# 6 Complexidade computacional

## 6.1 A CLASSE P

No capítulo anterior, constatamos a existência de problemas de decisão bem definidos, que não são passíveis de resolução algorítmica, e apresentamos alguns exemplos específicos desse tipo de problemas. Podemos, portanto, classificar todos os problemas computacionais em duas categorias: aqueles que podem ser resolvidos através de algoritmos e aqueles que não podem. Com os grandes avanços na tecnologia da computação das últimas décadas, é razoável esperar que atualmente todos os problemas do primeiro tipo possam ser resolvidos de uma maneira satisfatória. Infelizmente, a prática da computação revela que muitos problemas, embora solúveis em teoria, não podem ser resolvidos em qualquer sentido prático pelos computadores, *devido às suas excessivas exigências de tempo de processamento.*

Suponha a tarefa de escalonar a visita de um caixeiro viajante a 10 escritórios regionais. É fornecido um mapa com as 10 cidades e as distâncias entre elas, e é pedido que se produza o itinerário que minimiza a distância total percorrida. Esse é, claramente, o tipo de tarefa cuja resolução se deveria confiar a um computador. E, do ponto de vista teórico, é seguramente um problema solúvel: se há $n$ cidades a serem visitadas, o número de itinerários possíveis é finito – exatamente, $(n-1)!$, isto é, $1 \cdot 2 \cdot 3 \ldots (n-1)$. Portanto, pode-se facilmente conceber um algoritmo que sistematicamente examine cada um dos possíveis itinerários, e dentre eles selecione o mais curto. Pode-se até mesmo projetar com naturalidade uma máquina de Turing que desempenhe essa tarefa.

Mas ainda há um defeito nesse algoritmo: *Há muitas viagens a serem examinadas.* Para nosso modesto problema de 10 cidades, teríamos de examinar $9! = 362.880$ itinerários. Com alguma paciência, isso pode ser realizado por um computador, mas e se tivéssemos 40 cidades a visitar? O número de itinerários seria agora gigantesco: $39!$, o que é mais do que $10^{45}$. Mesmo que pudéssemos examinar $10^{15}$ viagens por segundo – uma exagerada exigência mesmo para o mais poderoso supercomputador existente ou já projetado – o tempo necessário para a finalização desse cálculo seria de vários bilhões de ciclos de vida do universo!

Evidentemente, o fato de um problema ser solúvel na teoria não implica imediatamente que ele possa ser resolvido de maneira adequada na prática. Nossa meta neste capítulo é desenvolver uma teoria matemática formal – um refinamento quantitativo da tese de Church-Turing – que capture essa noção intuitiva de "um algoritmo viável na prática". A questão é: quais algoritmos – e quais máquinas de Turing – devemos classificar como praticamente viáveis?



Como o exemplo introdutório do PROBLEMA DO CAIXEIRO-VIAJANTE revela, o parâmetro limitador aqui é o *tempo* ou *número de passos* exigidos pelo algoritmo para processar uma dada entrada. O algoritmo apresentado para o problema do caixeiro viajante mostrou-se inadequado na prática devido ao crescimento *exponencial* de suas exigências de tempo de processamento (é fácil ver que a função $(n-1)!$ apresenta um crescimento até mesmo mais rápido do que $2^n$). Em contraste, algoritmos com uma taxa de crescimento *polinomial*, como alguns que foram desenvolvidos em outras partes deste livro, seriam obviamente muito mais atraentes.

Aparentemente, a noção de "algoritmo viável na prática" pode ser concretizada limitando sua implantação apenas aos dispositivos computacionais que executam um número de passos limitado por um **polinômio** sobre o comprimento da cadeia de entrada. Mas quais dispositivos computacionais devemos escolher que preenchem exatamente esses requisitos? A máquina de Turing e sua variante de múltiplas fitas, a máquina de Turing multidimensional ou talvez o modelo de acesso aleatório? A simulação investigada na Seção 4.3 nos informa que, estando interessados em taxas de crescimento polinomial, *a escolha não importa* – desde que descartemos o modelo não-determinístico com sua resposta exponencial (ver as Seções 4.5 e 6.4). Se uma máquina de Turing determinística *de qualquer um desses tipos* pára após um número polinomial de passos, há uma máquina de Turing equivalente *de qualquer outro tipo* que também pára após um número polinomial de passos – somente os polinômios irão diferir. Assim sendo, podemos escolher sempre o modelo mais simples, ou seja, a máquina de Turing padrão. A escolha das taxas polinomiais de crescimento como nosso conceito de "eficiência", traz-nos um interessante benefício ao permitir a adoção de um modelo independente como o das máquinas de Turing.

Somos, portanto, conduzidos à seguinte definição:

---

**Definição 6.1.1:** Uma máquina de Turing $M = (K, \Sigma, \delta, s, H)$ é dita **polinomialmente limitada** se há a polinômio $p(n)$, tal que, para qualquer entrada $x$, não há configuração $C$ tal que $(s, \triangleright \sqcup x) \vdash_M^{p(|x|)+1} C$. Em outras palavras, tal máquina sempre pára após, no máximo, $p(n)$ passos, onde $n$ é o comprimento da cadeia de entrada.

Uma linguagem é dita **polinomialmente decidível**, se houver alguma máquina de Turing polinomialmente limitada que a decida. A classe de todas as linguagens polinomialmente decidíveis é denotada por $\mathcal{P}$.

---

Nosso refinamento quantitativo da tese de Church-Turing pode agora ser expresso da seguinte forma: *máquinas de Turing polinomialmente limitadas e a classe $\mathcal{P}$ representam adequadamente as noções intuitivas de algoritmos praticamente viáveis e dos problemas solúveis na prática, respectivamente.*

Em outras palavras, estamos propondo $\mathcal{P}$ como o análogo quantitativo da classe de linguagens recursivas. De fato, $\mathcal{P}$ realmente apresenta algumas das propriedades da classe das linguagens recursivas:

**Teorema 6.1.1:** *$\mathcal{P}$ é fechada em relação à operação de complementação.*

*Prova:* Se uma linguagem $L$ é decidível por uma máquina de Turing $M$ polinomialmente limitada, então seu complemento é decidido pela versão de $M$ que inverte as saídas $y$ e $n$. Obviamente, o limite polinomial não é afetado por essa inversão. ■

Adicionalmente, podemos utilizar a diagonalização para apresentar algumas linguagens particulares recursivas simples que não pertencem à classe $\mathcal{P}$. Considere o seguinte análogo quantitativo da linguagem de "parada" $H$ (ver Seção 5.3):
$E = \{\text{"}M\text{"}\ \text{"}w\text{"}: M \text{ aceita a entrada } w \text{ após, no máximo, } 2^{|w|} \text{ passos}\}$.

**Teorema 6.1.2:** $E \notin \mathcal{P}$.

*Prova:* A prova imita muito bem a da indecidibilidade do problema da parada (Teorema 5.3.1). Suponhamos que $E \in \mathcal{P}$. Então, a seguinte linguagem também estará em $\mathcal{P}$:

$$E_1 = \{\text{"}M\text{"}: M \text{ aceita "}M\text{" após, no máximo, } 2^{|\text{"}M\text{"}|} \text{ passos}\},$$

e, pelo Teorema 6.1.1, assim também seu complemento, $\bar{E}_1$ pertencerá à classe $\mathcal{P}$. Portanto, há uma máquina de Turing $M^*$ que aceita todas as descrições das máquinas de Turing que *falham* em aceitar sua própria descrição em um tempo $2^n$ (onde $n$ é o comprimento dessa descrição); $M^*$ rejeita todas as outras entradas. Além disso, $M^*$ sempre pára em $p(n)$ passos, onde $p$ é um polinômio. Podemos assumir que $M^*$ é uma máquina de fita única, porque, caso contrário, ela poderá ser convertida para esse formato sem perda da resposta polinomial.

Como $p$ é um polinômio, deve existir um inteiro positivo $n_0$, tal que $p(n) \leq 2^n$ para todos os $n \geq n_0$. Além disso, pode-se admitir que o comprimento da codificação de $M^*$ seja, no mínimo, $n_0$, isto é, $|\text{"}M^*\text{"}| \geq n_0$. Caso isto não ocorra, podemos adicionar a $M^*$ $n_0$ estados desnecessários que nunca serão alcançados em qualquer operação.

Pode-se agora colocar a seguinte questão: *o que faz a máquina $M^*$ quando se lhe submete a sua própria descrição, "$M^*$"?* Ela aceita ou rejeita? Ambas as respostas conduzem a contradições: se $M^*$ aceita "$M^*$" então, como $M^*$ decide $\bar{E}_1$ isto implica que $M^*$ não aceita "$M^*$" em $2^{|\text{"}M^*\text{"}|}$ passos; mas $M^*$ foi construído de modo tal que sempre pare em $p(n)$ passos em resposta a entradas de comprimento $n$, e $2^{|\text{"}M^*\text{"}|} > p(|\text{"}M^*\text{"}|)$, portando ela deveria rejeitar essa entrada. Analogamente, admitindo que $M^*$ rejeita sua própria descrição, deduzimos que ele aceita a si própria. Considerando que a única hipótese que leva a essa contradição foi que $E \in \mathcal{P}$, devemos concluir que $E \notin \mathcal{P}$. ■

**Problemas para a Seção 6.1**

**6.1.1** Mostre que $\mathcal{P}$ também é fechada em relação às operações de união, intersecção e concatenação.



**6.1.2** Mostre que $\mathcal{P}$ é fechada em relação à operação da estrela de Kleene. (A solução deste problema é mais difícil que as provas das propriedades de fechamento, do problema anterior. Para dizer se $x \in L^*$ para algum $L \in \mathcal{P}$, é necessário considerar todas as subcadeias de $x$, começando com as de menor comprimento e prosseguindo em direção às maiores – de maneira muito parecida com o algoritmo de programação dinâmica para reconhecimento livre de contexto; lembre-se da Seção 3.6.)

## 6.2 PROBLEMAS, PROBLEMAS...

Até que ponto a classe $\mathcal{P}$ corporifica a noção intuitiva de "problema satisfatoriamente solúvel"? Em que extensão se aceita a tese de que os algoritmos polinomiais são, precisa ou empiricamente, viáveis? É razoável dizer que, embora esta seja a única proposta séria nessa área, ela pode ser desafiada em vários campos[†]. Por exemplo, pode-se argumentar que um algoritmo com exigências de tempo $n^{100}$ ou mesmo $10^{100}n^2$, não é "praticamente viável," embora apresente um tempo de resposta polinomial. Além disso, um algoritmo com exigências de tempo $n^{\log \log n}$ pode ser considerado perfeitamente viável na prática, a despeito do fato de que seu crescimento não seja limitado por qualquer polinômio. O argumento empírico em defesa de nossa tese é que tais limites extremos de tempo, embora teoricamente possíveis, *raramente surgem na prática*: algoritmos polinomiais, que surgem em práticas computacionais, geralmente têm pequenos expoentes e coeficientes constantes, enquanto algoritmos não-polinômiais são em geral exponenciais e, portanto, de utilização bastante limitada na prática.

Uma crítica adicional a essa técnica é que ela classifica um algoritmo com base somente em seu desempenho no *pior caso* (o maior dos tempos de execução para todas as possíveis entradas de comprimento $n$). É muito duvidoso que a utilização do *caso médio* como referência – por exemplo, insistindo que a exigência de tempo de uma máquina de Turing, calculada sobre a média de todas as possíveis entradas de comprimento $n$, seja limitada a $p(n)$ – poderia prever melhor a utilidade prática do algoritmo. Apesar de a análise do caso médio parecer uma alternativa muito razoável, ela de fato tem fragilidades que a tornam vulnerável a desafios ainda maiores. Por exemplo, qual deve ser a distribuição de entradas mais adequada para a análise do caso médio? Parece não haver uma resposta satisfatória.

A despeito desses entraves, porém, a computação limitada polinomialmente é um conceito atraente para uma teoria elegante e útil. Por se enfocar áreas pouco nítidas de suas fronteiras, freqüentemente se esquece quão útil é essa classificação, que seleciona a maioria dos algoritmos viáveis e exclui a maioria dos impraticáveis. No entanto, o melhor modo de familiarizar-se com os tempos de resposta polinomiais, com seu escopo real e com suas

[†] Até mesmo a tese de Church-Turing foi extensamente desafiada em sua época, e por ambos os lados: há matemáticos para quem o raciocínio de Turing sobre a decidibilidade se refere a uma noção muito restrita, enquanto outros o classificaram como excessivamente liberal. A teoria da complexidade, assunto deste e do próximo capítulo, pode, por sua vez, ser vista como o último e mais sério desses desafios.

limitações, é explorar uma diversidade de interessantes problemas computacionais, sabidamente pertencentes a $\mathcal{P}$. Também é instrutivo contrastar tais problemas com certos exemplos de problemas que obstinadamente *parecem não estar em* $\mathcal{P}$. Muitas vezes, problemas difíceis e problemas fáceis costumam ser muito similares.

Já vimos alguns elementos e alguma subclasses interessantes de $\mathcal{P}$. Por exemplo, todas as linguagens regulares e todas as linguagens livres de contexto pertencem a $\mathcal{P}$ (ver os Teoremas 2.6.2 e 3.6.1, parte (c)). Também sabemos que o *fechamento reflexivo transitivo* de uma relação pode ser computado em tempo polinomial (Seção 1.6). A seguir, examinaremos uma interessante variável desse último problema.

Trata-se do problema da *alcançabilidade*: "Dado um grafo orientado $G \subseteq V \times V$, onde $V = \{v_1, \ldots, v_n\}$ é um conjunto finito, e dois vértices $v_i, v_j \in V$, existe um caminho de $v_i$ para $v_j$?"

(Todos os grafos mencionados no contexto de problemas computacionais são, naturalmente, finitos.) O problema da alcançabilidade está em $\mathcal{P}$? Estritamente falando, como $\mathcal{P}$ contém apenas linguagens, o problema da alcançabilidade pertenceria a outro escopo, mas constitui aquilo que chamamos de **problema**. Um problema é caracterizado por um conjunto de *entradas*, tipicamente infinitas, junto com uma *pergunta com resposta sim ou não*, relativa a cada entrada (uma entrada pode ou não atender a uma propriedade). No exemplo da alcançabilidade, o conjunto das entradas é o conjunto de todas as triplas $(G, v_i, v_j)$, onde $G$ é um grafo, e $v_i$, $v_j$ são dois vértices de $G$. Deseja-se saber se há ou não um caminho de $v_i$ para $v_j$ em $G$.

Esta não é a primeira vez que tratamos problemas neste livro. O problema da parada é, definitivamente, dessa mesma categoria: suas entradas são máquinas de Turing e cadeias, e se deseja saber se a máquina de Turing fornecida pára quando se lhe apresenta essa cadeia de entrada: "Dada uma máquina de Turing $M$ e uma cadeia de entrada $w$, $M$ aceita $w$?"

No Capítulo 5 estudamos esse mesmo problema em termos de sua "formulação lingüística", a linguagem

$$H = \{\text{"}M\text{" "}w\text{": a máquina de Turing } M \text{ pára em resposta à cadeia } w\}.$$

Analogamente, podemos estudar o problema da alcançabilidade em termos da linguagem

$$R = \{\kappa(G)\mathbf{b}(i)\mathbf{b}(j)\text{: Há um caminho de } v_i \text{ para } v_j \text{ em } G\},$$

onde $\mathbf{b}(i)$ denota a codificação binária do inteiro $i$, e $\kappa$ é uma função que, de algum modo razoável, codifica grafos na forma de cadeias. Várias codificações naturais de grafos podem ser adotadas. Vamos estabelecer a convenção de codificar um grafo $G \subseteq V \times V$ através da sua *matriz de adjacências*, linearizada na forma de uma cadeia (ver Exemplo 4.4.3 e Figura 4-21). Uma das principais constatações resultantes da discussão que se segue é que *os detalhes precisos das codificações raramente* são significativas para esse estudo.



Linguagens codificam problemas. Naturalmente qualquer linguagem $L \subseteq \Sigma^*$ pode, por sua vez, ser estudada na forma do problema de decisão para a linguagem $L$: "dada uma cadeia $x \in \Sigma^*$, $x \in L$?"

É muito conveniente pensar em um problema e na linguagem a ele associada, intercambiavelmente. Linguagens são mais aderentes às representações usando máquinas de Turing, enquanto problemas constituem enunciados mais claros de tarefas computacionais, para as quais devemos desenvolver algoritmos. Nas páginas seguintes iremos apresentar e discutir extensivamente muitos problemas interessantes; devemos tratar o problema e a linguagem correspondente como dois aspectos diferentes do mesmo fenômeno. Por exemplo, devemos a seguir destacar que o problema da alcançabilidade está em $\mathcal{P}$. Com isso, subentendemos que a linguagem correspondente $R$, acima definida, está em $\mathcal{P}$.

De fato, o problema da alcançabilidade pode ser resolvido computando inicialmente o fechamento reflexivo transitivo de $G$, em tempo $O(n^3)$ com o auxílio da máquina de Turing de acesso aleatório do Exemplo 4.4.3. A inspeção da entrada do fechamento reflexivo transitivo de $G$ correspondente a $v_i$ e $v_j$ iria, então, informar-nos sobre a existência ou não de um caminho de $v_i$ para $v_j$ em $G$. Sabendo que máquinas de acesso aleatório podem ser simuladas por máquinas de Turing normais em tempo polinomial, conclui-se que o problema da alcançabilidade está em $\mathcal{P}$.

Note-se que, estando interessados apenas em determinar se o tempo-limite é ou não polinomial, temos a possibilidade de expressá-lo não como uma função *do comprimento da entrada*, que é $m = |\kappa(G)\mathbf{b}(i)\mathbf{b}(j)|$, mas como uma função de $n = |V|$, o número de vértices. Como $M = O(n^3)$, essa imprecisão não irá interferir significativamente nas nossas conclusões mais importantes.

### Grafos eulerianos e hamiltonianos

Historicamente o primeiro problema referente a grafos, estudado e resolvido pelo grande matemático do século XVIII Leonard Euler, é o seguinte:

TRILHA DE EULER: dado um grafo $G$, existe algum caminho fechado em $G$ que utiliza cada aresta do grafo exatamente uma vez?

Tal caminho pode visitar cada vértice muitas vezes (ou mesmo nenhuma, caso o grafo apresente conjuntos desconexos de *vértices*, sem arestas dentro ou fora deles). Um grafo que contém tal caminho é chamado **euleriano** ou **unicursal**. Por exemplo, o grafo mostrado na Figura 6-1(a) é euleriano em virtude da existência do caminho fechado $(v_1, v_2, v_3, v_4, v_3, v_2, v_4, v_1)$, enquanto o grafo na Figura 6-1 (b) não o é.

Não é difícil constatar que a TRILHA DE EULER está em $\mathcal{P}$ – com isso queremos dizer, naturalmente, que a linguagem correspondente

$$L = \{\kappa(G)\text{: } G \text{ é Euleriano}\}$$

está em $\mathcal{P}$. Isso decorre da seguinte caracterização elegante dos grafos eulerianos, creditada a Euler. Diz-se que um vértice está isolado em um grafo se e somente se não houver no grafo arestas incidentes sobre ele.

