunto finito Σ. O conjunto Σ é chamado de *alfabeto* da máquina e contém, entre outros, dois símbolos especiais: o símbolo $\sqcup$, chamado de *branco*, e o símbolo $\triangleright$, que fica armazenado o tempo todo na célula mais à esquerda da fita.

A cabeça de leitura da máquina pode ser vista como um apontador móvel para uma determinada célula da fita. O conteúdo dessa célula pode ser lido e alterado pela máquina.

A cada instante, a central de controle da máquina está em um dos estados de um conjunto finito Q de estados. No instante inicial, a máquina começa em um estado particular de Q, chamado de estado *inicial* e denotado usualmente por s. Existe ainda um conjunto especial de estados $H \subseteq Q$, chamados de estados *finais*, cujo papel será explicado mais adiante.

Uma MTD funciona da seguinte maneira. Ela recebe como entrada uma cadeia de caracteres em $\big(\Sigma \setminus \{\sqcup, \triangleright\}\big)^*$. Inicialmente a cabeça de leitura aponta para a célula mais à esquerda da fita, e a partir da célula seguinte está armazenada a entrada da máquina, seguida por brancos. A máquina efetua uma seqüência de passos até terminar a execução. Se q é o estado corrente da máquina e q está em H, então ela termina a execução. Do contrário, ela realiza três ações, determinadas pelo par (q, σ), onde σ é o símbolo de Σ contido na célula que está sob a cabeça de leitura.

A primeira ação consiste na alteração do estado da máquina de q para um estado q' em Q. A segunda ação é a escrita de um símbolo σ' na célula que está

sob a cabeça de leitura. A terceira ação consiste no deslocamento da cabeça de leitura, que pode deslocar-se uma posição para a esquerda, uma posição para a direita ou pode continuar apontando para a mesma célula.

A cabeça de leitura da fita sempre desloca-se para a direita quando atinge a célula mais à esquerda e nunca altera o conteúdo dessa posição da fita. Por conveniência, assume-se que a máquina não pode escrever o símbolo $\triangleright$ em qualquer posição da fita que não seja a célula mais à esquerda.

A cadeia de caracteres escrita na fita quando a máquina termina a execução, ignorando-se o símbolo $\triangleright$ e os brancos à direita, é a saída da máquina para a entrada em questão.

Formalmente, define-se uma máquina de Turing determinística como uma quíntupla $M = (Q, \Sigma, \delta, s, H)$, onde Q é o conjunto de estados, Σ é o alfabeto de símbolos, δ é uma função de transição de $(Q \setminus H) \times \Sigma$ em $Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}$, $s \in Q$ é o estado inicial e $H \subseteq Q$ é o conjunto de estados finais. O conjunto $\{\leftarrow, \downarrow, \rightarrow\}$ descreve os possíveis movimentos da cabeça de leitura da máquina.

A função δ descreve um passo arbitrário da máquina e satisfaz uma série de restrições correspondentes às mencionadas acima. Suponha que a máquina esteja num estado q em $Q \setminus H$ e, sob a cabeça de leitura, esteja o símbolo σ. Se $(q, \sigma) = (q', \sigma', d)$, o próximo estado será q', o símbolo σ' será escrito no lugar de σ e a cabeça de leitura de M moverá de acordo com d.

A *configuração atual* de M é uma tripla (w_1, q, w_2), onde $w_1, w_2 \in \Sigma^*$ e $q \in Q$. A relação da tripla com M é que w_1 é a palavra à esquerda da cabeça de leitura, q é o estado atual e w_2 é a palavra à direita da cabeça de leitura ignorando-se brancos à direita. A cabeça de leitura aponta para a posição que contém o primeiro símbolo de w_2.

Dizemos que uma configuração (w_1, q, w_2) *produz em k passos* uma configuração (w_1', q', w_2'), o que é denotado por $(w_1, q, w_2) \vdash_M^k (w_1', q, w_2')$, se a máquina sai da primeira configuração e vai para a segunda em exatos k passos.

2.2. **Máquinas de Turing não-determinísticas.** A máquina de Turing não-determinística (MTND) é uma generalização da máquina de Turing determinística. Esse modelo tem um papel fundamental na teoria de complexidade pois está na base da definição da classe **NP**.

A maior parte das definições e idéias envolvidas na descrição das MTDs também se aplica às MTNDs. A diferença entre a MTND e a MTD está na função de transição e na maneira como as transições são feitas a cada passo. Nas máquinas determinísticas, existe uma única transição possível a partir de uma dupla (q, σ), onde q é um estado não-final e σ é um símbolo em Σ. Já nas máquinas não-determinísticas, pode existir mais de uma transição válida a partir de uma dupla (q, σ). Em cada passo da máquina, uma transição válida é escolhida arbitrariamente e é executada.

O número de configurações que podem ser produzidas a partir de cada configuração em $\Sigma^* \times Q \times \Sigma^*$ é limitado superiormente por $|\Sigma| \times |Q| \times 3$.

A existência de várias transições possíveis para cada par (q, σ) permite a ocorrência de computações distintas para uma MTND para uma mesma entrada. A transição numa MTND é portanto uma *relação* e não necessariamente uma função.

Formalmente, uma máquina de Turing não-determinística é uma quíntupla $M = (Q, \Sigma, \Delta, s, H)$, onde Q é o conjunto de estados, Σ é o alfabeto de símbolos, Δ é uma relação de transição de $(Q \setminus H) \times \Sigma$ em $Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}$, $s \in Q$ é o estado inicial e $H \subseteq Q$ é o conjunto de estados finais.

É evidente que as MTDs são casos particulares de MTNDs, em que Δ é uma função (toda função é uma relação).

Uma transição (q', σ', d) em $\Delta(q, \sigma)$, onde $q \in Q$ e $\sigma \in \Sigma$, é interpretada como antes. Nela, $q' \in Q$, $\sigma' \in \Sigma$ e $d \in \{\leftarrow, \downarrow, \rightarrow\}$. Assim, q' será o próximo estado da máquina, σ' deve ser escrito no lugar de σ e d indicará o deslocamento da cabeça de leitura. Tal transição será válida somente se $(q', \sigma', d) \in \Delta(q, \sigma)$. A definição de configuração é igual à que usamos na definição da máquina de Turing determinística. Para MTND, dizemos que uma configuração (w_1, q, w_2) produz a configuração (w_1', q', w_2') em k passos, o que é denotado por $(w_1, q, w_2) \vdash_M^k (w_1', q', w_2')$, se, estando a máquina na primeira configuração, após k passos válidos ela pode estar na segunda a partir de uma seqüência de transições válidas.

Observe que, como as MTNDs podem ter várias computações válidas possíveis para uma mesma entrada, ela pode produzir saídas diferentes para uma mesma entrada. Comentaremos mais sobre isso quando falarmos das linguagens decididas por uma máquina de Turing.

2.3. **Máquina de Turing probabilística.** Uma máquina de Turing probabilística (MTP) é uma máquina de Turing não-determinística $M = (Q, \Sigma, \Delta, s, H)$ que funciona de maneira diferente.

A cada passo, em vez de uma transição aplicável ser escolhida arbitrariamente, dentre todas as transições aplicáveis, escolhe-se uma com probabilidade uniforme. Assim, pode-se falar na probabilidade da MTP produzir, numa computação para uma certa entrada, uma saída específica. Todas as definições dadas para as MTNDs aplicam-se de maneira natural a uma MTP. Observe que uma MTD é uma MTP em que, a cada passo, há apenas uma transição aplicável.

Uma MTP corresponde à formalização do conceito de um computador acoplado a um gerador de números aleatórios.

É possível descrever o funcionamento de uma MTP postergando as escolhas aleatórias para o final. Ou seja, pode-se calcular a probabilidade de se estar em

uma configuração específica a cada passo e só ao final fazer um único sorteio para determinar a configuração em que terminamos.

2.4. **Máquina de Turing quântica.** A definição de uma máquina de Turing quântica (MTQ) assemelha-se à de uma MTP. Uma máquina de Turing quântica é uma MTND $M = (Q, \Sigma, \Delta, s, H)$ com a relação Δ substituída por uma função

$$\alpha : Q \times \Sigma \times Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\} \longrightarrow \mathbb{C}_{[0,1]}$$

tal que, para cada (q, σ) em $Q \times \Sigma$, vale que

$$\sum_{(q,\sigma,d) \in Q \times \Sigma \times \{\leftarrow,\downarrow,\rightarrow\}} |\alpha(q_1, \sigma_1, q, \sigma, d)|^2 = 1.$$

Cada número $\alpha(q_1, \sigma_1, q, \sigma, d)$ é chamado de *amplitude*. Note que α determina uma distribuição de probabilidade nas possíveis transições aplicáveis a um par (q, σ).

Define-se *superposição de configurações* como uma combinação linear de configurações, onde o coeficiente de cada termo é um número complexo e a soma dos quadrados dos módulos de todos os coeficientes é igual a um. Tal coeficiente é a amplitude da respectiva configuração.

A primeira diferença no funcionamento de uma MTQ frente às máquinas anteriores é que esta, a cada instante, encontra-se numa superposição de configurações. Se todas as configurações da MTQ com amplitude não-nula forem finais, a MTQ termina a execução. Caso contrário, ela efetua uma transição. A transição consiste no seguinte.

Digamos que $s = \sum_{j=1}^{t} \alpha_j c_j$ é a superposição corrente, onde $\sum_{j=1}^{t} |\alpha_j|^2 = 1$ e $\alpha_j \neq 0$ para todo j. Para cada j, seja C_j o conjunto das configurações que podem ser obtidas de c_i em um passo por meio de uma transição cuja amplitude é não-nula. Para cada c em C_j, seja α_c^j a amplitude correspondente à transição de c_j para c. Então, temos que a superposição resultante da transição é $s' = \sum_{j=1}^{t} \alpha_j \sum_{c \in C_j} \alpha_c^j c$. Quando uma mesma configuração c aparece em mais do que um conjunto C_j e $|\sum_{j:c \in C_j} \alpha_j \alpha_c^j|^2 \neq \sum_{j:c \in C_j} |\alpha_j \alpha_c^j|^2$, dizemos que houve *interferência*. A interferência é *negativa* se o lado esquerdo é menor que o lado direito e *positiva* caso contrário. O fenômeno de interferência é o principal responsável pela diferença entre uma MTQ e uma MTP.

Para que a transição de uma MTQ esteja bem definida, é preciso que a máquina satisfaça a seguinte propriedade: se, numa superposição obtida depois de k passos a partir da configuração inicial, há uma configuração com amplitude não-nula num estado final, então todas as configurações com amplitude não-nula na superposição estão num estado final.

A segunda diferença é que, ao final de sua execução, a MTQ efetua o que chamamos de *medição*. Digamos que $s = \sum_{j=1}^{t} \alpha_j c_j$ é a superposição final da

máquina, com $\sum_{j=1}^{t} |\alpha_j|^2 = 1$ e $\alpha \neq 0$ para todo j. Uma medição consiste na escolha aleatória de uma configuração $c = c_j$ com probabilidade $|\alpha_j|^2$, e na transição da superposição s para a superposição $s' = c$, ou seja, para a superposição em que apenas a configuração c tem amplitude não-negativa, ou, mais exatamente 1. A saída da MTQ é o que está escrito na fita após a medição.

A capacidade da MTQ estar numa superposição de configurações e, num passo, efetuar transições múltiplas, envolvendo diversas configurações é chamada de *paralelismo quântico*. Potencialmente é possível explorar o fenômeno de interferência bem como o paralelismo quântico para se obter algoritmos quânticos eficientes.

3. Modelo clássico de computação

Nessa seção, serão apresentados resultados e definições do modelo clássico de computação com o intuito de estabelecer um paralelo durante a apresentação dos correspondentes quânticos desses resultados e definições.