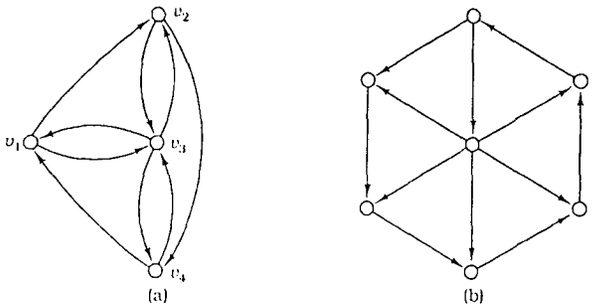



Figura 6-1[1]

*Um grafo G é Euleriano se e somente se apresentar as duas propriedades:*

*(a) Para qualquer par u, v∈ V, nenhum dos quais isolado, existe algum caminho de u a v; e*

*(b) Todos os vértices apresentam o mesmo número de arestas que entram e saem.*

É relativamente intuitivo que ambas as condições sejam necessárias para que o grafo seja euleriano. Deixamos a prova de suficiência como um exercício (Problema 6.2.1).

Portanto, é muito fácil testar se um grafo é Euleriano: primeiro, nos certificamos de que todos os vértices, exceto um eventual vértice isolado, estão conectados; isso pode ser feito em tempo polinomial computando-se o fechamento reflexivo transitivo do grafo e, então, testando se todos os vértices, exceto os isolados, estão conectados de todas as maneiras possíveis (afinal, os fechamentos transitivos reflexivos de um grafo contêm a resposta a todas as possíveis questões de conectividade no grafo). Sabemos que o fechamento reflexivo transitivo pode ser computado em um número polinomial de passos. Então, testamos se todos os vértices apresentam um número igual de arestas que entram e que saem – obviamente isso também pode ser feito em tempo polinomial.

Por acaso, esta é uma instância de um modelo que será repetido muitas vezes nas próximas páginas: mostramos que um problema (TRILHA DE EULER) está em $\mathcal{P}$ utilizando o fato previamente comprovado de que outro problema (em nosso caso, o da alcançabilidade) está em $\mathcal{P}$ – isto é, *reduzindo a um problema já resolvido.*

Talvez a principal conclusão deste e do próximo capítulo seja que existem muitos problemas naturais, decidíveis e estabelecidos de maneira simples cuja pertinência a $\mathcal{P}$ não *é conhecida ou acreditada.* Muitas vezes, tal problema é muito similar a outro que é conhecido por estar em $\mathcal{P}$! Considere, por exemplo, o seguinte problema, estudado por outro famoso matemático, desta vez do século XIX, William Rowan Hamilton – e muitos matemáticos depois dele:

[1] N. de R. - Os vértices do grafo (a) não estão especificados na edição original em inglês; a identificação dos vértices do grafo (b) é irrelevante.



CIRCUITO DE HAMILTON: Dado um grafo G, existe algum circuito que passa por cada um dos vértices de G exatamente uma vez?

Tal circuito é denominado de *circuito de Hamilton*, e um grafo que o apresenta é dito *Hamiltoniano*. Note-se que agora são os vértices e não as arestas que devem ser percorridas exatamente uma vez; nem todas as arestas precisam ser percorridas. Por exemplo, o grafo na Figura 6-1(b) é hamiltoniano, mas não euleriano, enquanto o outro na Figura 6-1(a) é tanto Euleriano como Hamiltoniano.

A despeito da superficial similaridade entre os problemas da TRILHA DE EULER E DO CIRCUITO DE HAMILTON, existem muitas diferenças entre eles. Após um século e meio de pesquisa por muitos matemáticos talentosos, nenhum logrou descobrir um algoritmo polinomial para o CIRCUITO DE HAMILTON. Embora, o seguinte algoritmo resolva o problema:

> Examine todas as possíveis permutações dos vértices;
> teste cada uma e verifique se ela é um circuito de Hamilton.

Infelizmente ele não é polinomial, assim como não o são os resultados de todas as tentativas simples de aprimorá-lo e acelerá-lo.

## Problemas de otimização

O PROBLEMA DO CAIXEIRO-VIAJANTE, apresentado informalmente no início deste capítulo, é outro problema simples para o qual, a despeito dos intensos esforços de várias décadas de pesquisa, nenhum algoritmo de tempo polinomial é conhecido. Dado um conjunto $\{c_1, c_2, \ldots, c_n\}$ de *cidades* e uma matriz $n \times n$ de inteiros não-negativos $d$, onde $d_{ij}$ denota a *distância* entre a cidade $c_i$ e a cidade $c_j$, admitindo que $d_{ii} = 0$ e $d_{ij} = d_{ji}$ para todos os $i, j$, pede-se encontrar a trajetória *mais curta* pelas cidades, isto é, uma bijeção $\pi$ do conjunto $\{1, 2, \ldots, n\}$ para ele próprio (onde $\pi(i)$ é a $i$-ésima cidade visitada nessa trajetória), tal que a quantidade

$$c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \ldots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)}$$

seja a menor possível.

Há um sério obstáculo para o estudo do problema do caixeiro-viajante, do ponto de vista de nossa estrutura de problemas e linguagens: diferente de outros problemas que vimos neste capítulo, o problema do caixeiro-viajante não é o tipo de problema que exige um "sim" ou "não" como resposta e que possa, portanto, ser estudado em termos de uma linguagem. Trata-se de um problema de *otimização*, no sentido de que exige a determinação da melhor (de acordo com alguma *função de custo*) entre muitas possíveis soluções.

Há um método que se mostra útil para transformar um problema de otimização em uma linguagem, de modo que possamos estudar sua complexidade: fornecer, para cada entrada, um *limite* correspondente para a função de custo. Neste caso, considere-se o seguinte problema, o conhecido problema do caixeiro-viajante:

"Dado um inteiro $n \geq 2$, uma matriz $n \times n$ de distâncias $d_{ij}$ e *um inteiro* $B \geq 0$ (representando o *orçamento* do caixeiro-viajante), pede-se determinar uma permutação $\pi$ de $\{1, 2, \ldots, n\}$ tal que $c(\pi) \leq B$."

Se pudéssemos resolver o problema original de otimização em tempo polinomial, isto é, se tivéssemos um algoritmo para computar a viagem mais econômica, então seríamos obviamente capazes de resolver essa versão do "orçamento" em tempo polinomial também: simplesmente computamos o custo da viagem mais econômica e o comparamos com $B$. Portanto, qualquer resultado *desfavorável* sobre a complexidade do problema do caixeiro-viajante, como acabamos de definir, se refletirá negativamente em nossas previsões para resolver a versão original do problema de otimização.

Devemos utilizar essa manobra para reduzir muitos problemas interessantes de otimização para a estrutura de linguagens que adotamos. No caso dos *problemas de maximização*, não fornecemos um orçamento $B$, mas em vez disso uma *meta* $K$. O exemplo a seguir, conhecido como o problema do conjunto independente, é um importante problema de maximização, transformado desta maneira:

"Dado um *grafo não-orientado* $G$ e um inteiro $K \geq 2$, existe algum subconjunto $C$ de $V$ com $|C| \geq K$ tal que para todos os $v_i, v_j \in C$, não existe nenhuma aresta entre $v_i$ e $v_j$?"

Este é mais um problema natural e simplesmente enunciado para o qual, a despeito do prolongado e intenso interesse dos pesquisadores, nenhum algoritmo de tempo polinomial foi encontrado.

Apresentamos a seguir mais dois problemas de otimização sobre grafos *não-orientados*. De alguma forma, o problema da clique, descrito a seguir, revela-se o oposto exato do problema dos conjuntos independentes:

"Dados um grafo *não-orientado* $G$ e um inteiro $K \geq 2$, existe algum subconjunto $C$ de $V$ com $|C| \geq K$, tal que para todo $v_i, v_j \in C$, há sempre alguma aresta entre $v_i$ e $v_j$?"

Para o próximo problema, denominado o problema da cobertura de vértices, dizemos que um conjunto de vértices *cobre* uma aresta se ele contiver, no mínimo, uma das extremidades dessa aresta:

"Dado um grafo não-orientado $G$ e um inteiro $B \geq 2$, existe algum subconjunto $C$ de $V$ com $|C| \leq B$ tal que $C$ cobre todas as arestas de $G$?"

Podemos considerar os vértices de um grafo não-orientado como sendo as salas de um museu, e cada aresta, como um corredor reto que une duas salas. Então, o PROBLEMA DA COBERTURA DE VÉRTICES pode ser útil para destacar o número mínimo possível de guardas para as salas, de modo que todos os corredores possam ser vigiados por um guarda.



Para ilustrar esses interessantes problemas, podemos observar que o maior conjunto independente para o grafo na Figura 6-2 apresenta três vértices; a maior clique contém quatro; e a menor cobertura de vértices possui seis. Quais seriam eles? Como argumentar que eles seriam ótimos?

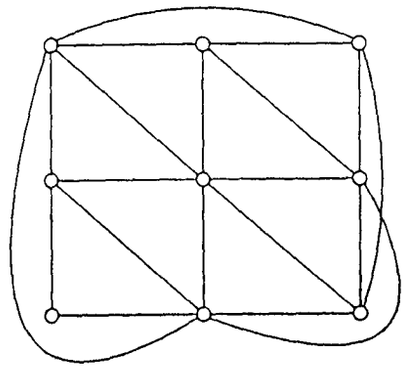


Figura 6-2

**Partições de inteiros**

Suponha que sejam fornecidos diversos inteiros positivos, digamos 38, 17, 52, 61, 21, 88, 25. Pergunta-se se eles podem ser separados em dois conjuntos disjuntos, que incluam todos os números fornecidos, de tal modo que a soma dos elementos de cada conjunto seja a mesma. No exemplo acima, a resposta é "sim", porque $38 + 52 + 61 = 17 + 21 + 88 + 25 = 151$. O problema geral da partição tem o seguinte enunciado:

"Dado um conjunto $S$ de $n$ inteiros não-negativos $a_1, \ldots, a_n$ *representados em binário*, pergunta-se se existe um subconjunto $P \subseteq \{1, \ldots, n\}$, tal que $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$?"

Esse problema pode ser resolvido pelo algoritmo simples explicado a seguir. Seja $H$ a metade da soma de todos os inteiros em $S$ (se esse número não for um inteiro, então a soma dos números em $S$ será um número ímpar, não podendo ser dividido em duas somas iguais; neste caso a resposta pode ser dada imediatamente: "não"). Para cada $i$, $0 \leq i \leq n$, defina-se o conjunto de números $B(i)$:

$B(i) = \{b \leq H: b$ é a soma de algum subconjunto dos números $\{a_1, \ldots, a_i\}\}$.

Se conhecemos $B(n)$, podemos facilmente resolver o problema da PARTIÇÃO simplesmente testando se $H \in B(n)$. Se isso ocorrer, então existirá um subconjunto de $S$ cuja soma seja exatamente $H$, e a resposta será "sim"; caso contrário, a resposta será "não".

Devemos notar que $B(n)$ pode ser computada pelo seguinte algoritmo:

$B(0) := \{0\}$.
para $i = 1, 2, \ldots, n$ faça
  $B(i) := B(i-1)$,
  para $j = a_i, a_i + 1, a_i + 2, \ldots, H$ faça
    se $j - a_i \in B(i-1)$ então acrescente $j$ a $B(i)$

Por exemplo, no caso particular que foi apresentado acima, com $a_1 = 38$, $a_2$ 17, $a_3 = 52$, $a_4 = 61$, $a_5 = 21$, $a_6 = 88$, $a_7 = 25$, os conjuntos $B(i)$ são os seguintes:

$B(0) = \{0\}$
$B(1) = \{0, 38\}$
$B(2) = \{0, 17, 38, 55\}$
$B(3) = \{0, 17, 38, 52, 55, 69, 90, 107\}$
$B(4) = \{0, 17, 38, 52, 55, 61, 69, 78, 90, 107, 113, 116, 130, 151\}$
$B(5) = \{0, 17, 21, 38, 52, 55, 59, 61, 69, 73, 76, 78, 82, 90, 99, 107, 111, 113, 116, 128, 130, 134, 137, 151\}$
$B(6) = \{0, 17, 21, 38, 52, 55, 59, 61, 69, 73, 76, 78, 82, 88, 90, 99, 105, 107, 109, 111, 113, 116, 126, 128, 130, 134, 137, 140, 143, 147, 149, 151\}$
$B(7) = \{0, 17, 21, 25, 38, 42, 46, 52, 55, 59, 61, 63, 69, 73, 76, 77, 78, 80, 82, 84, 86, 88, 90, 94, 98, 99, 101, 103, 105, 107, 109, 111, 113, 115, 116, 124, 126, 128, 130, 132, 134, 136, 137, 138, 140, 141, 143, 147, 149, 151\}$

Esse é um exemplo de PARTIÇÃO bem sucedida, porque a metade da soma $H = 151$ está contida em $B(7)$.

É fácil provar por indução em $i$, que esse algoritmo computa corretamente $B(i)$, para $i = 0, \ldots, n$, e que o faz em tempo $O(nH)$ (ver Problema 6.2.5). Então, provamos que o problema da partição está em $\mathcal{P}$?

O limite $O(nH)$, a despeito de sua aparência perfeitamente polinomial, *não é polinomial em relação ao comprimento da entrada*, já que os inteiros $a_1, \ldots, a_n$ são fornecidos em binário e, portanto, sua soma, em geral, será exponencialmente grande em relação ao comprimento da entrada. Por exemplo, se todos os $n$ inteiros em uma instância de PARTIÇÃO são da ordem de $2^n$, então $H$ será aproximadamente $\frac{n}{2}2^n$, enquanto o comprimento da entrada é somente $O(n^2)$. De fato, a PARTIÇÃO é um dos problemas notoriamente difíceis, juntamente com os PROBLEMAS DO CAIXEIRO-VIAJANTE, DO CIRCUITO DE HAMILTON e dos CONJUNTOS INDEPENDENTES, para os quais nenhum algoritmo verdadeiramente polinomial é conhecido e nem se espera que surja em um futuro próximo (e que são o assunto do próximo capítulo). Entretanto, o algoritmo acima realmente comprova que o seguinte problema está, de fato, em $\mathcal{P}$ (trata-se do problema da Partição Unária.

"Dado um conjunto de $n$ números naturais $\{a_1, \ldots, a_n\}$ *representados em unário*, pode-se determinar se existe um subconjunto $P \subseteq \{1, \ldots, n\}$ tal que $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$?"



Isto ocorre porque a entrada, na versão unária, tem comprimento próximo de $H$, e assim o algoritmo $O(nH)$ surpreendentemente torna-se "eficiente".

Esses dois problemas, o da PARTIÇÃO e o da PARTIÇÃO UNÁRIA, com sua contrastante complexidade, ilustram o seguinte importante fato acerca da representação de entradas: a representação matemática precisa de objetos como grafos, autômatos, máquinas de Turing e assim por diante, como entradas para problemas, apresenta pouca relevância para o estado ou a pertinência à correspondente linguagem ao conjunto $\mathcal{P}$, porque os comprimentos de todas as representações razoáveis do mesmo objeto estão relacionados através de um polinômio. A única convenção de codificação cuja violação pode levar a resultados duvidosos é que os *inteiros devem ser codificados em binário*[1] e *não* em unário.

### Equivalência de autômatos finitos

No Capítulo 2, vimos que o seguinte problema importante, relativo aos autômatos finitos, está em $\mathcal{P}$ (Teorema 2.6.1, parte (e)). Trata-se do problema da equivalência entre autômatos finitos determinísticos:

"Dados dois autômatos finitos determinísticos $M_1$ e $M_2$, é $L(M_1) = L(M_2)$?"

Em contraste, notamos que sabemos apenas como testar a equivalência de *autômatos finitos não-determinísticos* em tempo exponencial. Isto é, não sabemos se os seguintes dois problemas estão em $\mathcal{P}$:

EQUIVALÊNCIA DE AUTÔMATOS FINITOS NÃO-DETERMINÍSTICOS: dados dois autômatos finitos não-determinísticos $M_1$ e $M_2$, é $L(M_1) = L(M_2)$?

e

EQUIVALÊNCIA DE EXPRESSÕES REGULARES: dadas duas expressões regulares $R_1$ e $R_2$, é $L(R_1) = L(R_2)$?

Pode-se resolver o impasse transformando os dois autômatos não-determinísticos (ou expressões regulares) em autômatos finitos determinísticos (Teorema 2.6.1) e então verificar a equivalência dos autômatos determinísticos resultantes. O problema é, naturalmente, que a transformação de expressões regulares ou dos autômatos finitos não-determinísticos em autômatos finitos determinísticos pode aumentar exponencialmente o número de estados do autômato (ver Exemplo 2.5.4). Portanto, essa técnica não nos leva a constatar que os problemas da EQUIVALÊNCIA DE AUTÔMATOS FINITOS NÃO-DETERMINÍSTICOS e a EQUIVALÊNCIA DE EXPRESSÕES REGULARES estejam em $\mathcal{P}$.

### Problemas para a Seção 6.2

**6.2.1** Mostre que um grafo é Euleriano se e somente se ele for um grafo conexo e o número de arestas de entrada em cada vértice é igual ao

[1] Ou em decimal, hexadecimal ou em qualquer outra base de numeração; todas essas representações do mesmo inteiro têm comprimentos que guardam entre si uma relação dada por um fator constante.

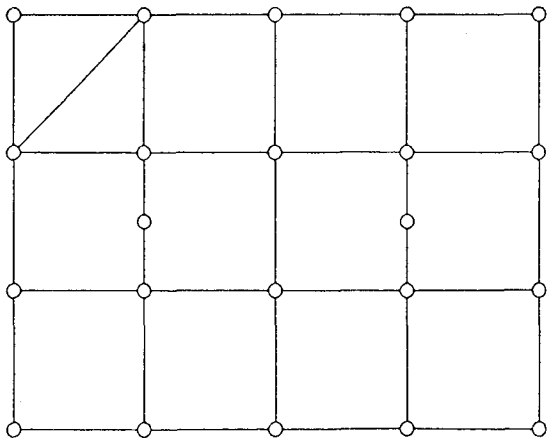
número de arestas de saída. (*Sugestão*: Uma parte dessa prova é fácil. Para a outra, comece em um vértice e percorra uma aresta para atingir outro vértice. Continue percorrendo as arestas até que não se possa localizar outro vértice; nesse ponto ter-se-á retornado ao vértice inicial (por quê?). Mostre como recomeçar e percorrer às partes do grafo que não foram percorridas.)