3.1. **Máquina de Turing determinística universal.** Na seção anterior, definimos as máquinas de Turing determinísticas. Em geral, as MTDs são modeladas para que resolvam um único problema. Porém, os computadores atuais não estão restritos a uma única tarefa.

Vamos mostrar como construir uma máquina de Turing capaz de executar várias tarefas. Mais precisamente, tal MTD será capaz de simular qualquer MTD (o que não é restritivo, pois toda MTND e toda MTP pode ser simulada por uma MTD). A descrição baseia-se na apresentação de Papadimitriou [Pap94].

Seja U a máquina de Turing universal que vamos construir. Essa máquina lê uma entrada e a interpreta como sendo a descrição de uma máquina de Turing M e uma entrada x para essa máquina. A máquina U deve funcionar de modo que, após sua parada, sua saída seja a mesma que M devolve ao ser executada tendo x como entrada.

Vamos determinar agora o padrão de entrada a ser reconhecido por U. Seja $M = (K, \Sigma, \delta, s, H)$ a máquina que será simulada, e seja x a entrada na qual a execução de M deverá ser simulada.

Cada símbolo de Σ e cada estado de Q será descrito como um número na forma binária. Os símbolos de Σ serão representados por números em $\{1, 2, \ldots, |\Sigma|\}$, os estados de K por números em $\{|\Sigma|+1, |\Sigma|+2, \ldots, |\Sigma|+|K|\}$ e os números $|\Sigma|+|K|+1, \ldots, |\Sigma|+|K|+5$ representarão os símbolos $\{\leftarrow, \downarrow, \rightarrow\}$, $\triangleright$ e $\sqcup$. Todos os símbolos serão representados usando a mesma quantidade de bits (serão usados zeros à esquerda quando necessário). O estado denotado por s é o estado inicial de M, e será representado pelo número $|\Sigma| + 1$.

Na parte da entrada que corresponde a M, inicialmente serão fornecidos como entrada os números $|K|$, $|\Sigma|$ e $|H|$. Em seguida, temos a representação de $|H|$ estados de M, que indicam os estados finais de M. Finalizando a descrição de M, a fita conterá uma descrição de δ, na forma de uma seqüência de $((q, \sigma), (p, \rho, D))$. Vamos assumir sem perda de generalidade que os parênteses e a vírgula pertencem a Σ.

Após a descrição da máquina, a fita de entrada de U vai conter o símbolo ";" (que deverá pertencer ao alfabeto de U), e após esse símbolo estará a descrição da entrada x para M, codificada da maneira descrita. Como cada símbolo e estado é representado com o mesmo número de bits, não há ambigüidades na descrição da máquina ou da entrada.

Tendo a entrada, U pode começar sua execução. Para facilitar a descrição, vamos supor que U utiliza 3 fitas, e que cada fita tem uma cabeça de leitura independente (ou seja, cada cabeça pode estar numa posição diferente das demais). Pode-se provar que uma MTD com uma fita pode simular qualquer MTD com k fitas e com k cabeças de leitura independentes, $k > 1$.

Na primeira fita, U receberá a entrada e devolverá a saída. Na segunda fita, U vai guardar a configuração atual de M. A configuração estará no formato (w, q, u), codificada da maneira que já descrevemos anteriormente. Inicialmente, a configuração inicial será $(\triangleright, s, x)$. A terceira fita guardará a descrição da função de transição δ de M.

Por fim, U deve simular M. Para isso, no início de cada passo, U percorre a segunda fita até descobrir o estado atual de M, localizando assim a posição da cabeça de leitura de M em sua fita. Em seguida, U percorre a terceira fita, procurando uma transição que tenha q como estado do domínio, onde q é o estado atual. Feito isso, U move a cabeça de leitura para a esquerda na segunda fita para identificar o símbolo σ que está sob a cabeça de leitura de M e verificar se a transição que está sendo analisada atualmente tem como domínio o par (q, σ). Se não for esse o par, U procura na terceira fita até encontrá-lo.

Após localizar a transição adequada, U faz as modificações na segunda fita, movimentando a cabeça, alterando o símbolo na posição que estava anteriormente e mudando o estado atual, conforme a descrição da função de transição. Se o estado atingido pertence a H, a máquina M pára. Caso contrário o processo se repete.

Na parada, o conteúdo da segunda fita de U será igual à palavra que M devolve após executar tendo x como entrada. Assim, U copia o conteúdo da segunda fita para a primeira fita e termina sua execução.

A construção dessa máquina utilizou uma série de recursos e ferramentas que facilitaram bastante nossa tarefa: as fitas múltiplas, o movimento independente das cabeças de leitura dessas fitas, etc. Nem todas essas ferramentas podem ser utilizadas no modelo quântico. Além disso, outros cuidados devem

ser tomados, e isso acaba resultando num aumento significativo da complexidade da correspondente construção.

3.2. **Principais classes de complexidade.** Vamos falar agora sobre as classes de complexidade mais importantes no modelo clássico de computação. Novamente vamos nos basear na abordagem feita por Papadimitriou [Pap94].

Para isso, definiremos o significado de tempo e espaço consumido nas máquinas do modelo clássico. Em seguida, vamos comentar a respeito de linguagens decididas por cada uma dessas máquinas, e depois disso definiremos as principais classes do modelo clássico de computação.

Tempo consumido pelas máquinas de Turing. O *tempo consumido* por uma máquina de Turing para uma entrada x é o número de passos que ela executa para ir da configuração inicial até uma configuração final (configuração cujo estado é final).

Um conjunto de máquinas de Turing importante na definição das principais classes de complexidade é o das máquinas *polinomialmente limitadas*. Dizemos que uma máquina de Turing M é polinomialmente limitada se há um polinômio $p(n) : \mathbb{N} \longrightarrow \mathbb{N}$ tal que a seguinte afirmação é verdadeira: para qualquer entrada x, não há configuração C tal que $(\triangleright, s, \underline{\sqcup} x) \vdash_M^{p(|x|)+1} C$. Ou seja, durante a computação de qualquer palavra de tamanho n, a máquina M pára em não mais que $p(n)$ passos.