**6.2.2** Prove que o algoritmo apresentado no texto resolve o problema da PARTIÇÃO em $O(nH)$ passos.

**6.2.3** Resolva o problema do caixeiro-viajante para cinco cidades *A*, *B*, *C*, *D* e *E* com a seguinte matriz de distâncias:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} B & C & D & E \end{array} \\ \begin{pmatrix} 8 & 4 & 5 & 9 \\ & 1 & 7 & 3 \\ & & 6 & 2 \\ & & & 5 \end{pmatrix} \end{array}$$

**6.2.4** Estudamos problemas de otimização em termos das suas versões de linguagem, definidas em termos de um "orçamento" *B* ou "meta" *K*. Escolha um dos problemas de otimização apresentados nessa seção e mostre que existe um algoritmo polinomial para o problema original se, e somente se, houver outro algoritmo polinomial para a sua versão "sim-não". (Provar uma das partes é trivial. Para a outra, torna-se necessário utilizar *busca binária* além de uma propriedade chamada auto-redutibilidade.)



**6.2.5** O grafo não-orientado acima apresenta algum circuito de Hamilton? Qual é a maior clique, o maior conjunto independente e a menor cobertura de vértices desse grafo?



## 6.3 SATISFATIBILIDADE* BOOLEANA

Na seção anterior, vimos muitos problemas computacionais interessantes que estão em $\mathcal{P}$ e alguns outros que se suspeita não estarem em $\mathcal{P}$. Mas talvez o problema mais fundamental, referente a ambos os tipos, relaciona-se com a **lógica booleana**.

A lógica booleana é uma notação matemática familiar empregada para expressar proposições compostas como "ou está chovendo agora, ou a vassoura não está no canto". Na lógica booleana, utilizamos **variáveis booleanas** $x_1$, $x_2$ ... para representar fatos simples, tais como "está chovendo agora". Cada variável denota uma afirmação que tanto pode ser verdadeira como falsa independentemente do valor das demais. Utilizamos **conectores booleanos** para combinar variáveis booleanas e construir **fórmulas booleanas** mais complexas. Para nossa proposta, neste livro, precisamos considerar somente fórmulas booleanas de um tipo específico, definido a seguir.

---

**Definição 6.3.1:** Seja $X = \{x_1, x_2, \ldots x_n\}$ um conjunto finito de **variáveis booleanas**, e $\overline{X} = \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$, onde $\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}$ são novos símbolos que significam **negações**, ou opostos, de $x_1, x_2, \ldots, x_n$, respectivamente chamam-se **literais** os elementos de $X \cup \overline{X}$; variáveis são ditas **literais positivos**, enquanto negações de variáveis são ditas **literais negativos**. Uma **cláusula** $C$ é um conjunto não-vazio de literais: $C \subseteq X \cup \overline{X}$. Por fim, uma **fórmula booleana na forma conjuntiva normal** (ou simplesmente **fórmula booleana**, uma vez que não serão consideradas outro tipo neste livro) é um conjunto de cláusulas definidas em $X$.

---

**Exemplo 6.3.1:** Seja $X = \{x_1, x_2, x_3\}$ e, portanto, $\overline{X} = \{\overline{x_1}, \overline{x_2}, \overline{x_3}\}$. $C = \{x_1, \overline{x_2}, x_3\}$ é uma cláusula. Apesar de uma cláusula ser um conjunto de literais, devemos empregar uma notação especial quando escrevemos cláusulas: devemos utilizar parênteses em lugar do usual par de chaves; e devemos separar os vários literais na cláusula (se houver mais de um) pelo delimitador $\vee$ (pronuncia-se **ou**), em lugar de uma vírgula. Como se trata de conjuntos, a ordem dos elementos não é importante, e a sua repetição não é permitida. Por exemplo, a cláusula $C$ acima será escrita $C = (x_1 \vee \overline{x_2} \vee x_3)$.

A seguinte é uma fórmula Booleana em forma conjuntiva normal:

$$F = \{(x_1 \vee \overline{x_2} \vee x_3), (\overline{x}_1), (x_2 \vee \overline{x_2})\}. \qquad (1)$$

Ela consiste de três cláusulas, uma das quais é $C$, apresentada acima. ◊

---

**Definição 6.3.2:** Até aqui definimos apenas a *sintaxe*, ou seja, a estrutura aparente, de fórmulas booleanas. A seguir, definiremos a *semântica*, ou seja, o significado, de tal fórmula. Seja $F$ uma fórmula booleana em forma normal conjuntiva definida sobre as variáveis em $X = \{x_1, x_2, \ldots, x_n\}$. Uma **atribuição**

* N. de R. A palavra "Satisfatibilidade" não consta no dicionário, e está sendo utilizada como tradução de "Satisfiability". O mesmo vale para seus derivados.

**de valores verdade** para $F$ é um mapeamento de $X$ para o conjunto $\{\top, \bot\}$, onde $\top$ e $\bot$ são dois novos símbolos que se lêem **verdadeiro** e **falso**, respectivamente. Dizemos que a atribuição de valores verdade $T$ **satisfaz** $F$, se para cada cláusula $C \in F$ há, *no mínimo*, uma variável $x_i$ tal que ou (a) $T(x_i) = \top$ e $x_i \in C$ ou (b) $T(x_i) = \bot$ e $\bar{x}_i \in C$, ou seja, a cláusula é satisfeita se *ela* contém, *no mínimo, um literal verdadeiro*. Considere-se $x_i$ verdadeiro se, e somente se, $T(x_i) = \top$ e $\bar{x}_i$ é considerado verdadeiro se, e somente se, $T(x_i) = \bot$.

Por fim, $F$ é dita **satisfatível**, se houver uma atribuição de valores verdade que a satisfaça.

---

**Exemplo 6.3.2:** A fórmula booleana $F$ em (1) acima é satisfeita pela atribuição de valores verdade $T$, onde $T(x_1) = \bot$, $T(x_2) = \top$ e $T(x_3) = \top$. Essa atribuição de valores verdade satisfaz a cláusula $C_1 = (x_1 \vee \bar{x}_2 \vee x_3)$ porque $T(x_3) = \top$ e $x_3 \in C_1$; $T$ satisfaz a cláusula $C_2 = (\bar{x}_1)$, porque $\bar{x}_1 \in C_2$ e $T(x_1) = \bot$; e ela finalmente satisfaz a terceira cláusula $C_3 = (x_2 \vee \bar{x}_2)$ (isso não é tão surpreendente, pois qualquer atribuição de valores verdade iria satisfazer $C_3$). Há muitas atribuições de valores verdade que não satisfazem $F$. Por exemplo, qualquer atribuição de valores verdade $T'$ com $T'(x_1) = \top$ falha em satisfazer $C_2$ e, portanto, falha em satisfazer $F$. Adicionalmente, $F$ é satisfeita porque há, no mínimo, uma atribuição de valores verdade que a satisfaz. ✧

**Exemplo 6.3.3:** Considere-se agora a fórmula:

$$F' = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}.$$

Vamos provar que esta fórmula não é satisfatível.

A cláusula $(x_1 \vee x_2 \vee x_3)$ requer que, no mínimo, uma das variáveis $x_1$, $x_2$ ou $x_3$ seja $\top$. Analogamente, a última cláusula requer que, no mínimo, uma delas seja $\bot$. Consideremos então as três cláusulas restantes e suponhamos que $T(x_1) = \top$. Para que nesse caso a cláusula $(\overline{x_1} \vee x_2)$ seja satisfeita, $T(x_2)$ deve ser $\top$; e, para satisfazer $(\overline{x_2} \vee x_3)$, devemos ter $T(x_3) = \top$. Por outro lado, se $T(x_1) = \bot$, então a cláusula $(\overline{x_3} \vee x_1)$ obriga $T(x_3)$ a ser $\bot$ e a cláusula $(\overline{x_2} \vee x_3)$ faz com que $T(x_2)$ seja necessariamente $\bot$. Assim, independente do valor assumido por $T(x_1)$, *os valores verdade* das *três variáveis devem ser os mesmos* para que a fórmula possa ser satisfeita.

Em suma, para que uma atribuição de valores verdade satisfaça $F'$, (a) deve mapear, no mínimo, uma variável para $\top$; (b) deve mapear, no mínimo, uma variável para $\bot$; e (c) deve mapear todas as três variáveis para o mesmo valor $\top$. Isso é claramente impossível e, portanto, $F'$ codifica uma contradição, logo ela é **insatisfatível**. ✧

Isso sugere o seguinte problema importante, conhecido como o problema da satisfatibilidade:

> "Dada uma fórmula booleana $F$ na forma normal conjuntiva, ela é satisfatível?"

Das evidências ilustradas no exemplo anterior, trata-se de um problema relativamente complexo. De fato, não há algoritmo de resposta polinomial no tempo, conhecido até o momento, para esse problema fundamental e bem estudado. Acredita-se que tal algoritmo não existe.



### 2-Satisfatibilidade

Suponhamos que as instâncias do problema da SATISFATIBILIDADE sejam restritas àquelas em que *todas as cláusulas têm dois literais ou menos*. Disso resulta um novo problema, conhecido como 2-SATISFATIBILIDADE. Dizemos que 2-SATISFATIBILIDADE é um *caso particular do problema da* SATISFATIBILIDADE. Em outras palavras, o conjunto de todas as possíveis entradas é um subconjunto do conjunto de entradas do problema da SATISFATIBILIDADE e as respostas dos dois problemas a cada entrada comum são as mesmas. Um típico exemplo deste problema é o seguinte:

$$\{(x_1 \vee x_2), (x_3 \vee \overline{x_2}), (x_1), (\overline{x_1} \vee \overline{x_2}), (x_3 \vee x_4), (\overline{x_3} \vee x_5), (\overline{x_4} \vee \overline{x_5}), (x_4 \vee \overline{x_3})\} \quad (2)$$

Descrevemos a seguir um método para determinar uma atribuição que satisfaça tal fórmula. Ao longo desse processo, algumas variáveis recebem valores $\top$ ou $\bot$, enquanto as demais permanecem indefinidas. Inicialmente, a nenhuma variável é atribuído o valor $\top$.

Suponhamos que em nossa fórmula haja uma cláusula com apenas um literal, como é caso da terceira cláusula $(x_1)$ do exemplo em (2). Então, claramente esse literal deve ser $\top$ em qualquer atribuição que satisfaz a fórmula. Assim, podemos imediatamente decidir que $T(x_1) = \top$. Prosseguindo, podemos omitir da fórmula todas as cláusulas que contenham $x_1$ como um de seus literais, porque essas cláusulas já estão satisfeitas (em nosso exemplo, omitimos a primeira cláusula). Se, entretanto, uma cláusula contém o literal oposto $\overline{x_1}$ então omitimos o *literal da cláusula*, porque esse literal é $\bot$ e, portanto, não é de nenhuma utilidade para satisfazer a cláusula. Em nosso exemplo, a primeira cláusula é omitida e a quarta é substituída por $(\overline{x_2})$. Portanto, atribuir um valor $\top$ a um literal que aparece isolado em uma cláusula, pode resultar em novas cláusulas com um único literal e assim por diante (em nosso exemplo (2) a seguir vamos determinar que $T(x_2) = \bot$).

Esse método de obter sucessivas cláusulas com um só literal até que não reste nenhum é denominado *eliminação*[1]. Se em qualquer estágio da eliminação for detectada uma *cláusula vazia* – talvez porque para algum $i$ ambas as cláusulas $(x_i)$ e $(\overline{x_i})$, tenham sido encontradas no passo anterior – então diz-se que houve uma falha na eliminação. Em qualquer caso, após $O(n)$ passos desse tipo, onde $n$ é o número de cláusulas na fórmula fornecida, a eliminação deverá ou falhar (neste caso, conclui-se que a fórmula é insatisfatível) ou terminar com um conjunto de cláusulas cada uma das quais contendo dois literais distintos. Em (2), por exemplo, a eliminação inicial termina determinando $T(x_1) = \top$, $T(x_2) = \bot$ e excluindo as primeiras quatro cláusulas.

Dessa forma, pode-se admitir que a fórmula possui exatamente dois literais em cada cláusula. Uma idéia simples para efetuarmos a busca de uma atribuição que satisfaça essa fórmula é a seguinte: Escolhe-se uma

[1] Será utilizando o termo "eliminação" e seus derivados como tradução do inglês "purge".

variável qualquer, cujo valor verdade esteja ainda indefinido, e tenta-se impor seu valor como $\top$, realizando a eliminação; então, restaura-se a forma original da fórmula, estabelece-se para a mesma variável o valor $\bot$ e realiza-se a eliminação novamente. Se ambas as eliminações falharem, desiste-se, concluindo que a fórmula é *insatisfatível*. Se, porém, no mínimo uma das duas eliminações tiver êxito, então associa-se à variável o valor que levou ao êxito, e prossegue-se. Em nosso exemplo, experimentando-se o valor $T(x_3) = \top$ nas quatro cláusulas que restam após a primeira eliminação:

$$\{(x_3 \vee x_4), (\overline{x_3} \vee x_5), (\overline{x_4} \vee \overline{x_5}), (x_4 \vee \overline{x_3})\} \tag{3}$$

Iniciamos uma nova eliminação, que falha após estabelecer $T(x_5) = \top$ (resultam dessa operação as cláusulas $(x_4)$ e $(\overline{x_4})$): devemos assim restaurar as quatro cláusulas em (3) e tentar $T(x_3) = \bot$. Encontra-se, nesse caso, uma atribuição que satisfaz a fórmula (sem cláusulas eliminadas), a saber: $T(x_4) = \top$ e $T(x_5) = \bot$.

É fácil ver (Problema 6.3.2) que esse algoritmo simples resolve corretamente o problema da satisfatibilidade sempre que houver, no máximo, dois literais em cada cláusula. Uma vez que o algoritmo realiza no máximo duas eliminações para cada variável, e que cada eliminação pode ser executada em tempo polinomial, segue-se que o problema da 2-SATISFATIBILIDADE está em $\mathcal{P}$.

**Problemas para a Seção 6.3**

**6.3.1** Determine todas as atribuições que satisfazem a fórmula booleana contendo as cláusulas seguintes:

$$(x_1 \vee \overline{x_2} \vee x_3), (\overline{x_1} \vee x_4), (x_2 \vee \overline{x_3} \vee \overline{x_4}).$$

**6.3.2** (a) Mostre que o algoritmo de eliminação descrito no texto resolve corretamente em tempo polinomial qualquer instância do problema da 2-SATISFATIBILIDADE. (*Sugestão:* suponha que o algoritmo de eliminação determina que a fórmula é insatisfatível, mesmo existindo uma atribuição que satisfaz a fórmula. Como o algoritmo de eliminação deixou de encontrar esta atribuição?)

(b) Demonstre qual é o menor limite polinomial para esse algoritmo.

(c) Mostre passo a passo a operação do algoritmo de eliminação quando aplicado à fórmula seguinte:

$$(x_1 \vee \overline{x_2}), (\overline{x_1} \vee \overline{x_4}), (x_2 \vee \overline{x_3}), (x_1 \vee x_4), (x_3 \vee x_4).$$

**6.3.3** Neste problema, tem-se uma prova alternativa de que o problema 2-SATISFATIBILIDADE está em $\mathcal{P}$. Qualquer cláusula com dois literais, digamos $(x \vee y)$, pode ser interpretada como duas *implicações*, a saber $(\overline{x} \rightarrow y)$ e $(\overline{y} \rightarrow x)$ (a cláusula $(x)$ pode ser entendida como $(\overline{x} \rightarrow x)$). Assim, partindo-se de alguma instância do problema da 2-SATISFATIBILIDADE, pode-se construir um grafo orientado, utilizando como vértices todos os literais que descrevem tais implicações. Mostre que uma instância do problema da 2-SATISFATIBILIDADE é insatisfatível se e somente se existir uma variável $x$ tal que haja um caminho de $x$ para $\overline{x}$ e um caminho de $\overline{x}$ para $x$ nesse grafo. Isso leva a concluir que o problema da 2-SATISFATIBILIDADE está em $\mathcal{P}$.



## 6.4 A CLASSE NP

Um dos principais objetivos da teoria da complexidade é o de revelar métodos matemáticos que nos ajudem a provar que problemas de real interesse não estejam em $\mathcal{P}$. Um desses métodos já foi estudado: o *argumento da diagonalização* utilizado para estabelecer, de forma muito análoga a insolubilidade do problema da parada $H$, que $E \notin \mathcal{P}$, onde $E$ é a linguagem

$$E = \{\text{"}M\text{"}\ \text{"}w\text{"}: M \text{ aceita a entrada } w \text{ após no máximo, } 2^{|w|} \text{ passos}\}$$

(Teorema 6.1.2). Entretanto, esse resultado não é satisfatório. A razão é que, diferente da noção de decidibilidade, $\mathcal{P}$ e a computação em tempo polinomial são conceitos cuja motivação é de ordem prática. Não basta exibir uma versão artificial do problema da parada, tal como $E$, e mostrar que ele não esteja em $\mathcal{P}$. Queremos em lugar disso identificar problemas reais e importantes na prática que não estejam em $\mathcal{P}$.

Na última seção foram apresentados diversos problemas naturais, razoáveis, e de interesse prático, que parecem não pertencer a $\mathcal{P}$: CIRCUITO DE HAMILTON, O PROBLEMA DO CAIXEIRO VIAJANTE, O PROBLEMA DOS CONJUNTOS INDEPENDENTES, PARTIÇÃO E SATISFATIBILIDADE. Apesar dos prolongados e intensos esforços de matemáticos e cientistas da computação para desenvolverem um algoritmo de resposta polinomial no tempo para cada um desses problemas, nenhum algoritmo desse tipo foi descoberto até agora. Seria então mais recomendável, utilizar as idéias e os métodos de complexidade computacionais para determinar que tais algoritmos de resposta polinomial não são possíveis, poupando-nos, assim, de tentativas inúteis e fadadas ao fracasso.

Infelizmente, existe uma dificuldade sutil em provar tal impossibilidade. A razão é que, como será argumentado a seguir, *todos esses problemas podem ser resolvidos por máquinas de Turing não-determinísticas polinomialmente limitadas*. E separar o determinismo do não-determinismo no nível do tempo polinomial é um dos mais importantes e profundos problemas não resolvidos na ciência da computação de hoje.

Foi constatado no Capítulo 4 que, se uma linguagem $L$ é decidível em tempo polinomial por uma máquina de Turing de qualquer variedade (fita única, fitas múltiplas, bidimensional e até acesso aleatório), então $L$ é decidível em tempo polinomial por uma máquina de qualquer tipo. E, quanto ao modelo da máquina de Turing apresentado no final do Capítulo 4, a *máquina de Turing não-determinística* (Seção 4.6)? Seria ela também equivalente aos outros tipos, e também limitada por um polinômio? Para discutirmos essa importante questão, vamos primeiro definir formalmente o que é exatamente uma linguagem decidida por uma máquina de Turing não-determinística, com um limite de tempo polinomial: trata-se de uma simples extensão direta da definição correspondente apresentada para as máquinas determinísticas.

**Definição 6.4.1:** Dizemos que uma máquina de Turing não-determinística $M = (K, \Sigma, \Delta, H)$ é **polinomialmente limitada** se existir *um* polinômio $p(n)$ tal que, para qualquer entrada $x$, não haja uma configuração $C$ de $M$ com $(s, \triangleright\underline{\sqcup}x) \vdash_M^{p(|x|)+1} C$. Em outras palavras, nenhuma computação nessa máquina dura mais que um número polinomial de passos. A classe $\mathcal{NP}$ é definida como sendo a classe de todas as linguagens que são decididas por máquinas de Turing não-determinísticas polinomialmente limitadas[1].

É importante nesse ponto recordar a peculiar definição do que significa dizer que uma máquina não-determinística decide uma linguagem $L$: para cada cadeia de entrada que *não pertença à linguagem L, todas as* computações possíveis da máquina não-determinística devem rejeitar tal entrada; para cada cadeia de entrada pertencente a $L$, exige-se apenas que haja *pelo menos uma computação* que aceite tal entrada. Eventualmente as demais computações podem rejeitar a entrada, bastando portanto que ao menos uma delas a aceite.

O conjunto de todas as possíveis computações de uma máquina de Turing não-determinística, em resposta a uma dada cadeia de entrada, pode ser bem representado por uma estrutura em forma de árvore (veja Figura 6-3), onde os vértices representam as configurações, e as linhas descendentes, os passos. Escolhas não-determinísticas correspondem aos vértices que apresentam mais de uma linha descendente partindo deles. O tempo é medido na dimensão vertical. Na Figura 6-3, por exemplo, a cadeia de entrada é aceita após cinco passos de computação.

Esse cenário faz transparecer a razão do poder computacional do não-determinismo: há um número muito grande de configurações, produzidas

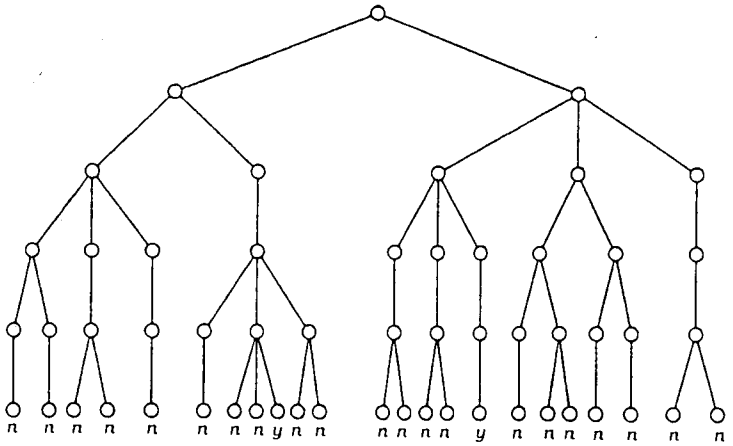



Figura 6-3

[1] N. de R. Deve-se tomar cuidado para entender $\mathcal{NP}$ corretamente como *não-determinístico polinomial* e não confundir com a interpretação incorreta de "*não polinomial*".



em um tempo bastante curto (a distância vertical a partir da raiz). Como se pode verificar pelos próximos exemplos, esse poder do não-determinismo pode ser explorado para "resolver" alguns dos problemas complexos estudados na última seção.

**Exemplo 6.4.1:** Na seção anterior vimos que é muito possível que o problema da SATISFATIBILIDADE não pertença à $\mathcal{P}$. Vamos agora mostrar que ele pertence a $\mathcal{NP}$. Devemos projetar uma máquina de Turing não-determinística $M$ que decide em tempo polinomial não-determinístico quaisquer codificações satisfatíveis de fórmulas booleanas em forma conjuntiva normal.

$M$ opera da seguinte forma: dada uma cadeia de entrada $w$, verifica inicialmente se $w$ codifica realmente uma fórmula Booleana na forma conjuntiva normal (se não for, há uma rejeição imediata) e conta o número de variáveis $n$, que nela aparecem. Isso é fácil de executar DETERMINISTICAMENTE, em tempo polinomial. Nessa ocasião, a segunda fita de $M$ contém a cadeia $\triangleright I^n$, onde o número de $I$'s e igual ao número de variáveis da fórmula apresentada.

Neste ponto, $M$ entra em uma *fase não-determinística*, durante a qual grava em sua segunda fita uma seqüência de $n$ símbolos $\top$'s e $\bot$'s em lugar dos $I$'s. A seqüência exata de $\top$'s e $\bot$'s é "*qualquer seqüência, não-deterministicamente*", ou mais precisamente, "*todas as seqüências, correspondentes a cada um dos diferentes ramos da árvore não-determinística de computações*". É fácil projetar tal máquina não-determinística: basta adicionarmos um novo estado $q$ a $K$, e adicionarmos a $\Delta$ (ignorando as outras fitas, onde a atividade não leva a lugar algum) as novas transições $(q, I, q, \top)$, $(q, I, q, \bot)$, $(q, \top, q, \rightarrow)$, $(q, \bot, q, \rightarrow)$, $(q, \sqcup, q', \sqcup)$, onde $q'$ é o estado a partir do qual a computação prosseguirá.

Em sua fase final, $M$ é determinística: $M$ interpreta a cadeia sobre $(\top, \bot)^n$ contida em sua segunda fita, uma a uma, como sendo uma *atribuição de valores verdade* para a fórmula fornecida. $M$ visita então cada cláusula dessa entrada e verifica se ela contém um literal que é $\top$ sob a atribuição de valores verdade. Se todas as cláusulas apresentam um literal $\top$, $M$ aceita a entrada. Caso contrário, rejeita-a.

É simples e direto constatar que $M$, acima descrita, comprova que o problema da SATISFATIBILIDADE está em $\mathcal{NP}$: todas as computações têm comprimento limitado por algum polinômio. Para determinar se a cadeia de entrada codifica uma fórmula booleana satisfatível, $M$ admite (por tentativa) que a atribuição correspondente a algum ramo arbitrário da árvore de computação não-determinística satisfaça a fórmula. Tendo aceito pelo menos uma computação, não importarão as eventuais rejeições remanescentes, e, portanto, a cadeia de entrada será aceita. Se a fórmula fornecida for insatisfatível ou se não for uma fórmula válida, então, todas as computações serão rejeitadas. ✧

**Exemplo 6.4.2:** O PROBLEMA DO CAIXEIRO-VIAJANTE (conforme definido na Seção 6.2 com o "orçamento" $B$ fornecido) também pertence à classe $\mathcal{NP}$. A máquina de Turing não-determinística que finaliza com sucesso grava em sua

segunda fita, não-deterministicamente, uma cadeia sobre o conjunto {0, 1, ␣'s} de comprimento igual ao da cadeia de entrada. Então, a máquina efetua um processamento determinístico, no qual verifica se a cadeia gravada em sua segunda fita *codifica uma bijeção* $\pi$ dos inteiros 1, ..., $n$ onde $n$ é o número de cidades na entrada dada – a bijeção $\pi$ é codificada escrevendo-se $\pi(1)$, $\pi(2)$, ... em binário, separados por ␣'s. Se a cadeia constitui, de fato, a codificação de uma bijeção, a máquina irá, deterministicamente, calcular o custo da viagem e compará-lo com o "orçamento" $B$ fornecido como entrada. Se o custo for menor, a máquina aceita a entrada corrente; em todas as outras situações (se a suposta cadeia não é a codificação de uma bijeção, ou se ela representa uma viagem com custo maior do que $B$) a máquina a rejeita. É claro que uma cadeia pertencerá à linguagem decidida por essa máquina se, e somente se, ela codificar uma instância que constitui uma solução do PROBLEMA DO CAIXEIRO-VIAJANTE.

É fácil mostrar de forma análoga que os outros problemas aparentemente difíceis, apresentados na seção anterior: CONJUNTO INDEPENDENTE, CIRCUITO DE HAMILTON e PARTIÇÃO (com exceção das EQUIVALÊNCIA DE AUTÔMATOS FINITOS NÃO-DETERMINÍSTICOS) também pertencem à classe $\mathcal{NP}$. ✧

Observe com que inteligência os procedimentos não-determinísticos dos dois exemplos anteriores exploram a *assimetria fundamental* contida na definição de computação em tempo limitado não-determinístico. Eles experimentam todas as possíveis soluções disponíveis para o problema através de computações independentes, e terminam o processamento aceitando a cadeia assim que obtêm sucesso em uma das tentativas, ignorando todas as outras que não funcionam ou que não se completarem.

Tenta-se aqui uma analogia com a classe das linguagens recursivamente enumeráveis, outra classe cuja definição apresentava uma assimetria similar entre as situações de aceitação da rejeição. Como acontece neste caso, absolutamente não é claro que $\mathcal{NP}$ seja fechada em relação à complementação (embora seja claro no caso da classe $\mathcal{P}$, e também da classe das funções recursivas). Além disso, é imediato que $\mathcal{P} \subseteq \mathcal{NP}$ (o análogo do fato que cada linguagem recursiva é recursivamente enumerável). Essa é a razão por que máquinas determinísticas podem ser consideradas casos particulares de máquinas não-determinísticas, cuja relação de transição se reduz sempre a uma função.

Seria $\mathcal{P}$ igual a $\mathcal{NP}$? Em outras palavras, as máquinas de Turing não-determinísticas seriam simplesmente mais uma versão das máquinas de Turing, e seriam equivalentes às demais com relação à classe de linguagens decididas em tempo polinomial? À primeira vista, tem-se a sensação intuitiva de que o não-determinismo é um recurso tão poderoso e "diferente" que isso não seria verdade. As árvores que representam o conjunto de computações de uma máquina de Turing não-determinística (ver a Figura 6-3) têm muitos vértices (ou seja, muitas configurações), todos em uma profundidade moderada. Uma máquina de Turing determinística só consegue competir com as não-determinísticas, em relação ao número de configurações alcançáveis, quando opera em um número exponencial de passos. As máquinas não-determinísticas que decidem o problema da SATISFATIBILIDADE e o PROBLEMA DO CAIXEIRO-VIAJANTE efetuam quase sem esforço, uma busca em um número exponencialmente grande de possibilidades; seria realmente notável se o



mesmo efeito pudesse ser conseguido deterministicamente, de maneira metódica e em tempo polinomial.

Essa dificuldade de utilizar uma máquina de Turing determinística para efetuar uma busca em um grande conjunto de "propostas" refletiu-se também na prova do Teorema 4.5.1. Mostramos nesse teorema que uma máquina de Turing não-determinística pode ser simulada por outra, determinística; mas essa simulação não foi uma simulação direta passo a passo, como as dos Teoremas 4.3.1 e 4.3.2 (para os quais fomos capazes de provar os limites polinomiais). A simulação de uma máquina de Turing não-determinística recorreu ao exame exaustivo de todas as possíveis computações. Novamente, temos a sensação intuitiva de que isso é inerente ao não-determinismo, uma vez que ele nos oferece múltiplas escolhas a cada passo e assim há um número exponencialmente grande de possíveis computações a serem verificadas.

Para comparar os desempenhos no tempo das máquinas não-determinísticas e das determinísticas, devemos antes definir uma classe de linguagens muito mais geral:

**Definição 6.4.2:** Uma máquina de Turing $M = (K, \Sigma, \delta, s, H)$ **diz-se exponencialmente limitada** se houver um polinômio $p(n)$ tal que para qualquer entrada $x$, não haja configuração $C$ tal que $(s, \triangleright\sqcup x) \vdash_M^{2^{p(|x|)}+1} C$, isto é, *tal máquina* sempre pára após, no máximo, um número exponencial de passos.

Por fim, define-se $\mathcal{EXP}$ como a classe de todas as linguagens decididas por alguma máquina de Turing exponencialmente limitada.

**Teorema 6.4.1:** *Se* $L \in \mathcal{NP}$, *então* $L \in \mathcal{EXP}$.

***Prova:*** Seja dada uma máquina de Turing não-determinística polinomialmente limitada $M$ que decide $L$ em um tempo limite $p(n)$. Vamos mostrar como construir uma máquina de Turing determinística $M'$ que decide a mesma linguagem em tempo $c^{p(n)}$ para alguma constante $c$ (o teorema daí decorre, considerando um polinômio $k \cdot p(n)$, para algum $k$, tal que $2^k > c$). $M'$ é a mesma máquina que foi construída na prova do Teorema 4.5.1. $M'$ simula $M$ para todas as possíveis computações de comprimento 1, depois, para todas as possíveis computações de comprimento 2 e assim por diante, até o comprimento $p(n) + 1$, quando uma computação de aceitação foi descoberta ou todas as computações possíveis foram rejeitadas. Para simular uma computação de comprimento $l$ em $M$, $M'$ precisa de $O(l)$ passos – para copiar a entrada, para produzir a próxima cadeia sobre $\{1, 2, ..., r\}^l$ (onde $r$ é o *grau não-determinístico de M*, um número fixo que depende apenas de $M$ e igual ao número máximo possível de quádruplas em $\Delta$ que compartilham os mesmos dois primeiros componentes) e para simular $M$, seguindo as escolhas sugeridas por essa cadeia. Então, $M'$ pode realizar a simulação da operação de $M$ sobre uma entrada de comprimento $n$ em um tempo

$$\sum_{l=1}^{p(n)+1} r^l \leq (r+1)^{p(n)+1},$$

a qual completa a prova fazendo-se $c = r + 2$. ■

Conforme já foi mencionado, determinar $\mathcal{P} = \mathcal{NP}$ é uma questão não-resolvida de extrema importância para a teoria da complexidade. Determinar se $\mathcal{NP} = \mathcal{EXP}$, é outra questão que, embora um pouco menos importante, ainda permanece em aberto. O que *realmente* sabemos é o seguinte: na série de inclusões

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP},$$

a *classe* $\mathcal{EXP}$ *inclui propriamente a classe* $\mathcal{P}$. A razão é que a linguagem $E$, que se provou não pertencer a $\mathcal{P}$ (no Teorema 6.1.2), está certamente em $\mathcal{EXP}$: uma máquina de Turing pode, em tempo exponencial, simular a operação de $M$ sobre a cadeia de entrada $w$ por $2^{|w|}$ passos. *Portanto, apesar de suspeitarmos que ambas as inclusões mostradas acima são próprias, tudo o que podemos provar é que pelo menos uma delas é própria – e não sabemos qual...*

### Certificados sucintos

As máquinas de Turing não-determinísticas, que projetamos nos Exemplos 6.4.1-2 para decidir o problema da SATISFATIBILIDADE e o PROBLEMA DO CAIXEIRO VIAJANTE, são bastante simples e apresentam semelhanças entre si: elas começam gerando não-deterministicamente uma cadeia e então, verificando deterministicamente se a cadeia gerada apresenta ou não uma certa propriedade exigida. Se a cadeia de entrada pertence à linguagem, então pelo menos uma cadeia apropriada existe. Caso a cadeia de entrada não pertence à linguagem, então não existirá cadeia alguma que apresente a propriedade exigida.

Tal cadeia denomina-se **certificado** ou **testemunha**. Como veremos, todos os problemas em $\mathcal{NP}$, e somente eles, apresentam certificados. Um certificado deve ser *polinomialmente sucinto*, isto é, seu comprimento deve ser dado no máximo, pelo valor de um polinômio sobre o comprimento da cadeia de entrada. Ele deve também ser *verificável* em tempo polinomial. No caso do problema da SATISFATIBILIDADE, essa verificação implica testar se alguma atribuição satisfaz todas as cláusulas da fórmula fornecida como entrada; no caso do PROBLEMA DO CAIXEIRO-VIAJANTE, o teste é para verificar se a viagem proposta tem um custo total compatível com o orçamento; para CONJUNTO INDEPENDENTE, verifica-se se o conjunto fornecido é do tamanho correto e isento de arestas; e assim por diante. Por fim, todas as entradas que dão resposta "sim" a um problema devem ter pelo menos um certificado, enquanto todas as demais não devem apresentar nenhum certificado.

A idéia dos certificados pode ser formalizada no domínio das linguagens, fornecendo uma interessante definição alternativa para a classe $\mathcal{NP}$.

**Definição 6.4.3:** Seja $\Sigma$ um alfabeto e ";" um símbolo não pertencente a $\Sigma$. Considere a linguagem $L' \subseteq \Sigma^*; \Sigma^*$. Dizemos que $L'$ é **polinomialmente equilibrado** se nela existe um polinômio $p(n)$, tal que, se $x; y \in L'$, então $|y| \leq p(|x|)$.



**Teorema 6.4.2:** *Seja* $L \subseteq \Sigma^*$ *uma linguagem, onde* ";" $\notin \Sigma$ *e* $|\Sigma| \geq 2$. *Então,* $L \in \mathcal{NP}$ *se e somente se existir uma linguagem polinomialmente equilibrada* $L' \subseteq \Sigma^*; \Sigma^*$, *tal que* $L' \in \mathcal{P}$ *e* $L = \{x$: há um $y \in \Sigma^*$, tal que $x; y \in L'\}$.

***Prova:*** Intuitivamente, a linguagem $L'$ apresenta como sentenças todos os certificados de todas as entradas, isto é,

$$L = \{x; y: y \text{ é um certificado para } x\}.$$

Para cada $x \in \Sigma^*$, há um conjunto de $y$'s, tal que $x; y \in L'$: esse é o conjunto de certificados de $x$. Se $x$ pertence a $L$, então seu conjunto de certificados tem pelo menos um elemento; se $x \notin L$, então esse conjunto de certificados será vazio.

Se uma tal linguagem $L'$ existe, então uma máquina de Turing NÃO-DETERMINÍSTICA pode decidir $L$ experimentando todos os certificados (de maneira muito parecida com a máquina de Turing NÃO-DETERMINÍSTICA que decide o problema da SATISFATIBILIDADE) e, então, acionando a máquina de Turing determinística que decide $L'$. Inversamente, qualquer máquina de Turing NÃO-DETERMINÍSTICA $M$ que decide $L$ oferece uma sólida estrutura de certificados para $L$: o certificado para uma entrada $x$ é precisamente qualquer computação de aceitação de $M$ para a entrada $x$ – este certificado é sucinto e polinomialmente verificável.

A prova formal é deixada como exercício para o leitor (Problema 6.4.5). ■

O conceito de certificados sucintos é melhor ilustrado nos termos do conjunto $C \subseteq \mathbf{N}$ de *números compostos*[1] (ver as discussões no Exemplo 4.5.1). Seja dado um número natural em sua representação decimal usual – por exemplo, 4.294.967.297 – e nos perguntamos se ele é composto. Não há um modo claro e eficiente de responder a essa pergunta. Entretanto, cada número em $C$ realmente tem um certificado sucinto. Por exemplo, o número 4.294.967.297, que é composto, tem como certificado o par de inteiros 6.700.417 e 641 cujo produto é 4.294.967.297. Para verificar o certificado, é necessário apenas realizar a multiplicação para constatar que $4.294.967.297 \in C$. E essa é a sutileza de um certificado: uma vez que você o tenha encontrado, você pode eficientemente provar sua validade. Mas encontrá-lo pode não ser fácil: a fatoração de 4.294.967.297 foi descoberta pelo matemático Leonard Euler (1707-1783) em 1732, bem depois, 92 anos precisamente, de Pierre de Fermat (1601-1665), outro grande matemático, ter conjecturado que não existiria tal fatoração!