Nas MTDs, é fácil descrever a contagem de tempo, pois dadas uma entrada e uma máquina, a execução consumirá sempre o mesmo número de passos. Assim, uma MTD polinomialmente limitada é uma máquina que resolve um problema consumindo tempo polinomial no tamanho da entrada para todas as entradas válidas.

No caso das MTNDs, esse conceito é um pouco mais obscuro, pois dadas uma entrada e uma máquina existem várias computações distintas, que podem consumir quantidades diferentes de tempo. Dizemos que uma MTND é polinomialmente limitada se existe um polinômio que limita o número de passos para qualquer computação da máquina. Ou seja, toda computação válida deve consumir tempo polinomial no tamanho da entrada.

Por fim, as MTPs podem seguir o mesmo esquema das MTNDs, onde há uma limitação polinomial para todas as computações possíveis. Porém, em algumas classes, é mais interessante limitar polinomialmente a esperança do tempo consumido por uma computação. Ou seja, algumas classes de complexidade exigem apenas que o tempo esperado seja polinomial, permitindo assim a ocorrência de entradas e computações que consumam tempo não-polinomial.

Espaço consumido pelas máquinas de Turing. Em geral, o espaço consumido por máquinas de Turing não tem a mesma importância que o tempo. Isso porque

as limitações no tempo são mais proibitivas do que as de espaço, refletindo o que acontece na prática atualmente, posto que memória é um recurso cada vez mais barato e os processadores estão encontrando limitações para aumento de desempenho cada vez mais difíceis de serem ultrapassadas.

Porém, existem classes de problemas que são definidas em função do espaço consumido pelas máquinas de Turing que os resolvem. Da mesma forma que no estudo do consumo de tempo, temos interesse especial nos problemas que podem ser resolvidos por máquinas de Turing que consomem espaço polinomial.

As diferenças entre as máquinas de Turing de naturezas distintas não são muito grandes no consumo de espaço. Mais adiante deixaremos claro o porquê dessa afirmação, mostrando que a classe **NP** está contida na classe **PSPACE**. As duas classes serão definidas adiante no texto.

Dizemos que o *espaço consumido* por uma MT é a maior quantidade de células distintas usadas pela máquina para realizar uma computação viável para uma determinada entrada. No caso das MTDs, esse número refere-se as células usadas na única computação possível. Já para MTNDs e MTPs, o valor é o maior número possível de células utilizadas em uma computação válida.

Como toda MT só pode se deslocar de uma célula a cada passo, claramente o espaço consumido em uma computação é limitado superiormente pelo número de passos utilizados pela máquina.

Dizemos que uma MT *consome espaço polinomial* se o espaço consumido pela MT é limitado superiormente por um polinômio no tamanho da entrada.

Linguagens e máquinas de Turing. Ao definir as máquinas de Turing, utilizamos como parâmetro o alfabeto Σ. Esse alfabeto é um conjunto que contém todos os símbolos reconhecidos pela máquina de Turing em questão. Logo, uma entrada válida para a máquina deve ser uma palavra composta por símbolos de $\Sigma \setminus \{\sqcup, \triangleright\}$.

Um conjunto de palavras cujos símbolos pertencem a um alfabeto Σ é chamado de *linguagem*. Na computação, as linguagens são importantes, e algumas possuem propriedades que as tornam especialmente interessantes. Porém, dada qualquer linguagem L, o que geralmente nos interessa é verificar se uma determinada palavra pertence ou não a L.

Se existe uma máquina de Turing M que, dada uma palavra x, responde se x pertence ou não a uma determinada linguagem L, então dizemos que M *decide* a linguagem L. As respostas são devolvidas por M através dos símbolos 0 e 1, que indicam rejeição e aceitação, respectivamente, da palavra de entrada. Esses símbolos devem aparecer sozinhos na fita da máquina após a mesma ter parado, na segunda célula da esquerda para a direita. Uma linguagem é *recursiva* se pode ser decidida por uma máquina de Turing.

É interessante notar que no caso da decisão de linguagens, a resposta das máquinas de Turing é de dois tipos: negação ou aceitação. Por outro lado, em alguns casos queremos fornecer uma entrada para uma máquina de Turing e obter uma saída que não pode ser descrita apenas com duas respostas distintas. Por exemplo, se quisermos construir uma máquina que recebe a representação binária de um número x como entrada e devolve como resposta a representação binária do número $x + 2$, claramente devemos fazer uma máquina capaz de devolver mais do que duas respostas distintas.

Nesse caso, dizemos que estamos lidando com um *problema*, e não com uma linguagem. Rigorosamente, existem diferenças entre problemas e linguagens, mas como podemos transformar uma coisa na outra de maneira razoavelmente simples, vamos falar de problemas e linguagens nesse texto de maneira indistinta.

A partir do conceito de decisão de linguagens, podemos definir o comportamento das máquinas de Turing. No caso das MTDs, o conceito pode ser aplicado de maneira direta. Como para cada máquina e para cada entrada existe uma única computação possível, dizemos que uma MTD M decide uma linguagem L se toda palavra x em L é aceita por M e se toda palavra x fora de L é rejeitada por M.

No caso das MTNDs, já vimos que podem existir várias computações e várias saídas possíveis para uma máquina e uma entrada. Assim, dizemos que uma MTND M decide uma linguagem L se toda palavra $x \in L$ é aceita por M em alguma de suas computações, e se toda palavra $x \notin L$ é rejeitada por M em todas as suas computações. Essa definição acaba mostrando a característica das MTNDs que faz com que elas sejam mais eficientes computacionalmente e menos realistas do que as MTDs.

As classes **P** *e* **PSPACE**. A partir do conceito de máquinas de Turing polinomialmente limitadas, vamos definir as linguagens *polinomialmente decidíveis*. Dizemos que uma linguagem é polinomialmente decidível se há uma máquina de Turing determinística polinomialmente limitada que a decide.