### Problemas para a Seção 6.4

**6.4.1** Mostre que $\mathcal{NP}$ é uma classe fechada em relação às operações de união, intersecção, concatenação e estrela de Kleene. (As provas para a classe $\mathcal{NP}$ são mais simples do que as correspondentes, referentes a $\mathcal{P}$.)

[1] N. de R. Números compostos são números que possuem fatores que não são nem a unidade nem o próprio número. São também conhecidos como números múltiplos.

**6.4.2** Definamos co$\mathcal{NP}$, como sendo a seguinte classe de linguagens: $\{\bar{L} : L \in \mathcal{NP}\}$. É muito interessante estudar se $\mathcal{NP}$ = co$\mathcal{NP}$, isto é, se $\mathcal{NP}$ é fechado em relação à operação de complementação. Mostre que, se $\mathcal{NP} \neq$ co$\mathcal{NP}$, então $\mathcal{P} \neq \mathcal{NP}$.

**6.4.3** Um homomorfismo $h$ (ver Problemas 2.3.11 e 3.5.3) diz-se "non-erasing"[1] se ele não mapeia nenhum símbolo para $\varepsilon$. Mostrar que $\mathcal{NP}$ é fechado em relação às operações de homomorfismo "non-erasing".
$\mathcal{NP}$ é fechado em relação aos homomorfismos gerais? E a classe das linguagens recursivas? E as recursivamente enumeráveis? E a classe $\mathcal{P}$? Veja o Problema 7.2.4 do próximo capítulo ao estudar a classe $\mathcal{P}$.

**6.4.4** Defina certificados sucintos apropriados para os problema da PARTIÇÃO e das CLIQUES.

**6.4.5** Prove o Teorema 6.4.2.

## REFERÊNCIAS


*A teoria da complexidade computacional, muito antecipada nos anos 50, e, mais recentemente, nos anos 60, foi formalmente desenvolvida por Hartmanis e Stearns no artigo:*

- J. Hartmanis and R. E. Stearns "On the computational complexity of algorithms", *Transactions of the American Mathematical Society*, 117, pp. 285-305, 1965;

*O Teorema 6.1.2 decorre do resultado apresentado nesse artigo. A tese de que a classe $\mathcal{P}$ representa adequadamente a noção de um "problema eficientemente solúvel", também implícita em trabalhos anteriores, em meados da década de 60 principalmente nos dois trabalhos seguintes:*

- A. Cobham "The intrinsic computational difficulty of functions," *Proceedings of the 1964 Congress for Logic, Mathematics and the Philosophy of Science*, pp. 2430, New York: North Holland, 1964, e
- J. Edmonds "Paths, trees and flowers", *Canadian Journal of Mathematics*, 17, 3, pp. 449-467, 1965.

*Neste último artigo, a classe $\mathcal{NP}$ também foi informalmente apresentada (em termos de certificados, ver Teorema 6.4.2), tendo sido o primeiro a conjecturar que $\mathcal{P} \neq \mathcal{NP}$. Um tratamento muito mais extenso da complexidade computacional, pode ser encontrado em:*

- C. H. Papadimitriou *Computational Complexity*, Reading, Massach.: Addison Wesley, 1994.


[1] N. de R. Este termo ("non-erasing") não tem uma tradução satisfatória em português. Seu sentido aproximado é "não-apagante", ou seja, um homomorfismo que nunca elimina símbolos da fórmula.

# 7 Completude NP

## 7.1 REDUÇÕES DE TEMPO POLINOMIAL

Muitos dos conceitos e técnicas utilizadas em teoria da complexidade são análogas aos conceitos e técnicas que desenvolvemos para estudar a indecidibilidade, porém limitadas no tempo. Na prova do Teorema 6.1.2 pudemos utilizar uma variante da diagonalização de resposta polinomial no tempo para mostrar que certas linguagens não pertencem a $\mathcal{P}$. Apresentamos a seguir as *reduções de tempo polinomial*, o análogo das reduções que empregamos no Capítulo 5 para determinar a indecidibilidade. Utilizaremos reduções de tempo polinomial para estudar a complexidade de vários problemas importantes e aparentemente difíceis da classe $\mathcal{NP}$, apresentados no capítulo anterior: o problema da SATISFATIBILIDADE, o PROBLEMA DO CAIXEIRO-VIAJANTE, o PROBLEMA DO CONJUNTO INDEPENDENTE, o PROBLEMA DA PARTIÇÃO e outros. Como veremos, esses problemas compartilham a seguinte importante propriedade de *completude*: *todos os* problemas em $\mathcal{NP}$ podem ser reduzidos a eles por meio de reduções de tempo polinomial – de maneira análoga à maneira como todas as linguagens recursivamente enumeráveis se reduzem ao problema da parada da máquina de Turing. Tais problemas demoninam-se **$\mathcal{NP}$-completos.**

Infelizmente, a analogia com o problema de parada só vai até aí. Parece não haver sequer um único argumento de diagonalização que determine que os problemas $\mathcal{NP}$-completos não pertencem a $\mathcal{P}$; os argumentos baseados na diagonalização aparentemente se aplicam somente a linguagens difíceis e artificiais, tal como ocorre com a linguagem $E$ de tempo exponencial do Teorema 6.1.2.

Embora tais problemas $\mathcal{NP}$-completos não permitam provar que $\mathcal{P} \neq \mathcal{NP}$, eles realmente ocupam um importante lugar em nosso estudo da complexidade: *se* admitirmos que $\mathcal{P} \neq \mathcal{NP}$ – uma conjectura amplamente aceita, embora longe de estar provada – então, como conseqüencia, todos os problemas $\mathcal{NP}$-completos não pertencerão a $\mathcal{P}$ (isso é discutido no Teorema 7.1.1 adiante). Essa evidência indireta de dificuldade é o máximo que podemos garantir para um problema da classe $\mathcal{NP}$, sem provar que $\mathcal{P} \neq \mathcal{NP}$.

Vamos então definir a variante de tempo polinomial da redução (comparar com a Definição 5.4.1):

---

**Definição 7.1.1:** Diz-se que a função $f$: $\Sigma^* \to \Sigma^*$ é **computável em tempo polinomial,** se for computável por uma máquina de Turing $M$ polinomialmente limitada.

Sejam agora $L_1$, $L_2 \subseteq \Sigma^*$ duas linguagens. Uma função computável em tempo polinomial $\tau$: $\Sigma^* \to \Sigma^*$ é dito uma **redução polinomial de $L_1$ para $L_2$** se, para cada $x \in \Sigma^*$, $x \in L_1$ se e somente se $\tau(x) \in L_2$.

---

Reduções polinomiais são importantes porque revelam interessantes afinidades entre problemas computacionais. Estritamente falando, uma redução polinomial, como a acima definida, relaciona duas linguagens, não dois problemas. Entretanto sabemos, com base na discussão da Seção 6.2, que podemos usar linguagens para codificar todos os tipos de problemas computacionais importantes, como o CIRCUITO DE HAMILTON, a SATISFATIBILIDADE e o CONJUNTO INDEPENDENTE. Nesse sentido, podemos dizer que $\tau$ é uma **redução polinomial do Problema A para o Problema** B se for uma redução polinomial entre as correspondentes linguagens, isto é, $\tau$ transforma, em tempo polinomial, instâncias do Problema A para instâncias do Problema B, de tal modo que $x$ é uma instância do Problema A com resposta "sim", se e somente se $\tau(x)$ for uma instância do Problema B que também fornece a resposta "sim".

Tendo-se uma redução polinomial $\tau$ do Problema A para o Problema B, é possível adaptar qualquer algoritmo de tempo polinomial de B para obter outro para A (ver a Figura 7-1). Para dicidir se uma instância qualquer $x$ do Problema A é de fato uma instância de A com resposta "sim", pode-se inicialmente computar $\tau(x)$ e testar se essa é ou não uma instância de B com resposta "sim". Tendo um algoritmo polinomial para B, esse método também resolve A em tempo polinomial, uma vez que os passos responsáveis pela redução e o algoritmo que resolve a instância de B podem ser executados em tempo polinomial. Em outras palavras, a existência de uma redução polinomial de A para B é uma evidência de que B é no *mínimo mais complexo que* A. Se B for eficientemente solúvel, então A também deverá ser, e, se A exigir um tempo exponencial, então B também exigirá.

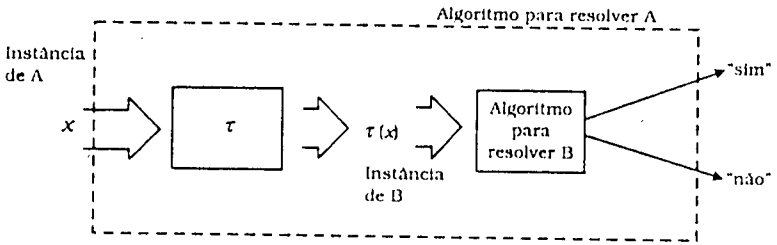



Figura 7-1

Apresentamos algum exemplos de reduções a seguir.

**Exemplo 7.1.1:** Vamos descrever uma redução polinomial do CIRCUITO DE HAMILTON para o problema da SATISFATIBILIDADE. Seja dada uma instância do CIRCUITO DE HAMILTON, isto é, um grafo $G \subseteq V \times V$, onde $V = \{1, 2, \ldots, n\}$. Devemos descrever um algoritmo $\tau$ que produza uma fórmula Booleana em forma normal conjuntiva $\tau(G)$, tal que $G$ apresente um circuito de Hamilton (um caminho fechado que visita cada vértice de $G$ exatamente uma vez) se e somente se $\tau(G)$ é satisfazível.

A fórmula $\tau(G)$ envolverá $n^2$ variáveis Booleanas, as quais denotamos $x_{ij}$, com $1 \leq i, j \leq n$. Intuitivamente, $x_{ij}$ será uma variável Booleana com o seguinte *significado*: "O vértice $i$ de $G$ é o $j$-ésimo vértice do circuito de Hamilton de $G$". As várias cláusulas de $\tau(G)$ então expressarão, na linguagem da lógica Booleana, os vínculos que um circuito de Hamilton deve satisfazer.



Devemos construir cada vínculo desse tipo como uma cláusula. Para $j = 1, \ldots, n$ temos a cláusula

$$(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj}).$$

Para uma atribuição satisfazer essa cláusula, ela deve impor, para no mínimo uma das variáveis $x_{1j}, x_{2j}, \ldots, x_{nj}$, o valor T; portanto, essa cláusula expressa (interpretando suas variáveis Booleanas) o vínculo de que no mínimo um vértice deve aparecer na $i$-ésima posição do circuito de Hamilton.

Naturalmente, um único vértice de $G$ pode ser o $i$-ésimo elemento do circuito de Hamilton. Portanto, acrescentamos à nossa fórmula Booleana todas as $O(n^3)$ cláusulas da forma

$$(\overline{x_{ij}} \vee \overline{x_{ik}})$$

para $i, j, k = 1, \ldots, n$ e $j \neq k$. Como no mínimo um literal nessa cláusula deve ser satisfeito, essas cláusulas expressam o vínculo de que tanto o vértice $j$ como o vértice $k$ podem aparecer como $i$-ésimo no circuito.

Até aqui as cláusulas garantem que exatamente um vértice apareça como $i$-ésimo elemento do circuito de Hamilton. Mas é ainda necessário impor que o vértice $i$ apareça exatamente uma vez no circuito. Isso pode ser expresso pelas cláusulas

$$(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$$

para $i = 1, \ldots, n$ e

$$(\overline{x_{ij}} \vee \overline{x_{kj}})$$

para $i, j, k = 1, \ldots, n$ e $i \neq k$.

Até aqui essas cláusulas expressam o vínculo de que os $x_{ij}$'s representam uma *bijeção* ou *permutação* dos vértices de $G$. Devemos a seguir expressar com novas cláusulas a exigência de que essa permutação seja, de fato, um circuito de $G$. Fazemos isso adicionando a cláusula

$$(\overline{x_{ij}} \vee \overline{x_{k,j+1}}) \qquad (1)$$

para $j = 1, \ldots, n$ e *para cada par (i, k) de vértices tal que (i, k) não seja uma aresta de G*. Aqui com $x_{k,n+1}$ queremos indicar a variável Booleana $x_{k1}$; isto é, a operação de adição, no segundo índice, é feita módulo $n$. Intuitivamente, as cláusulas em (1) afirmam que, se não existir uma aresta $(i, k)$ em $G$, então $i$ e $k$ não poderão aparecer em posições sucessivas no suposto circuito de Hamilton. Isso completa a construção de $\tau(G)$.

É fácil argumentar que a construção de $\tau(G)$ pode ser efetuada em tempo polinomial. Há $O(n^3)$ cláusulas a serem construídas, com um número total de $(n^3)$ literais. A estrutura dessas cláusulas é extremamente simples e

depende apenas de $n$ ou então depende das arestas de $G$, segundo uma regra simples e direta, o que torna relativamente fácil construir uma máquina de Turing de tempo polinomial que computa a função $\tau$.

A seguir, devemos discutir que $G$ possuirá um circuito de Hamilton se e somente se $\tau(G)$ for satisfazível. Havendo alguma atribuição $T$ que satisfaça $\tau(G)$, como $T$ deve impor que, no mínimo um literal seja $\top$ em cada uma das cláusulas exceto em (1) acima, conclui-se que $T$ codifica uma bijeção sobre os vértices de $G$, isto é, para cada $i$ exatamente um $T(x_{i,j})$ vale $\top$, e para cada $j$ exatamente um $T(x_{i,j})$ vale $\top$. Denota-se $\pi(i)$ o único $j$ para o qual $T(x_{i,\pi(i)}) = \top$.

O conjunto de cláusulas em (1) acima também deve ser satisfeito. Isso significa que sempre que $i = \pi(j)$ e $k = \pi(j + 1)$ (onde, novamente por causa da soma modulo $n$, $n + 1$ vale 1) então $(j, k)$ deverá ser uma aresta de $G$. Segue-se que $(\pi(1), \pi(2), \ldots, \pi(n))$ é de fato um circuito de Hamilton de $G$ e a parte *se* do teorema está provada.

Inversamente, seja $G$ um grafo que apresenta um circuito de Hamilton, digamos $(\pi(1), \pi(2), \ldots, \pi(n))$. Então, é fácil ver que a atribuição $T$, onde $T(x_{ij}) = \top$ se e somente se $j = \pi(i)$, satisfaz todas as cláusulas de $\pi(G)$ e a prova está completa.

Posteriormente, nesse capítulo, é apresentada uma redução *no sentido oposto* (Teoremas 7.3.1 e 7.3.2).

Pode ser difícil e exigir cuidado e experiência a correta interpretação das reduções polinomiais quanto às informações que elas revelam acerca da dificuldade dos problemas envolvidos. O leitor é convidado, neste ponto, a ponderar se tal redução daria uma informação positiva ou negativa acerca da complexidade do CIRCUITO DE HAMILTON e do problema de SATISFATIBILIDADE. Tendo um algoritmo de tempo polinomial para o CIRCUITO DE HAMILTON, o que se poderia concluir sobre o problema da SATISFATIBILIDADE, com base NESSA REDUÇÃO? Tendo um algoritmo de tempo polinomial para o problema da SATISFATIBILIDADE, o que se poderia concluir admitindo que o circuito de Hamilton seja um problema difícil? ◊

**Exemplo 7.1.2:** Deve-se, neste caso, escalonar $n$ tarefas entre duas máquinas. Ambas têm a mesma velocidade, cada tarefa pode ser executada em qualquer máquina e não há restrições na ordem em que as tarefas têm de ser executadas. São dados os tempos de processamento $a_1, \ldots, a_n$ das tarefas e um prazo $D$, todos em binário. É possível completar todas essas tarefas usando as duas máquinas, dentro desse prazo?

Em outras palavras, há como particionar o conjunto de números binários dados em dois outros, tais que a soma dos números de cada conjunto seja no máximo $D$? Chamamos esse problema, que tem uma clara relação com o problema da PARTIÇÃO, definido no último capítulo, ESCALONAMENTO DE DUAS MÁQUINAS. Apresentamos ainda outro problema, muito mais intimamente relacionado com o problema da PARTIÇÃO:

> MOCHILA: Dado um conjunto $S = \{a_1, \ldots, a_n\}$ de inteiros não-negativos e um inteiro $K$, todos representados em binário, existe um subconjunto $P \subseteq S$, tal que $\left(\sum_{a_i \in P} a_i\right) = K$?

O nome deste problema foi escolhido em alusão a um viajante que tenta encher sua mochila até o seu limite, com itens de pesos variados.)



Esses três problemas (PARTIÇÃO, MOCHILA e AGENDAMENTO DE DUAS MÁQUINAS) podem ser relacionados através de reduções polinomiais, e serão apresentadas *seis reduções polinomiais*, capazes de reduzir qualquer um desses três problemas a qualquer dos outros dois.

Suponhamos uma instância de MOCHILA, com inteiros $a_1, \ldots, a_n$ e $K$; desejamos reduzi-la a uma instância correspondente de PARTIÇÃO. Se $K$ for igual a $H = \frac{1}{2}\sum_{i=1}^{n} a_i$, a metade da soma de todos os inteiros fornecidos, então, a redução desejada precisa apenas remover $K$ da entrada: a instância resultante da partição será equivalente à instância fornecida de MOCHILA. O problema é, naturalmente, que $K$ não será, em geral, igual a $H$. Mas isso pode ser remediado com facilidade acrescentando ao conjunto de $a_i$'s dois novos inteiros grandes $a_{n+1} = 2H + 2K$ e $a_{n+2} = 4H$ (note que esses números são inteiros mesmo que $H$ não seja). Considere-se agora o conjunto de inteiros resultante $\{a_1, \ldots, a_{n+2}\}$ como uma instância de PARTIÇÃO, e a redução estará completa. É óbvio que essa redução pode ser executada em tempo polinomial.

Devemos agora mostrar que a redução funciona corretamente, isto é, que a instância de partição terá solução se e somente se a instância original de mochila tiver solução. Se o novo conjunto de inteiros puder ser particionado em dois conjuntos com somas iguais, então os dois novos inteiros $a_{n+1}$ e $a_{n+2}$ devem estar em conjuntos diferentes (sua soma sempre excede a dos inteiros restantes). Seja $P$ o subconjunto de $S = \{a_1, \ldots, a_n\}$ presente no mesmo conjunto que $a_{n+2} = 4H$. Temos que

$$4H + \sum_{a_i \in P} a_i = 2H + 2K + \sum_{a_i \in S-P} a_i$$

Somando $\Sigma_{a_i \in P} a_i$ aos dois lados da igualdade, e considerando que $\Sigma_{a_i \in S} a_i = 2H$, obtemos

$$4H + 2\sum_{a_i \in P} a_i = 4H + 2K,$$

ou $\Sigma_{a_i \in P} a_i = K$. Portanto, se a instância resultante da PARTIÇÃO tiver solução, então a instância original de MOCHILA também terá. Inversamente, tendo uma solução de MOCHILA, então o acréscimo de $a_{n+2}$ a ela produzirá uma solução da instância de PARTIÇÃO.