A classe **P** (*polynomial-time*) então é o conjunto de todas as linguagens polinomialmente decidíveis. A classe **P** é extremamente importante, pois contém a maior parte dos problemas que podem ser resolvidos de maneira eficiente. Dizemos que um problema é resolvido de maneira eficiente se existir um algoritmo que o resolve em tempo polinomial no tamanho da entrada.

Como propriedades, temos que **P** é uma classe fechada sob união, intersecção, concatenação e estrela de Kleene. Essas propriedades são importantes e suas provas são interessantes, mas vamos omití-las nesse texto.

De maneira bastante parecida com a que definimos a classe **P**, podemos definir a classe **PSPACE** (*polynomial-space*). Vale ressaltar que, enquanto a primeira faz a classificação das linguagens em função do tempo consumido, a

segunda o faz em função do espaço consumido. Dizemos que uma linguagem pertence à classe **PSPACE** se ela é decidida por uma máquina de Turing determinística que consome espaço polinomial.

É fácil ver que $\mathbf{P} \subseteq \mathbf{PSPACE}$: toda MTD que consome tempo polinomial certamente consome espaço polinomial, já que, a cada passo, a cabeça de leitura da máquina só pode se mover de no máximo uma célula à direita. Por outro lado, o inverso não é verdade. É fácil imaginar uma MTD que consome tempo superpolinomial mas espaço polinomial. Assim, é concebível (e até de se esperar) que existam linguagens em **PSPACE** que não estejam em **P**. Surpreendentemente, não se sabe até hoje se este é o caso ou não, ou seja, não se conhece nenhuma linguagem que esteja em **PSPACE** porém não em **P**.

As classes **NP** *e* **coNP**. As classes **NP** e **coNP** podem ser descritas de maneira parecida com a que descrevemos a classe **P**. Conforme explicaremos adiante, essas classes são muito importantes no estudo da teoria de complexidade clássica.

Vamos descrever **NP** (*nondeterministic polynomial-time*) e **coNP** em função das máquinas de Turing não-determinísticas polinomialmente limitadas. Dizemos que uma linguagem pertence à classe **NP** se ela pode ser decidida por uma máquina de Turing não-determinística polinomilamente limitada. A classe das linguagens cujo complemento pertence a **NP** é denominada **coNP** (o complemento de uma linguagem L sob uma alfabeto Σ é a linguagem $\Sigma^* \setminus L$).

As classes **NP** e **coNP** contêm vários problemas para os quais não se conhece algoritmo polinomial, e formam com a classe **P** a fronteira entre o que pode e o que não pode ser resolvido eficientemente no modelo clássico. Essas classes também são importantes porque contêm uma grande quantidade de problemas de grande interesse prático.

Claramente, toda linguagem que está em **P** também está em **NP** e em **coNP**, visto que uma MTD é uma MTND e é fácil modificá-la numa MTD que reconhece a linguagem complementar. Porém, não sabemos muito mais a respeito da relação entre essas três classes. A questão mais importante e conhecida é se **NP** está contido em **P**. Se isso for verdade, saberemos que muitos problemas para os quais hoje não se conhecem algoritmos exatos razoáveis terão uma solução polinomial. Porém, são poucos os que acreditam nessa hipótese.

NP $\subseteq$ **PSPACE**. Uma relação conhecida entre classes de complexidade que é de certa relevância por não ser óbvia é que **NP** $\subseteq$ **PSPACE**. Isso significa que toda linguagem decidida por uma MTND limitada polinomialmente pode ser decidida por uma MTD que consome espaço polinomial.

Vamos apresentar a prova de maneira razoavelmente informal, pois o que queremos mostrar é a idéia que está por trás da simulação de uma MTND por uma MTD.

Dada uma MTND M polinomialmente limitada que decide uma linguagem L, queremos mostrar que essa máquina pode ser simulada por uma MTD M' que consome espaço polinomial e decide a mesma linguagem L.

A grande diferença entre M e M' está no não-determinismo. Sendo uma MTND, M requer apenas que uma de suas possíveis computações aceite uma palavra da linguagem L, e também requer que todas as computações rejeitem as palavras que não estão em L.

Logo, ao simular M, a máquina M' deve ser capaz de simular todas as computações possíveis de M para poder rejeitar uma palavra corretamente. A conclusão de que uma palavra está na linguagem pode ser mais rápida (encontrar uma computação que aceita a palavra é suficiente), mas de qualquer forma não é possível determinar a priori quantas simulações devem ser feitas.

Assim, considerando o pior caso, é fácil ver que toda simulação de M por M' que consiste em testar cada uma das computações possíveis pode consumir tempo exponencial no tamanho da entrada.

Porém, sabemos que M é polinomialmente limitada. Logo, todas as suas computações consomem tempo polinomial no tamanho da entrada. A partir desse fato, podemos perceber que a simulação de todas as computações possíveis de M por M' vai consumir espaço polinomial no tamanho da entrada (pois toda computação que consome tempo polinomial consome espaço polinomial, posto que a cabeça de leitura só se desloca no máximo uma posição para a direita a cada passo).

Logo, qualquer simulação de M por M' que simule cada computação em espaço polinomial serve para mostrar a relação. E a construção de uma simulação com essa característica é razoavelmente simples. A idéia principal consiste numa espécie de busca em largura num grafo.

O que M' vai fazer é simular primeiro todas as computações possíveis com t passos em M. Só após a simulação dessas computações é que M' vai simular as computações com $t+1$ passos.

Para controlar as computações, M' pode indexar as transições das computações com números. Esses números serão obtidos a partir de um contador, que será representado na base binária. Logo, apesar do número de computações poder ser exponencial, esse número usado por M' será polinomial no tamanho da entrada.

A cada passo da simulação, M' incrementa seu contador e simula a computação de M referente a seu valor. O conteúdo da fita para cada simulação pode ser guardado numa fita auxiliar durante a simulação desse passo, e antes da próxima simulação essa fita pode ser apagada. Dessa forma, o espaço usado para guardar as simulações também será polinomial no tamanho da entrada.

Como as transições de M podem ser simuladas pelos estados de M, temos que a simulação pode ser feita consumindo espaço polinomial no tamanho da entrada. Assim, temos que $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

As classes **RP** *e* **coRP**. As classes que discutimos a seguir envolvem computações probabilísticas, e são de fundamental importância tanto no modelo clássico como no modelo quântico.

A primeira classe que vamos definir é a classe **RP** (*randomized polynomial-time*). Na literatura, essa classe é definida em função das *máquinas de Turing Monte-Carlo*. Infelizmente não há um consenso quanto a uma definição para essas máquinas (algumas fontes falam que essas máquinas devem ser limitadas polinomialmente, enquanto outras não estabelecem essa restrição). Logo, vamos fazer a definição da classe em função de suas propriedades principais.

Para que uma linguagem L pertença à classe **RP**, deve existir uma máquina de Turing probabilística M limitada polinomialmente que tenha as seguintes propriedades:

(1) Se uma palavra x dada na entrada não está na linguagem L, então a máquina M sempre rejeita a palavra x.
(2) Se uma palavra x dada na entrada está na linguagem L, então a máquina M aceita x com probabilidade maior que 0,5.

Essas duas propriedades caracterizam a máquina de Turing probabilística Monte-Carlo.

A classe **RP** contém várias linguagens decididas por MTPs que em geral são bastante eficientes. Porém, como a definição mostra, tais máquinas não são precisas, pois podem rejeitar palavras erroneamente. A única garantia que temos é que nenhuma palavra que não está na linguagem será aceita pela máquina.

A existência desses erros pode ser contornada de modo que os resultados obtidos sejam satisfatórios. Uma maneira usual de fazer isso é executar a máquina com a mesma entrada algumas vezes, rejeitando uma palavra somente se tal palavra for rejeitada em todas as execuções. Assim, podemos reduzir bastante a probabilidade de uma rejeição indevida.

Tendo definido a classe **RP**, podemos definir a classe **coRP**, que pode ser vista como seu complemento. Para que uma linguagem L pertença à classe **coRP**, deve existir uma máquina de Turing probabilística M limitada polinomialmente que tenha as seguintes propriedades:

(1) Se uma palavra x dada na entrada não está na linguagem L, então a máquina M rejeita x com probabilidade maior que 0,5.
(2) Se uma palavra x dada na entrada está na linguagem L, então a máquina M nunca rejeita a palavra x.

As definições deixam claro o caráter complementar das duas classes. Enquanto as linguagens que estão em **RP** admitem máquinas que devolvem respostas positivas falsas, as linguagens de **coRP** admitem máquinas que devolvem respostas negativas falsas.

As classes **ZPP** *e* **BPP**. Na última seção, vimos classes de linguagens que são decididas por MTPs limitadas polinomialmente que podem aceitar ou rejeitar erroneamente uma determinada palavra.

Agora, vamos definir a classe de linguagens que exigem das máquinas que as respostas sejam corretas, mas que abrem mão da certeza da polinomialidade na execução.

Para que uma linguagem L pertença à classe **ZPP** (*zero-error probability polynomial-time*), deve existir uma máquina de Turing probabilística M que sempre aceita palavras que pertencem a L e sempre rejeita palavras que não pertencem a L cujo tempo esperado de execução é polinomial.

Pela definição, podemos perceber que a classe **ZPP** é muito parecida com a classe **P**, diferindo apenas no fato de que as máquinas utilizadas podem consumir tempo superpolinomial em algumas computações.

Por fim, vale destacar a seguinte propriedade: a classe das linguagens que pertencem tanto a **RP** como a **coRP** é a classe **ZPP**.

Outra classe de complexidade importante, que de certa forma pode ser relacionada com as demais classes probabilísticas já citadas, é a classe **BPP** (*bounded-error probabilistic polynomial-time*). Esta é a classe das linguagens que exigem máquinas que devolvem respostas erradas com probabilidade estritamente menor que 0,5. Atualmente, sabemos que **BPP** é igual a sua classe complementar, mas não conhecemos sua relação com a classe **NP**. Sabemos também que a classe **RP** está contida em **BPP**.

4. Modelo quântico de computação

4.1. **Tempo, espaço e linguagens no modelo quântico.** Durante as discussões sobre o modelo clássico de computação, fizemos considerações sobre o consumo de tempo e de espaço pelas máquinas de Turing determinísticas, não-determinísticas e probabilísticas. Faremos agora comentários sobre esses recursos agora para as máquinas de Turing quânticas.

O tempo consumido por uma MTQ com entrada x é o número de passos que a máquina efetua com entrada x até terminar a execução. O espaço consumido por uma MTQ M com entrada x é definido de maneira análoga ao espaço consumido por uma MTND com entrada x. Dentre todas as configurações que, durante a computação de M com x como entrada, tiveram amplitude não-nula, seja c aquela com o maior número possível de células em que M escreveu algo.

O número de células de c em que M escreveu algo é o espaço consumido por M com entrada x.

No contexto de linguagens, estamos interessados em MTQs que, para cada entrada x, tem saída ou $x1$ ou $x0$. (Lembre-se que MTQs devem ser reversíveis, por isso a resposta usualmente binária é precedida pela entrada.) Diz-se que uma MTQ M desse tipo decide de maneira exata uma linguagem L se, para todo x em L, a máquina M produz como saída $x1$, e para todo x fora de L, a máquina M tem como saída $x0$. Para um número p entre 0 e 1, diz-se que M decide L com probabilidade p se M, com entrada x em L, produz $x1$ com probabilidade pelo menos p e, com entrada x fora de L, produz $x0$ com probabilidade pelo menos p.

4.2. **As classes EQP e BQP.** As duas classes de complexidade que iremos introduzir aqui foram propostas por Bernstein e Vazirani [BV97].