Isto ilustra a redução de MOCHILA para PARTIÇÃO. Iniciamos com uma instância arbitrária de MOCHILA e construímos uma instância equivalente de PARTIÇÃO. Uma redução inversa é trivial, porque PARTIÇÃO é um *caso particular* de MOCHILA. Dada qualquer instância de PARTIÇÃO com inteiros $a_1, \ldots, a_n$, a redução para MOCHILA transforma a instância fornecida de PARTIÇÃO na instância de MOCHILA contendo os mesmos números e o limite $H = \frac{1}{2}\sum_{i=1}^{n} a_i$.

Porém, se $K$ não for um número inteiro, isto é, se a soma dos inteiros fornecidos for um número ímpar, podemos obter a redução a partir de uma instância impossível arbitrária de MOCHILA (por exemplo, com $n = 1$, $a_1 = 2$ e $K = 1$). A instância fornecida de PARTIÇÃO também era impossível.

A redução de PARTIÇÃO para ESCALONAMENTO DE DUAS MÁQUINAS também é fácil: dada uma instância de PARTIÇÃO, com inteiros $a_1, \ldots, a_n$, a redução produz uma instância de ESCALONAMENTO DE DUAS MÁQUINAS com $n$ tarefas, tempos de execução $a_1, \ldots, a_n$ e prazo $D = \lfloor \frac{1}{2} \sum_{i=1}^{n} a_i \rfloor$ (se $\frac{1}{2}\sum_{i=1}^{n} a_i$ não é um inteiro, então é óbvio que esta é uma instância impossível). É fácil ver que o resultado da instância de ESCALONAMENTO DE DUAS MÁQUINAS será solúvel se e somente se a instância original de PARTIÇÃO também o for. Isso se deve ao fato de as tarefas poderem ser particionadas em dois conjuntos com somas de, no máximo, $D$ se e somente se elas puderem ser particionadas em dois conjuntos com somas *exatamente* iguais a $D$.

A redução de ESCALONAMENTO DE DUAS MÁQUINAS para PARTIÇÃO é um pouco mais elaborada. Seja dada uma instância de ESCALONAMENTO DE DUAS MÁQUINAS com tarefas de comprimentos $a_1, \ldots, a_n$ e prazo $D$. Seja $I = 2D - \sum_{i=1}^{n} a_i$. Intuitivamente, $I$ é o *tempo* total *de inatividade* para qualquer agenda válida (veja a Figura 7-2). Ele representa a *folga* disponível na resolução do problema agendado. Adicionamos agora vários novos números ao conjunto de comprimentos de tarefa, tal que (a) a soma desses novos números seja $I$; e (b) além disso, possamos obter qualquer soma entre 0 e $I$ por meio da adição de algum subconjunto desses novos números. Com isso, o resultado da instância de PARTIÇÃO seria equivalente à instância original de ESCALONAMENTO DE DUAS MÁQUINAS, porque então poderíamos transformar qualquer escala viável das tarefas originais em uma partição equitativa do novo conjunto de números, bastando para isso acrescentar a cada um dos dois conjuntos um subconjunto dos números recém-introduzidos escolhido de tal forma que faça a soma de ambos chegar a $D$.

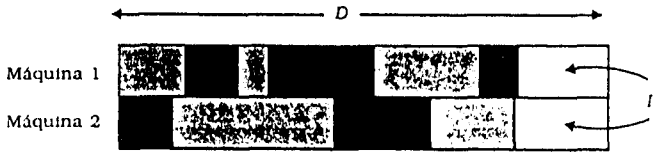



Figura 7-2

Resta escolher tais inteiros, acrescentá-los a $I$, de maneira que qualquer número entre zero e $I$ possa ser obtido pela soma de um subconjunto desses números. A primeira vista isso parece trivial, bastando adicionar $I$ cópias do inteiro 1. Entretanto, essa redução *não se processa em tempo polinomial*, pela mesma razão que o algoritmo para PARTIÇÃO, que esboçamos no capítulo anterior, não é polinomial: o inteiro $I$ não é limitado por um polinômio sobre comprimento da entrada – sendo que a entrada consiste dos comprimentos das tarefas e do prazo, todos codificados em binário, o número $I$ pode ser exponencialmente grande em relação ao tamanho da entrada.

Existe uma solução um pouco mais complicada: todos os números que adicionamos são potências de 2 menores do que $\frac{1}{2}I$, além de um outro inteiro suficiente para completar a soma no valor $I$. Por exemplo, se $I = 56$, então os inteiros a serem acrescentados seriam 1, 2, 4, 8, 16 e 25. Todos os inteiros entre 0 e 56, e somente esses, podem ser compostos somando-se algum subconjunto desses inteiros. Isso completa a descrição da redução de ESCALONAMENTO DE DUAS MÁQUINAS para PARTIÇÃO: essa redução é polinomial, porque o número de inteiros que introduzimos é limitado pelo *logaritmo de* $I$ – o qual é menor do que o comprimento da entrada. ◊



No exemplo acima não foram discutidas reduções diretas de MOCHILA para ESCALONAMENTO DE DUAS MÁQUINAS e vice-versa, mas isso é desnecessário, pois, como mostraremos a seguir, reduções polinomiais compõem-se de um modo transitivo. Convém recordar que a composição de duas funções $f: A \to B$ e $g: B \to C$ é a função $f \circ g: A \to C$, onde para todos os $x \in A$, $f \circ g(x) = g(f(x))$.

---

**Lema 7.1.1:** *Se $\tau_1$ é uma redução polinomial de $L_1$ para $L_2$ e $\tau_2$ é uma redução polinomial de $L_2$ para $L_3$, então sua composição $\tau_1 \circ \tau_2$ é uma redução polinomial de $L_1$ para $L_3$.*

---

***Prova:*** Seja $\tau_1$ computada por uma máquina de Turing $M_1$ no tempo polinomial $p_1$, e $\tau_2$ computada por $M_2$ em tempo $p_2$, também polinomial. Então $\tau_1 \circ \tau_2$ pode ser computado pela máquina $M_1M_2$. Em resposta à entrada $x \in \Sigma_1^*$, $M_1M_2$ dará como saída $\tau_1 \circ \tau_2(x)$ em um tempo limitado por $p_1(|x|) + p_2(p_1(|x|))$ – que é um polinômio. O último termo reflete o fato de que $|\tau_1(x)|$ não pode ser maior que $p_1(|x|)$.

Falta demonstrar que $x \in L_1$ se e somente se $\tau_1 \circ \tau_2(x) \in L_3$. Mas isso é trivial: $x \in L_1$ se e somente se $\tau_1(x) \in L_2$, e também $\tau_1(x) \in L_2$ se e somente se $\tau_1 \circ \tau_2(x) \in L_3$. ■

Chegamos finalmente à seguinte importante definição.

---

**Definição 7.1.2:** Uma linguagem $L \subseteq \Sigma^*$ é dita **$\mathcal{NP}$-completa se**
(a) $L \in \mathcal{NP}$, e
(b) para qualquer linguagem $L' \in \mathcal{NP}$ existe alguma redução polinomial de $L'$ para $L$.

---

Exatamente como a decidibilidade de $H$ sintetizou toda a questão da decidibilidade das linguagens Turing-aceitáveis, também a questão de se qualquer linguagem $\mathcal{NP}$-completa está em $\mathcal{P}$ revela-se como sendo equivalente à questão $\mathcal{P} = \mathcal{NP}$.

---

**Teorema 7.1.1:** *Seja $L$ uma linguagem $\mathcal{NP}$-completa. Então, $\mathcal{P} = \mathcal{NP}$ se e somente se $L \in \mathcal{P}$.*

---

***Prova:*** (*Somente se*) Suponhamos que $\mathcal{P} = \mathcal{NP}$. Como $L$ é $\mathcal{NP}$-completa e, portanto, $L \in \mathcal{NP}$ (já que toda linguagem $\mathcal{NP}$-completa deve estar em $\mathcal{NP}$), segue-se que $L \in \mathcal{P}$.

(*Se*) Seja $L$ uma linguagem $\mathcal{NP}$-completa que é decidida por uma máquina de Turing determinística $M_1$ no tempo polinomial $p_1(n)$ e seja $L'$ qualquer linguagem em $\mathcal{NP}$; devemos mostrar que $L' \in \mathcal{P}$.

Como $L$ é $\mathcal{NP}$-completa e $L' \in \mathcal{NP}$, então existirá alguma redução polinomial $\tau$ de $L'$ para $L$ (agora estamos utilizando a segunda parte da defi-

nição das linguagens $\mathcal{NP}$-completas). Suponha que $\tau$ seja computada por alguma máquina de Turing $M_2$ no tempo $p_2(n)$, também polinomial. Então, pergunta-se se a máquina de Turing $M_2M_1$ decide $L'$ em tempo polinomial. Para verificar que $M_2$ $M_1$ decide $L'$, note-se que $M_2M_1$ aceita a entrada $x$ se e somente se $\tau(x) \in L$; e uma vez que $\tau$ é uma redução polinomial, $\tau(x) \in L$ se e somente se $x \in L'$.

Finalmente, para analisar os requisitos de tempo de $M_2M_1$, note-se que sua parte inicial $M_2$ gasta, em resposta à entrada $x$, o tempo $p_2(|x|)$ para produzir uma entrada para $M_1$. Essa entrada terá comprimento de, no máximo, $p_2(|x|)$, porque $M_2$ grava apenas um símbolo por passo. Portanto, a computação de $M_1$ sobre essa entrada gastará, no máximo, o tempo $p_1(p_2(|x|))$. A máquina global irá parar, em resposta à entrada $x$, em um tempo $p_2(|x|) + p_1(p_2(|x|) + |x|)$, e este é um polinômio em $|x|$.

Como $L'$ foi definida, por hipótese, como sendo uma linguagem em $\mathcal{NP}$, e como concluímos anteriormente que $L' \in \mathcal{P}$, segue-se que $\mathcal{NP} = \mathcal{P}$. ■

### Problemas para a Seção 7.1

**7.1.1** Em 3-COLORAÇÃO, é dado um grafo não-orientado, e se pergunta se seus vértices podem ser coloridos usando três cores tais que não haja dois vértices adjacentes da mesma cor.
(a) Mostre que 3-COLORAÇÃO está em $\mathcal{NP}$.
(b) Descreva uma redução de tempo polinomial de 3-COLORAÇÃO para SATISFAZIBILIDADE.

**7.1.2** Alguns autores definem uma noção mais geral de redução, muitas vezes chamada **redução polinomial de Turing**. Sejam $L_1$ e $L_2$ duas linguagens. Uma **redução polinomial de Turing** de $L_1$ para $L_2$ é uma máquina de Turing de duas fitas com três estados distintos, $q_?$, $q_1$ e $q_0$, que decide $L_2$ em tempo polinomial e cuja computação é definida a seguir. A relação de produção de $M$ para todas as configurações envolvendo estados diferentes de $q_?$ é definida exatamente como para máquinas de Turing usuais. Para $q_?$, porém, dizemos que $(q_?, \triangleright u_1\underline{a}_1v_1, \triangleright u_2\underline{a}_2v_2) \vdash_M (q, \triangleright u'_1\underline{a}'_1v'_1, \triangleright u'_2\underline{a}'_2v'_2)$ se e somente se
(1) $u_1 = u'_1$, $a'_1 = a_1$, $v'_1 = v_1$, $u_2 = u'_2$, $a'_2 = a_2$, $v'_2 = v_2$, e
(2) um dos itens seguintes se aplica:
(2a) $u_2a_2v_2 \in L_1$ e $q' = q_1$, ou
(2b) $u_2a_2v_2 \notin L_1$ e $q' = q_0$.
Em outras palavras, a partir do estado $q_?$, $M$ nunca troca nenhuma de suas fitas; ele apenas vai para o estado $q_1$ ou $q_0$, *dependendo de estar ou não a cadeia contida em sua segunda fita em* $L_1$. Além disso, *isso é contado como um passo de* $M$.
(a) Mostre que se existe uma redução polinomial de Turing de $L_1$ para $L_2$ e uma de $L_2$ para $L_3$, então há uma redução polinomial de Turing de $L_1$ para $L_3$.
(b) Mostre que se existe uma redução polinomial de Turing de $L_1$ para $L_2$ e $L_2 \in \mathcal{P}$, então $L_1 \in \mathcal{P}$.



(c) Apresente uma redução polinomial de Turing do CIRCUITO DE HAMILTON para CAMINHO DE HAMILTON (a versão em que não se exige que o caminho que visita cada vértice exatamente uma vez seja fechado). É possível apresentar uma redução polinomial direta (que não empregue a redução utilizada na prova do Teorema 7.3.2 abaixo) entre esses dois problemas?

## 7.2 TEOREMA DE COOK

Ainda não mostramos que linguagens $\mathcal{NP}$-completas existem – mas elas existem. Nas duas últimas décadas, as pesquisas sobre complexidade computacional revelaram centenas dessas linguagens $\mathcal{NP}$-completas (ou *problemas $\mathcal{NP}$-completos*, uma vez que devemos continuar a explorar as similaridades entre problemas computacionais, tais como SATISFATIBILIDADE E CIRCUITO DE HAMILTON e as linguagens que os codificam). Muitos desses problemas $\mathcal{NP}$-completos correspondem a importantes problemas práticos da pesquisa operacional, lógica, matemática combinatória, inteligência artificial e outras áreas de aplicação. Antes do descobrimento da completude $\mathcal{NP}$, dedicou-se em vão muito esforço para encontrar algoritmos polinomiais para muitos desses problemas. O conceito de *completude $\mathcal{NP}$* unifica as experiências de pesquisadores de diversas áreas, mostrando que nenhum desses problemas é solúvel por meio de algoritmos de tempo polinomial, a não ser que $\mathcal{P} = \mathcal{NP}$ – uma circunstância que contradiz tanto as intuições como as experiências. Essa percepção produziu o efeito benéfico de desviar os esforços de pesquisa previamente focalizados na solução de problemas particulares $\mathcal{NP}$-completos, em direção a outras metas mais tratáveis, que serão estudadas na seção 7.4. Esse redirecionamento dos esforços da pesquisa constituiu o mais profundo impacto da teoria da computação sobre as práticas computacionais.

### Ladrilhamento limitado

Uma vez provado que um primeiro problema é $\mathcal{NP}$-completo, outros problemas podem ser identificados como $\mathcal{NP}$-completos, reduzindo-os a um exemplar conhecido de problema $\mathcal{NP}$-completo e utilizando a transitividade das reduções polinomiais (conforme o Lema 7.1.1). Mas a primeira prova de completude $\mathcal{NP}$ deve ser uma aplicação da definição: devemos determinar que todos os problemas em $\mathcal{NP}$ se reduzem ao problema dado. Historicamente, o primeiro problema provado como sendo $\mathcal{NP}$-completo por Stephen A. Cook, em 1971, foi o da SATISFATIBILIDADE. Em vez de apresentar essa prova diretamente, começaremos com uma versão do problema do ladrilhamento, que foi mostrado como sendo indecidível no Capítulo 5.

No problema original do ladrilhamento é dado um sistema de ladrilhamento $D$, e foi perguntado se existe algum modo de ladrilhar o primeiro quadrante (infinito), de maneira que quaisquer dois ladrilhos vertical ou horizontalmente adjacentes estejam relacionados conforme prescrito, e um dado ladrilho seja posicionado na origem. Podemos definir um problema

menos ambicioso, o LADRILHAMENTO LIMITADO, no qual se pergunta se existe algum ladrilhamento válido, não de todo o quadrante, *mas de um quadrado de dimensão* $s \times s$, onde $s > 0$ é um inteiro fornecido. Desta vez, em lugar de especificar somente o ladrilho posicionado em (0, 0), especificamos toda a primeira linha de ladrilhos, isto é, dado um sistema de ladrilhamento $\mathcal{D} = (D, H, V)$ (onde está omitido o ladrilho inicial, agora irrelevante), um inteiro $s > 0$ e uma função $f_0$: $\{0, \ldots, s-1\} \to D$. Pergunta-se se existe algum ladrilhamento $s \times s$ por $\mathcal{D}$ que estende $f_0$, isto é, uma função $f$: $\{0, 1, \ldots, s-1\} \times \{0, 1, \ldots, s-1\} \to D$, tal que

$f(m, 0) = f_0(m)$ para todos os $m < s$;
$(f(m, n), f(m+1, n)) \in H$ para todos os $m < s-1$, $n < s$;
$(f(m, n), f(m, n+1)) \in V$ para todos os $m < s$, $n < s-1$.

O problema do LADRILHAMENTO LIMITADO pode ser enunciado da seguinte forma:

"Dado um sistema de ladrilhamento $\mathcal{D}$, um inteiro $s$ e uma função $f_0$: $\{0, \ldots, s-1\} \to D$, representada por sua seqüência de valores $(f_0(0), \ldots, f_0(s-1))$, existe um ladrilhamento $s \times s$ por $\mathcal{D}$ que estende $f_0$?"

**Teorema 7.2.1:** *O LADRILHAMENTO LIMITADO é $\mathcal{NP}$-completo.*

***Prova:*** Vamos primeiro discutir que este problema pertence à classe $\mathcal{NP}$. O certificado neste caso é uma lista completa dos valores $s^2$ de uma função de ladrilhamento $f$. É possível verificar em tempo polinomial se esta função concorda com as três exigências. Além disso, ela é sucinta: seu comprimento total é $s^2$ vezes o número de símbolos que ela necessita para representar um ladrilho, e $s$ é limitado superiormente pelo comprimento da cadeia de entrada, *porque a entrada inclui a listagem de* $f_0$. Na realidade, o propósito dessa alteração do nosso formalismo do ladrilhamento foi precisamente o de assegurar que o problema esteja em $\mathcal{NP}$: se somente especificarmos um ladrilho inicial, o problema torna-se muito difícil – pode-se demonstrar que ele não está em $\mathcal{P}$ (ver o Problema 7.2.2).

Mostramos a seguir que todas as linguagens em $\mathcal{NP}$ se reduzem através de reduções polinomiais, ao LADRILHAMENTO LIMITADO. Assim, considere-se uma linguagem arbitrária $L \in \mathcal{NP}$. Apresentamos uma redução polinomial de $L$ para o LADRILHAMENTO LIMITADO, isto é, uma função computável em tempo polinomial $\tau$, tal que para cada $x \in \Sigma^*$, $\tau(x)$ codifica um sistema de ladrilhamento $\mathcal{D} = (D, H, V)$, bem como um inteiro $s > 0$ e a codificação de uma função $f_0$, tal que exista um ladrilhamento $s \times s$ com $\mathcal{D}$ que estende $f_0$ se e somente se $x \in L$.

Para obter essa redução, é preciso de alguma forma explorar as poucas informações disponíveis acerca de $L$. Tudo o que sabemos sobre $L$ é que ela é uma linguagem em $\mathcal{NP}$, ou seja, sabemos que existe alguma máquina de Turing não-determinística $M = (K, \Sigma, \delta, s)$, tal que (a) todas as computações de $M$ sobre a entrada $x$ param após $p(|x|)$ passos para algum polinômio $p$, e (b) existirá uma computação de aceitação sobre entrada $x$ se e somente se $x \in L$.