A classe **EQP** é o conjunto das linguagens que são decididas de maneira exata por uma MTQ que consome tempo polinomial no tamanho da entrada. Essa classe corresponde à classe **P** do modelo clássico.

A classe **BQP** é o conjunto das linguagens que são decididas com probabilidade $2/3$. Essa classe corresponde à classe **BPP** do modelo clássico.

A seguir, vamos mostrar algumas relações envolvendo essas classes e as classes tradicionais de complexidade.

4.3. **Relações envolvendo as classes quânticas.** Inicialmente, vamos mostrar que $\mathbf{P} \subseteq \mathbf{EQP}$ e que $\mathbf{BPP} \subseteq \mathbf{BQP}$. Essas relações são intuitivamente claras, dado que as classes da computação clássica são casos restritos das respectivas classes quânticas. Seguem abaixo demonstrações breves das duas relações, apresentadas no artigo de Bernstein e Vazirani [BV97].

Teorema 4.1. $\mathbf{P} \subseteq \mathbf{EQP}$.

Demonstração. Seja L uma linguagem que pertence à classe **P**. Logo, existe uma MTD limitada polinomialmente que, para uma entrada x, produz como saída 1 se $x \in L$ e 0 caso contrário.

Para toda MTD, existe uma MTD equivalente (duas máquinas são equivalentes se sempre devolvem as mesmas respostas para as mesmas entradas) que é uma versão restrita de uma MTQ. Ao considerar os problemas de decisão, podemos perceber que existe uma MTQ limitada polinomialmente que, para uma entrada x, produz como saída 1 se $x \in L$ e 0 caso contrário e que nunca erra. Logo, L é decidida de maneira exata por uma MTQ, e portanto pertence a **EQP**, e $\mathbf{P} \subseteq \mathbf{EQP}$. □

Teorema 4.2. $\mathbf{BPP} \subseteq \mathbf{BQP}$.

Demonstração. Seja M uma MTP que decide com probabilidade $p > 0.5$ uma linguagem L. Para mostrar que $L \in \mathbf{BQP}$, podemos mostrar como construir uma MTQ que simula o comportamento de uma MTP.

Dada uma seqüência de bits aleatórios, uma computação de uma MTP pode ser simulada por uma MTD usando tais bits basicamente sem acréscimo no tempo consumido. Uma seqüência de bits aleatórios pode ser obtida por uma MTQ da maneira descrita a seguir.

Esses sorteios aleatórios podem ser realizados por uma MTQ, que recebe como entrada uma fita apenas com o símbolo 1, realiza uma transição onde duas configurações farão parte da próxima superposição. Uma das configurações vai manter a fita com seu conteúdo original, e a outra vai substituir 0 por 1. A amplitude das duas configurações será $\frac{1}{\sqrt{2}}$. Feita essa transição, a máquina faz a medição e pára, devolvendo 0 ou 1 com probabilidade 0,5 cada.

Assim, como podemos simular uma MTP usando uma MTQ e fazer os sorteios necessários também usando MTQs, a linguagem L pode ser decidida com probabilidade $p' = p > 0.5$ por uma MTQ limitada polinomialmente. Logo, $L \in \mathbf{BQP}$, e portanto temos que $\mathbf{BPP} \subseteq \mathbf{BQP}$. □

As duas provas mostradas acima servem para reforçar idéias que devem ser intuitivas após a leitura das definições envolvidas. Nosso objetivo agora é mostrar uma relação menos imediata e mais importante, que envolve o consumo de espaço na simulação de uma MTQ por uma MTD.

Quando falamos em simulação de uma MTQ por uma MTD, estamos nos referindo a uma simulação onde queremos saber a probabilidade de cada saída possível ser produzida. Não conhecemos nenhuma simulação de uma MTQ por uma MTD que consuma tempo polinomial no tempo consumido pela MTQ, porém o teorema abaixo mostra que é possível fazer uma tal simulação utilizando espaço polinomial no espaço consumido pela MTQ. Nesse texto, mostraremos apenas as linhas gerais da prova desse teorema.

Teorema 4.3. $\mathbf{BQP} \subseteq \mathbf{PSPACE}$.

Demonstração. Seja L uma linguagem em $\mathbf{BQP}$ e $M = (Q, \Sigma, \alpha, s, H)$ uma MTQ polinomialmente limitada que decide L com probabilidade $p > 2/3$. Vamos descrever uma MTD M' que decide L em espaço polinomial. A máquina M', para cada entrada x, calcula a probabilidade p_x de M aceitar x (ou seja, de M produzir x_1 como saída) e aceita ou rejeita x de acordo com o valor de p_x.

Dizemos que uma configuração $w_1 q w_2$ tem *tamanho* k se a cadeia de caracteres $w_1 w_2$ tem k caracteres. Dadas duas configurações c_1 e c_2, denotamos por $\alpha(c_1, c_2)$ o valor da amplitude correspondente à transição de M que leva c_1 a c_2. Se não há nenhuma transição que leva c_1 a c_2, então $\alpha(c_1, c_2) = 0$. Seja c_0 a configuração inicial $(\triangleright, s, x)$.

A máquina M', para cada t a partir de $|x|$, simula t passos de M e calcula a probabilidade p_x de M produzir, em t passos, a saída x_1. Essa probabilidade é dada por

$$p_x = \left| \sum_{c_1,\ldots,c_t} \prod_i \alpha(c_{i-1}, c_i) \right|^2,$$

onde o somatório é sobre todas as configurações $c_1, \ldots, c_t$ de tamanho no máximo t e $c_t = (\triangleright, h, x_1)$, para algum h em H. Note que o número de termos no somatório é no máximo $T = t|\Sigma|^{t^2}|Q|$.