O inteiro $s$ construído por $\tau$ para a entrada $x$ é $s = p(|x|) + 2$, dois a mais do que o tempo limite de $M$ na entrada $x$.

O sistema de ladrilhamento $\mathcal{D}$ descrito em $\tau(x)$ é muito similar ao que foi construído na prova da indecidibilidade do problema do ladrilhamento ilimitado (Teorema 5.6.1). Descrevem-se os ladrilhos em $\mathcal{D}$ por suas marcações nas arestas; mais uma vez, as marcações das arestas horizontais, entre as linhas $t$ e $t+1$, representarão o conteúdo da fita de $M$ em uma computação válida, com entrada $x$, logo após o $t$-ésimo passo (sendo $M$ não determinística, pode haver várias computações desse tipo e portanto pode haver várias maneiras possíveis e válidas de ladrilhar a área $s \times s$).



A 0-ésima linha do quadro $s \times s$, descrita por $f_0$, será ocupada pelos ladrilhos que compõem a configuração inicial $(s, \triangleright \sqcup x)$, ou seja, $f_0(0)$ é um ladrilho com aresta superior marcada com o símbolo $\triangleright$, $f_0(1)$ é um ladrilho com aresta superior marcada com $(s, \sqcup)$, e para $i = 1, \ldots, |x|$ $f_0(i+1)$ é um ladrilho com aresta superior marcada com $x_i$, o $i$-ésimo símbolo da cadeia de entrada $x$. Finalmente, para todos os $i > |x| + 1$, $f_0(i)$ é um ladrilho com aresta superior marcada com $\sqcup$ (veja a Figura 7-3). Portanto, as marcações nas arestas horizontais entre a 0-ésima e as primeiras linhas descreverão corretamente a configuração inicial de $M$ para a entrada $x$.

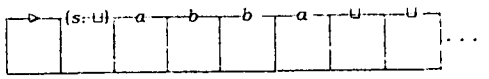



Figura 7-3

Os ladrilhos restantes em $\mathcal{D}$ são exatamente idênticos aos da prova do Teorema 5.6.1. Como a máquina é não-determinística, pode haver mais de um ladrilho com marcação horizontal inferior $(q, a) \in K \times \Sigma$, possivelmente correspondente às muitas escolhas de ação, quando $M$ lê um símbolo $a$ no estado $q$, e cada um deles é construído como naquela prova. Há somente uma pequena diferença: existe um ladrilho com os lados superior e inferior marcados com $(y, a)$ para cada símbolo $a$, efetivamente permitindo que a organização lado a lado continue após a computação ter atingido o estado $y$ – mas não no caso de ele parar no estado $n$. Isso completa a construção da instância $\tau(x)$ do LADRILHAMENTO LIMITADO. Naturalmente a construção dessa instância pode ser realizada em um tempo polinomial em $|x|$.

Mostramos a seguir que existe um ladrilhamento $s \times s$ efetuado por $\mathcal{D}$ se e somente se $x \in L$. Suponhamos que exista um ladrilhamento $s \times s$. Nossa definição de $f_0$ assegura que as marcações horizontais das arestas entre a 0-ésima e a primeira linha devem transcrever corretamente a configuração inicial de $M$ para a entrada $x$. Por indução, decorre que as marcações horizontais das arestas compreendidas entre a $n$-ésima e a $n+1$-ésima linhas transcreverão corretamente a configuração logo após o $n$-ésimo passo de alguma computação legal de $M$ sobre a entrada $x$. Como nenhuma computação de $M$ sobre a entrada $x$ dura mais que $p(|x|) = s-2$ passos, as marcações superiores da $s-2$-ésima linha devem obrigatoriamente conter um dos símbolos $y$ ou $n$. Como existe uma $s-1$-ésima linha e não existe ladrilho com marcações inferiores $n$, devemos concluir que o símbolo $y$ surge e que, portanto, a computação é aceita. Concluímos que, se algum ladrilhamento do quadro $s \times s$ com $\mathcal{D}$ existir, então $M$ aceitará $x$.

Inversamente, se existir alguma computação de aceitação da máquina $M$ para a entrada $x$, então, ela pode ser facilmente simulada por um ladri-

lhamento (possivelmente repetindo a última linha várias vezes, caso a computação termine em menos que $p(|x|)$ passos no estado $y$), e isso completa esta prova. ■

Podemos agora utilizar o ladrilhamento limitado para provar o principal resultado desta seção:

---

**Teorema 7.2.2 (Teorema de Cook):** o problema da SATISFATIBILIDADE é $\mathcal{NP}$-completo.

---

***Prova:*** Já apresentamos argumentos que garantem que o problema está em $\mathcal{NP}$; devemos a seguir reduzir o LADRILHAMENTO LIMITADO para o problema da SATISFATIBILIDADE.

Dada qualquer instância do problema do LADRILHAMENTO LIMITADO, digamos, o sistema de ladrilhamento $\mathcal{D} = (D, H, V)$, com tamanho $s$ e linha inferior $f_0$, onde $D = \{d_1, \ldots, d_k\}$, vamos mostrar como construir uma fórmula Booleana $\tau(\mathcal{D}, s, f_0)$, tal que $\mathcal{D}$ executa um ladrilhamento $f$ de dimensão $s \times s$ se e somente se $\tau(\mathcal{D}, s)$ é satisfazível.

As variáveis Booleanas em $\tau(\mathcal{D}, s, f_0)$ são $x_{mnd}$ para cada $0 \leq m, n < s$ e $d \in D$. A correspondência pretendida entre essas variáveis e o problema do ladrilhamento é que a variável $x_{mnd}$ é $\top$ se, e somente se, $f(m, n) = d$. Descrevemos a seguir cláusulas que garantem que $f$ é de fato um ladrilhamento válido $s \times s$, realizado por $\mathcal{D}$.

Para cada $m, n < s$, a cláusula

$$(x_{mnd_1} \vee x_{mnd_2} \vee \ldots \vee x_{mnd_k}),$$

garante que cada posição tem no mínimo um ladrilho. Para cada $m, n < s$ e cada dois ladrilhos distintos $d \neq d' \in D$, a cláusula $(\overline{x_{mnd}} \vee \overline{x_{mnd'}})$, garante que uma posição não pode ter mais que um ladrilho. As cláusulas descritas até aqui garantem que os $x_{mnd}$'s representam uma função $f$ de $\{0,\ldots, s-1\} \times \{0,\ldots, s-1\}$ para D.

A seguir, construiremos cláusulas que garantam que a função descrita pelos $x_{mnd}$'s é um ladrilhamento válido efetuado por $\mathcal{D}$: As cláusulas $(x_{i0f_0(i)})$ para $i = 0,\ldots, s-1$, forçam que $f(i, 0)$ seja $f_0(i)$. Para cada $n < s$ e $m < s-1$ e para cada $(d, d') \in D^2 - H$, tem-se a cláusula

$$(\overline{x_{mnd}} \vee \overline{x_{m+1,n,d'}})$$

que proíbe a vizinhança horizontal imediata de dois ladrilhos que não sejam horizontalmente compatíveis. Para garantir a compatibilidade vertical, para cada $n < s-1$ e $m < s$ e para cada $(d, d') \in D^2 - V$, tem-se a cláusula $(\overline{x_{mnd}} \vee \overline{x_{m,n+1,d'}})$. Isso completa a construção das cláusulas em $\tau(\mathcal{D}, s)$. Resta mostrar que $\tau(\mathcal{D}, s, f_0)$ é satisfazível se e somente se existe algum ladrilhamento $s \times s$ por $\mathcal{D}$ que estende $f_0$.

Suponha, então, que $\tau(\mathcal{D}, s, f_0)$ é satisfazível, por alguma atribuição, a que chamamos $T$. Como as grandes disjunções são satisfeitas por $T$, para cada $m$ e $n$ há pelo menos um $d \in D$, tal que $T(x_{mnd}) = \top$. Como as cláusulas $(\overline{x_{mnd}} \vee \overline{x_{mnd'}})$ são todas satisfeitas por $T$, não há $m$ e $n$ tais que dois ou mais $x_{mnd}$'s sejam $\top$ sob $T$. Portanto, $T$ representa uma função $f: \{0, \ldots, s-1\} \times \{0, \ldots, s-1\} \to D$.



Precisamos provar que $f(m, n)$ é um ladrilhamento válido $s \times s$ que estende $f_0$. Como as cláusulas $(x_{i0f_0(i)})$ são todas satisfeitas, então $f(i, 0) = f_0(i)$, conforme exigido. Então, o vínculo de adjacência horizontal deve ser satisfeito, porque, se não for satisfeito nas posições $(m, n)$ e $(m+1, n)$, então uma das cláusulas $(\overline{x_{mnd}} \vee \overline{x_{m+1,n,d'}})$ permanecerá insatisfeita. O mesmo ocorre para a adjacência vertical; concluímos que $f(m, n)$ é, de fato, um ladrilhamento válido $s \times s$ que estende $f_0$.

Inversamente, suponhamos que exista um ladrilhamento $s \times s$ estendendo $f_0$. Então, defina-se a seguinte atribuição $T$: $T(x_{mnd}) = \top$ se e somente se $f(m, n) = d$. É fácil verificar que $T$ satisfaz todas as cláusulas e assim a prova fica completa. ■

Portanto, não há algoritmo de resposta polinomial no tempo para o problema da SATISFATIBILIDADE, a não ser que $\mathcal{P} = \mathcal{NP}$. Como vimos na Seção 6.3, o caso especial de 2-SATISFATIBILIDADE pode ser resolvido em tempo polinomial. O teorema seguinte sugere que o próximo caso considerado seja intratável. Em analogia com 2-satisfatibilidade, a 3-SATISFATIBILIDADE é um caso particular de SATISFATIBILIDADE, em que todas as cláusulas envolvem três literais ou menos.

---

**Teorema 7.2.3:** O problema da 3-SATISFATIBILIDADE *é $\mathcal{NP}$-completo.*

---

***Prova:*** Ele naturalmente pertence à classe $\mathcal{NP}$, pois é um caso particular de outro problema que sabemos pertencer a $\mathcal{NP}$.

Para demonstrar a completude, devemos reduzir o problema da SATISFATIBILIDADE ao da 3-SATISFATIBILIDADE. Esse é um tipo comum de redução, em que um problema é reduzido ao seu próprio caso especial. Tais reduções procuram mostrar como, a partir de qualquer instância do problema geral, é possível eliminar os aspectos que impedem que essa instância recaia no caso particular. Na presente situação, mostra-se de que maneira, começando com qualquer conjunto $F$ de cláusulas, é possível chegar, em tempo polinomial, a um conjunto *equivalente* $\tau(F)$ de cláusulas com, no máximo, três literais em cada cláusula.

A redução é simples. Para cada cláusula em $F$, com $k > 3$ literais:

$$C = (\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_k),$$

Sejam $y_1, \ldots, y_{k-3}$ novas variáveis booleanas, que não figuram em nenhuma outra parte da fórmula booleana $\tau(F)$. Substituímos a cláusula $C$ pelo o seguinte novo conjunto de cláusulas:

$$(\lambda_1 \vee \lambda_2 \vee y_1), (\overline{y_1} \vee \lambda_3 \vee y_2), (\overline{y_2} \vee \lambda_4 \vee y_3), \ldots, (\overline{y_{k-4}} \vee \lambda_{k-2} \vee y_{k-3}), (\overline{y_{k-3}} \vee \lambda_{k-1} \vee \lambda_k).$$

Dividimos em cláusulas "curtas" todas as cláusulas "longas" de $F$ dessa forma, utilizando um conjunto diferente de variáveis $y_i$ em cada uma. O resultado da fórmula Booleana é $\tau(F)$, é fácil ver que $\tau$ pode ser executada em tempo polinomial.

Devemos provar que $\tau(F)$ é satisfazível se e somente se $F$ for satisfazível. A intuição consiste em interpretar a variável $y_i$ como sendo indicador de que "pelo menos um dos literais $\lambda_{i+2}, \ldots, \lambda_k$ seja verdadeiro", e a cláusula $(\overline{y_i} \vee \lambda_{i+2} \vee y_{i+1})$ como se indicasse que, "se $y_i$ é verdadeiro, então ou $\lambda_{i+2}$ é verdadeiro ou $y_{i+1}$ é verdadeiro".

Formalmente, suponhamos que a atribuição $T$ satisfaça $\tau(F)$. Devemos mostrar que $T$ também satisfaz cada cláusula de $F$. Isso é trivial para as cláusulas curtas; e, se $T$ mapeia todos os $k$ literais de uma cláusula original longa como $\perp$, então as $y_i$ variáveis não seriam capazes de, isoladamente, satisfazerem todas as cláusulas resultantes: a primeira cláusula forçaria $y_1$ a ser $\top$, a segunda, $y_2$ a ser $\top$, e, finalmente, a penúltima cláusula faria com que $y_{k-3}$ fosse $\top$, contradizendo a última cláusula (convém notar que esse é exatamente o *algoritmo de eliminação*, resolvendo essa instância de 2-SATISFATIBILIDADE).

Inversamente, se existir alguma atribuição $T$ que satisfaz $F$, então $T$ pode ser estendida para uma atribuição que satisfaz $\tau(F)$ da seguinte forma: para cada cláusula longa $C = (\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_k)$ de $F$, seja $j$ o menor índice para o qual $T(\lambda_j) = \top$ (como $T$ por hipótese satisfaz $F$, então tal $j$ existe). Então, definimos os valores das novas variáveis $y_1, \ldots, y_{k-3}$ como $T(y_i) = \top$ se $i \leq j-2$, e como $T(y_i) = \perp$ caso contrário. É fácil agora ver que $T$ satisfaz $\tau(F)$, e a prova de equivalência está completa. ■

Consideremos, finalmente, a seguinte versão de otimização do problema da SATISFATIBILIDADE:

MAX SAT: Dado um conjunto $F$ de cláusulas e um inteiro $K$, existe alguma atribuição que satisfaz pelo menos $K$ das cláusulas?

---

**Teorema 7.2.4:** MAX SAT *é um problema* $\mathcal{NP}$*-completo.*

---

***Prova:*** Que este problema pertence à classe $\mathcal{NP}$ é óbvio. Vamos então reduzir SATISFATIBILIDADE para MAX SAT.

Essa redução é extremamente simples, mas de um tipo que é muito comum e *muito* útil para determinar resultados de *completude* $\mathcal{NP}$ (ver diversos outros exemplos no Problema 7.3.4). Apenas observamos que MAX SAT é uma **generalização** de SATISFATIBILIDADE, ou seja, cada instância de SATISFATIBILIDADE é um tipo especial de instância de MAX SAT. E toda instância de SATISFATIBILIDADE pode ser pensada como uma instância de MAX SAT *para a qual o parâmetro $K$ é igual ao número de cláusulas.*

Formalmente, a redução deve ser a seguinte: dada uma instância $F$ de SATISFATIBILIDADE com $m$ cláusulas, construímos uma instância equivalente $(F, m)$ de MAX SAT simplesmente anexando a $F$ o parâmetro $K = m$, facilmente calculado. Obviamente, existe alguma atribuição que satisfaz pelo menos $K = m$ cláusulas de $F$ (que tem exatamente $m$ cláusulas) se e somente se existir uma atribuição que satisfaça todas as cláusulas de $F$. ■

Constata-se também que a restrição de MAX SAT a cláusulas com no máximo dois literais também é um problema $\mathcal{NP}$-completo (comparar com a 2-SATISFATIBILIDADE).

**Problemas para a Seção 7.2**

**7.2.1** Consideremos o que acontece na redução de $L$ para o LADRILHAMENTO LIMITADO quando $L$ pertence a classe $\mathcal{P}$ – ou seja, quando a máquina $M$ da prova do Teorema 7.2.1 é realmente determinística. Neste caso, o resultado do sistema de ladrilhamento pode ser expresso como um *fechamento* de um conjunto, referente a determinadas relações.

Seja $M$ uma máquina de Turing determinística que decide a linguagem $L$ em tempo polinomial $p(n)$, seja uma entrada de $M$ e $s$, $(D, H, V)$ e $f_0$ os componentes da instância do ladrilhamento limitado resultante da redução empregada na prova do Teorema 7.2.1, aplicados a $x$. Considere os conjuntos $P = \{0, 1, 2, \ldots, s-1\}$ e $S = P \times P \times D$. Seja $S_0 \subseteq S$ o seguinte conjunto:

$$\{(m, 0, f_0(m)): 0 \leq m < s\}.$$

Seja que $R_H \subseteq S \times S$ a seguinte relação:

$$R\{((m-1, n, d), (m, n, d')): 1 \leq m, n < s, (d, d') \in H\},$$

e seja ainda $R_V \subseteq S \times S$ dado por:

$$R\{((m, n-1, d), (m, n, d')): 0 \leq m < s, 2 \leq n < s, (d, d') \in V\}.$$

Mostrar que $x \in L$ se e somente se o fechamento de $S_0$ em relação a $R_H$ e $R_V$ contiver, para cada $0 \leq i, j < s$, uma tripla $(i, j, d)$ para algum $d \in D$. Em outras palavras, nessas condições não somente qualquer propriedade de fechamento poderá ser computada em tempo polinomial (isso foi mostrado ao final da seção 1.6), mas também, inversamente, *qualquer computação polinomial poderá ser expressa como uma propriedade de fechamento.* Naturalmente, a utilização de relações tão grandes como $R_H$ faz com que esse resultado pareça um pouco artificial, embora ele também se aplique a configurações muito mais significativas (ver as referências ao final do capítulo).

**7.2.2** Consideremos o LADRILHAMENTO BINÁRIO LIMITADO, uma versão do LADRILHAMENTO LIMITADO no qual não é fornecida a primeira linha de ladrilhos, mas somente o ladrilho da origem, $d_0$; o lado do quadrado a ser ladrilhado, $s$, é dado em binário.

(a) Mostre que existe uma redução da linguagem

$$E_1 = \{\text{"}M\text{"}: M \text{ pára em resposta à cadeia vazia em } 2^{|\text{"}M\text{"}|} \text{ passos}\}$$

para LADRILHAMENTO BINÁRIO LIMITADO.

(b) Prove que o problema do LADRILHAMENTO BINÁRIO LIMITADO não pertence à classe $\mathcal{P}$.

(c) Seja $\mathcal{NEXP}$ a classe de todas as linguagens decididas por *máquinas de Turing não-determinísticas* em tempo $2^{n^k}$ para algum $k > 0$. Mostre que (i) LADRILHAMENTO BINÁRIO LIMITADO está em $\mathcal{NEXP}$ e (ii) todas as linguagens em $\mathcal{NEXP}$ são polinomialmente redutíveis ao LADRILHAMENTO BINÁRIO LIMITADO, ou seja, LADRILHAMENTO BINÁRIO LIMITADO pode ser classificado como $\mathcal{NEXP}$*-completa.*

**7.2.3** (a) Mostrar que o problema da SATISFATIBILIDADE permanece $\mathcal{NP}$-completo mesmo que seja restrito a instâncias em que cada variável apareça, no máximo, três vezes. (*Sugestão*: substituir as ocorrências de variáveis $x$ por novas variáveis $x_1, \ldots, x_k$. Então, acrescentar uma fórmula em que cada uma dessas variáveis apareça duas vezes, concluindo assim que "todas essas variáveis seriam equivalentes".)