A máquina M' deve portanto calcular o somatório descrito acima. O primeiro problema que aparece é a incapacidade de M fazer cálculos com números irracionais. MTDs só podem armazenar valores inteiros limitados, enquanto que cada $\alpha(c_{i-1}, c_i)$ não precisa nem mesmo ser racional. Logo, a simulação será uma aproximação, que deverá ter um erro tolerável. Por fim, vale destacar que cada um dos no máximo T termos da soma terá t fatores.

Se cada $\alpha(c_{i-1}, c_i)$ for representado com m bits, onde m é grande comparado à $\log T$, o erro será pequeno. Logo, cada elemento da matriz será representado com um par de inteiros de m bits (um bit para a parte real e outro para a parte imaginária), obtendo assim uma precisão de 2^{-m}. Assim, a primeira dificuldade já foi eliminada.

O que M deve fazer é computar cada termo do somatório e guardar a soma dos termos. Como cada soma usa pouco espaço auxiliar (ou seja, não utiliza uma quantidade exponencial de espaço em cada operação) e só é necessário guardar a cada operação a soma atual, podemos garantir que o espaço necessário para a operação é polinomial.

O que resta considerar agora é a obtenção de cada $\alpha(c_{i-1}, c_i)$. É fácil, em tempo polinomial nos comprimentos de c_{i-1} e c_i, detectar se c_i pode ou não ser derivada de c_{i-1} em um passo. Se não pode, então $\alpha(c_{i-1}, c_i) = 0$. Se pode, então ao mesmo tempo pode-se determinar a transição $(q_1, \sigma_1, q, \sigma, d)$ em $Q \times \Sigma \times Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}$) que, quando aplicada a c_{i-1}, leva a c_i. A máquina M' então aciona uma "sub-rotina" que recebe um número m e devolve, em espaço polinomial em m, o valor de $\alpha(q_1, \sigma_1, q, \sigma, d)$ com precisão 2^{-m}. O valor devolvido pela subrotina é a aproximação desejada de $\alpha(c_{i-1}, c_i)$. Mostrar que de fato há uma MTD que efetua a tal sub-rotina e consome espaço polinomial em m não é tarefa trivial e envolve uma série de etapas. Omitiremos essa prova nesse texto.

É importante notar que o número de tais sub-rotinas não depende da entrada, mas apenas da máquina M. Assim M' está bem definida, desde que existam MTDs que executem tais sub-rotinas. Mais do que isso, cada etapa que M' executa consome espaço polinomial (assumindo que as sub-rotinas consumam espaço polinomial), portanto de fato $\mathbf{BQP} \subseteq \mathbf{PSPACE}$. □

Dos resultados acima, temos que $\mathbf{BPP} \subseteq \mathbf{BQP} \subseteq \mathbf{PSPACE}$. Assim, não é possível mostrar que $\mathbf{BPP} \neq \mathbf{BQP}$ sem resolver a clássica questão de se $\mathbf{P}$ está propriamente contido em **PSPACE** ou não. Por outro lado, no mesmo artigo, Bernstein e Vazirani [BV97] mostram um oráculo em relação ao qual $\mathbf{BPP} \neq \mathbf{BQP}$. Outros resultados nessa linha, porém direcionados à questão "$\mathbf{P} \neq \mathbf{NP}$?", foram provados por Bennet *et al.* [BBBV97]. Por exemplo, Bennet *et al.* mostraram que, em relação a um oráculo, $\mathbf{NP} \not\subseteq \mathbf{BQP}$.

Referências

[AKS02] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. Preprint. Disponível em `http://www.cse.iitk.ac.in/news/primality.pdf`, 2002.

[BBBV97] C.H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani. Strengths and weaknesses of quantum computing. *SIAM J. Comput.*, 26(5):1510–1523, 1997.

[BV97] E. Bernstein and U. Vazirani. Quantum complexity theory. *SIAM J. Comput.*, 26(5):1411–1473, 1997.

[Chu33] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 25:839–864, 1933.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *Annals of Mathematics*, 58:345–363, 1936.

[Cob65] A. Cobham. The intrinsic computational difficult of functions. In Y. Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science*, pages 24–30. North-Holland, 1965.

[Coo71] S. Cook. The complexity of theorem proving procedures. In *Proc. 3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Deu85] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proc. Roy. Soc. London Ser. A*, 400(1818):97–117, 1985.

[Deu89] D. Deutsch. Quantum computational networks. *Proc. Roy. Soc. London Ser. A*, 425(1868):73–90, 1989.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[Fey82] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6 & 7):467–488, 1982.

[Göd31] K. Gödel. On formally undecidable propositions of Principia Mathematica and related systems. *Monatshefte fur Math. und Physik*, 38:173–198, 1931.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, 1979.

[Kar72] R.M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*, pages 85–103. Plenum, New York, 1972.

[Kle36] S. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

[Kle52] S. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, NJ, 1952.

[Lev73] L.A. Levin. Universal sorting problems. *Problems of Information Transmission*, 9:265–266, 1973.

[Pap94] C.H. Papadimitriou. *Computational complexity*. Addison-Wesley Publishing Company, Reading, MA, 1994.

[Pos36] E. Post. Finite combinatory process. *Journal of Symbolic Logic*, 1:103–105, 1936.

[Pre04] J. Preskill. Lecture notes for physics 219/computer science 219. `http://www.theory.caltech.edu/people/preskill/ph229`, 2004.

[RP00] E. Rieffel and W. Polak. Introduction to quantum computing. *ACM Computing Surveys*, 32(3):300–335, 2000.

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science (Santa Fe, NM, 1994)*, pages 124–134. IEEE Comput. Soc. Press, Los Alamitos, CA, 1994.

[Sho97] P.W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[Tur36] A. Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2:230–265, 1936.

[Tur37] A. Turing. Rectifications to 'on computable numbers...'. In *Proc. London Mathematical Society*, volume 4, pages 544–546, 1937.

Instituto de Matemática e Estatística, Universidade de São Paulo, Rua do Matão 1010, 05508–090 São Paulo, SP

Endereços Eletrônicos: `cardonha@ime.usp.br`

URL: `http://www.linux.ime.usp.br/~cardonha/quantum`