(b) O que aconteceria se cada variável aparecesse no máximo duas vezes?

**7.2.4** Usando como base o Problema 6.4.3, no qual se discute o fechamento da classe $\mathcal{NP}$ em relação a *homomorfismos não-apagáveis*, mostrar que $\mathcal{P}$ é fechada em relação aos homomorfismos não-apagáveis se e somente se $\mathcal{P} = \mathcal{NP}$. (*Sugestão*: uma das partes da prova decorre de (a). Para a outra parte, considere-se a seguinte linguagem, a qual pertence obviamente à classe $\mathcal{P}$:

$L = \{xy: x \in \{0, 1\}^*$ codifica uma fórmula booleana $F$, e $y \in \{\top, \bot\}^*$ é uma atribuição que satisfaz $F\}$.)

**7.2.5** Considere-se MAX 2-SAT, que é o seguinte caso especial de MAX SAT: Dado um conjunto $F$ de cláusulas, contendo, no máximo, dois literais cada, e um inteiro $K$, existe alguma atribuição que satisfaça pelo menos $K$ cláusulas?
Mostrar que MAX 2-SAT é um problema $\mathcal{NP}$-completo. (Essa prova não é simples. Considerem-se as cláusulas $(x)$, $(y)$, $(z)$, $(w)$, $(\bar{x} \vee \bar{y})$, $(\bar{y} \vee \bar{z})$, $(\bar{z} \vee \bar{x})$, $(x \vee \bar{w})$, $(y \vee \bar{w})$, $(z \vee \bar{w})$. Mostrar que esse conjunto de dez cláusulas tem a seguinte propriedade: todas as atribuições satisfatíveis em $x$, $y$, $z$ podem ser estendidas para satisfazerem sete cláusulas e não mais, exceto por uma, dessas atribuições, que pode somente ser estendida para satisfazer seis cláusulas. Seria possível utilizar esse "dispositivo" para reduzir 3-SATISFATIBILIDADE para MAX 2-SAT?)

## 7.3 OUTROS PROBLEMAS NP-COMPLETOS

Tendo provado um primeiro problema $\mathcal{NP}$-completo, é possível reduzi-lo a outro problema, e este a outros, concluindo serem todos eles $\mathcal{NP}$-completos. A Figura 7-4 ilustra uma descrição das reduções provadas nesta seção e na anterior. Os problemas propostos e as referências apresentam também muitos outros problemas $\mathcal{NP}$-completos.

Os problemas $\mathcal{NP}$-completos aparecem em todos os campos e aplicações em que sejam realizadas computações sofisticadas. É muito interessante ser capaz de identificar problemas $\mathcal{NP}$-completos – quer reconhecendo-os como problemas $\mathcal{NP}$-completos[†] já conhecidos, quer provando-os como sendo $\mathcal{NP}$-completos. Esse conhecimento poupa aos pesquisadores e programadores muitas tentativas infrutíferas de atingir metas impossíveis, e redireciona seu esforço para outras linhas de raciocínio mais promissoras (ver a Seção 7.4). Os problemas $\mathcal{NP}$-completos, tais como COLORAÇÃO EM GRAFOS (veja o Problema 7.1.1), SATISFATIBILIDADE e CONJUNTOS INDEPENDENTES são importantes, porque aparecem com freqüência e sob várias máscaras nos aplicativos. Outros, como o PROBLEMA DO CAIXEIRO-VIAJANTE, são importantes não somente por causa das suas aplicações práticas, mas também por terem

[†] Isso não é sempre fácil, porque problemas $\mathcal{NP}$-completos tendem a aparecer, na prática, de formas não muito claras.



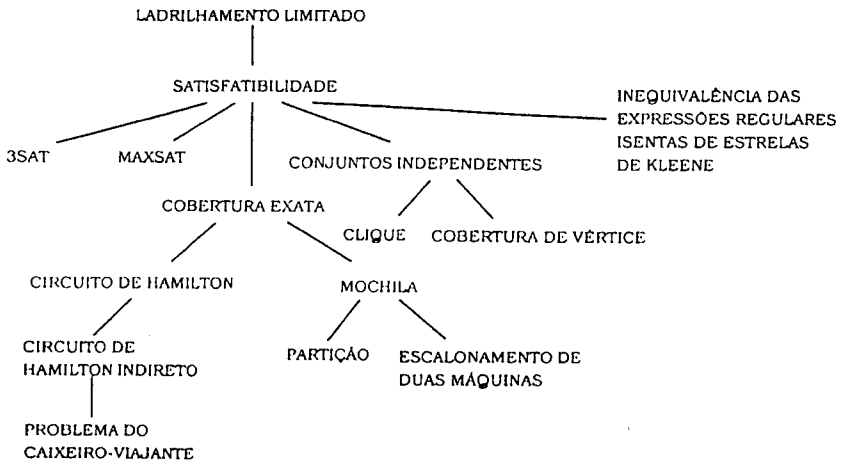



Figura 7-4

sido muito estudados. Ainda outros, tais como o problema apresentado a seguir, são importantes, pois muitas vezes são pontos de partida úteis para reduções de *completude* $\mathcal{NP}$ (SATISFATIBILIDADE tem uma grande importância por todos os três motivos).

Dados um conjunto finito $U = \{u_1, \ldots, u_n\}$ (o conjunto – *universo*) e a família de $m$ subconjuntos de $U$, $\mathcal{F} = \{S_1, \ldots, S_m\}$, pergunta-se se existiria uma *cobertura exata* $\mathcal{C} \subseteq \mathcal{F}$ tal que os conjuntos em $\mathcal{C}$ são disjuntos e sua união é $U$. Esse problema denomina-se COBERTURA EXATA.

Por exemplo, seja $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$ o universo, e seja uma família de subconjuntos

$\mathcal{F} = \{\{u_1, u_3\}, \{u_2, u_3, u_6\}, \{u_1, u_5\}, \{u_2, u_3, u_4\}, \{u_5, u_6\}, \{u_2, u_4\}\}$.

Uma cobertura exata existe nessa instância: $C = \{\{u_1, u_3\}, \{u_5, u_6\}, \{u_2, u_4\}\}$.

**Teorema 7.3.1:** O problema da COBERTURA EXATA é $\mathcal{NP}$-*completo*.

***Prova:*** É claro que COBERTURA EXATA é $\mathcal{NP}$: dada uma instância $(U, \mathcal{F})$ do problema, a subfamília que se deseja corresponde a um certificado válido. O certificado é polinomialmente conciso em relação ao tamanho da instância (uma vez que ele é um subconjunto de $\mathcal{F}$, o qual é por sua vez uma das cláusulas da entrada), podendo-se verificar, em tempo polinomial, se, de fato, todos os elementos de $U$ figuram, exatamente uma vez, nos conjuntos de $\mathcal{C}$.

Para provar a *completude* $\mathcal{NP}$ desse problema, devemos reduzir SATISFATIBILIDADE para o problema da COBERTURA EXATA: sendo dada uma fórmula Booleana $\mathcal{F}$ com cláusulas $\{C_1, \ldots, C_\ell\}$ sobre as variáveis Booleanas $x_1, \ldots, x_n$, devemos mostrar como construir, em tempo polinomial, uma instância equivalente $\tau(F)$ do problema da COBERTURA EXATA. Denotaremos os literais da cláusula $C_j$ como $\lambda_{jk}$, $k = 1, \ldots, m_j$, onde $m_j$ é o número de literais em $C_j$.

O universo de $\tau(F)$ é o conjunto

$$U = \{x_i: 1 \le i \le n\} \cup \{C_j: j = 1, \ldots, \ell\} \cup \{p_{jk}: 1 \le j \le \ell, k = 1, \ldots, m_j\}.$$

Assim, há um elemento para cada variável Booleana, um para cada cláusula e também um elemento para cada posição em cada cláusula.

Consideramos agora os conjuntos contidos em $\mathcal{F}$. Para cada elemento $p_{jk}$, temos em $\mathcal{F}$ um conjunto $\{p_{jk}\}$, e portando é trivial cobrir os $p_{jk}$'s, restando cobrir os elementos correspondentes às variáveis Booleanas e às cláusulas. Cada variável $x_i$ pertence a dois conjuntos em $\mathcal{F}$, a saber: o conjunto

$$T_{i,\top} = \{x_i\} \cup \{p_{jk}: \lambda_{jk} = \overline{x_i}\},$$

que também contém todas as ocorrências negadas de $x_i$, e o conjunto

$$T_{i,\bot} = \{x_i\} \cup \{p_{jk}: \lambda_{jk} = x_i\},$$

com as ocorrências positivas (note os símbolos invertidos). Por fim, cada cláusula $C_j$ pertence a $m_j$ conjuntos, um para cada literal nele presente, a saber $\{C_j, p_{jk}\}$, $k = 1, \ldots, m_j$. Estes são todos os conjuntos em $F$ e isso completa a descrição de $\tau(F)$.

Vamos ilustrar a redução para a fórmula Booleana $F$ dada, com cláusulas $C_1 = (x_1 \vee \overline{x_2})$, $C_2 = (\overline{x_1} \vee x_2 \vee x_3)$, $C_3 = (x_2)$ e $C_4 = (\overline{x_2} \vee \overline{x_3})$. O universo de $\tau(F)$ é

$$U = \{x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31}, p_{41}, p_{42}\},$$

e a família dos conjuntos $\mathcal{F}$ compreende os seguintes conjuntos:

$\{p_{11}\}, \{p_{12}\}, \{p_{21}\}, \{p_{22}\}, \{p_{23}\}, \{p_{31}\}, \{p_{41}\}, \{p_{42}\},$
$T_{1,\bot} = \{x_1, p_{11}\},$
$T_{1,\top} = \{x_1, p_{21}\},$
$T_{2,\bot} = \{x_2, p_{22}, p_{31}\},$
$T_{2,\top} = \{x_2, p_{12}, p_{41}\},$
$T_{3,\bot} = \{x_3, p_{23}\},$
$T_{3,\top} = \{x_2, p_{42}\},$
$\{C_1, p_{11}\}, \{C_1, p_{12}\}, \{C_2, p_{21}\}, \{C_2, p_{22}\},$
$\{C_2, p_{23}\}, \{C_3, p_{31}\}, \{C_4, p_{41}\}, \{C_4, p_{42}\},$

Vamos mostrar que $\tau(F)$ tem uma cobertura exata se e somente se $F$ é satisfazível. Suponhamos que exista uma cobertura exata $C$. Como cada $x_i$ deve ser coberto, exatamente um dos dois conjuntos $T_{i,\top}$ e $T_{i,\bot}$ contendo $x_i$ deve estar em $C$. Interpretamos que $T_{i,\top} \in \mathcal{C}$ significa que $T(x_i) = \top$ e que $T_{i,\bot} \in \mathcal{C}$ significa $T(x_i) = \bot$; isso define uma atribuição $T$, e mostramos também que $T$ satisfaz $F$. Para isso, consideremos uma cláusula $C_j$. O elemento de $U$ correspondente a essa cláusula deve ser coberto por um conjunto $\{C_j, p_{jk}\}$, para $1 \leq k \leq m_j$. Isso significa que o elemento $p_{jk}$ não está contido em qualquer outro conjunto participante da cobertura exata $\mathcal{C}$; em particular, não pertence ao conjunto em $\mathcal{C}$ que contém a variável que ocorre (negada ou não) no $k$-ésimo literal de $C_j$. Mas isso significa que $T$ torna o $k$-ésimo literal de $C_j$ $\top$ e, portanto, $C_j$ é satisfeito por $T$. Portanto, $F$ é satisfazível.

Inversamente, suponhamos que exista uma atribuição $T$ que satisfaz $F$. Então construímos a seguinte cobertura exata $\hat{\mathcal{C}}$: para cada $x_i$, $\mathcal{C}$ contém o conjunto $T_{i,\top}$ se $T(x_i) = \top$ e o conjunto $T_{i,\bot}$ se $T(x_i) = \bot$. Além disso, para cada cláusula $C_j$, $\mathcal{C}$ contém um conjunto $\{C_j, p_{jk}\}$ tal que o $k$-ésimo literal de $C_j$ é tornado $\top$ por $T$ e, portanto, $p_{jk}$ não está contida em qualquer conjunto



selecionado em $\mathcal{C}$ até aqui – sabemos que tal $k$ existe uma vez que, por hipótese, $T$ satisfaz $F$. Finalmente, tendo coberto todos os $x_i$'s e $C_j$'s, $\mathcal{C}$ inclui conjuntos unitários suficientes para cobrir quaisquer elementos $p_{jk}$ restantes que não tenham sido cobertos por outros conjuntos. No exemplo acima, a cobertura exata que corresponde à atribuição satisfatível $T(x_1) = \top$, $T(x_2) = \top$, $T(x_3) = \bot$ contém estes conjuntos: $T_{1,\top}$, $T_{2,\top}$, $T_{3,\bot}$, $\{C_1, p_{11}\}$, $\{C_2, p_{22}\}$, $\{C_3, p_{31}\}$, $\{C_4, p_{42}\}$, e também os conjuntos unitários $\{p_{12}\}$, $\{p_{21}\}$, $\{p_{23}\}$, $\{p_{41}\}$. Concluímos que $\tau(F)$ tem uma cobertura exata, completando a prova. ■

### O problema do caixeiro viajante

Podemos utilizar o problema de COBERTURA EXATA para estabelecer a *completude de $\mathcal{NP}$* do CIRCUITO DE HAMILTON.

**Teorema 7.3.2:** O problema do CIRCUITO DE HAMILTON *é $\mathcal{NP}$-completo.*

***Prova:*** Já sabemos que este problema é da classe $\mathcal{NP}$. Mostra-se a seguir como reduzir COBERTURA EXATA para CIRCUITO DE HAMILTON, apresentando para isso um algoritmo com tempo de resposta polinomial que, dada uma instância $(U, \mathcal{F})$ da cobertura exata, produz um grafo orientado $G = \tau(U, \mathcal{F})$ tal que $G$ tem um circuito de Hamilton se, e somente se, $(U, \mathcal{F})$ tem uma cobertura exata.

Essa construção é baseada em um grafo simples que tem interessantes propriedades similares as do problema do CIRCUITO DE HAMILTON – no jargão da completude $\mathcal{NP}$, tais grafos são chamados **dispositivos** (*gadgets*). A Figura 7-5(a) mostra esse dispositivo. Imagine que ele seja parte de um grafo maior $G$, conectado ao resto de $G$ por meio dos quatro vértices marcados como pontos sólidos. Em outras palavras, há outros vértices e arestas no grafo, mas nenhuma outra aresta além das mostradas é adjacente aos três vértices centrais. Além disso, suponha que $G$ possui um circuito de Hamilton, um circuito que percorre cada vértice de $G$ exatamente uma vez. Nessa situação, através de quais arestas o CIRCUITO DE HAMILTON irá percorrer os três vértices do meio? É fácil ver que existem de fato somente duas possibilidades: ou as arestas $(a, u)$, $(u, v)$, $(v, w)$, $(w, b)$ ou então as arestas $(c, w)$, $(w, v)$, $(v, u)$, $(u, d)$ são parte do circuito de Hamilton. Todas as outras possibilidades deixam alguns dos três vértices excluídos do circuito, o qual portanto, não seria de Hamilton.

Visto por outro ângulo, a estrutura desse simples dispositivo pode ser interpretada como *duas arestas* $(a, b)$ e $(c, d)$ com a seguinte reserva adicional: em qualquer circuito de Hamilton do grafo global $G$ ou $(a, b)$ *é percorrido ou* $(c, d)$ *o é, mas não ambos*. Essa situação pode ser visualizada na Figura 7-5(b), onde as duas arestas, que em geral seriam independentes, estão conectadas por um sinal de *ou exclusivo*, significando que uma e uma só delas será percorrida por qualquer circuito de Hamilton. Sempre que essa construção for utilizada em uma descrição de grafo, isso indicará que de fato o grafo contém o subgrafo completo mostrado na Figura 7- 5(a). Aliás, é possível que sejam usadas várias arestas, conectadas por tais subgrafos à mesma aresta (ver Figura 7-5(c)). O resultado é o mesmo: todas as arestas de um lado ou de outro são percorridas, mas não de ambos.

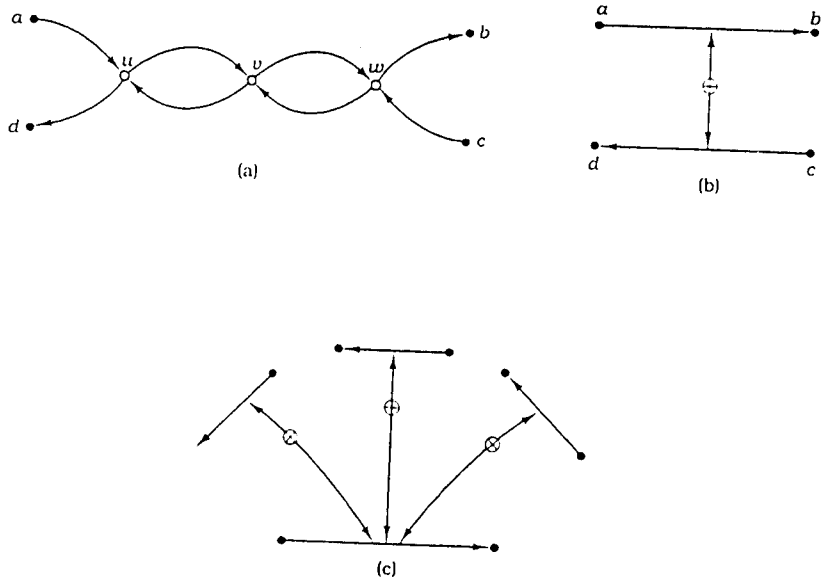



Figura 7-5

Esse dispositivo constitui uma base para a construção do nosso grafo $G = \tau(U, \mathcal{F})$ correspondente à instância de COBERTURA EXATA com $U = \{u_1, \ldots, u_n\}$ e $\mathcal{F} = \{S_1, \ldots, S_m\}$. Descreve-se o grafo $G$ a seguir. Há vértices $u_0, u_1, \ldots, u_n$ e $S_0, S_1, \ldots, S_m$, isto é, um para cada elemento do universo e um para cada conjunto na instância dada, mais dois vértices. Para $i = 1, \ldots, m$, há *duas* arestas $(S_{i-1}, S_i)$. Naturalmente, em grafos não faz sentido haver duas arestas diferentes conectando o mesmo par de vértices. A única razão por que permitimos isso no presente caso é que as arestas serão mais tarde unidas por subgrafos de "ou exclusivo", como na Figura 7-4 e, portanto, não haverá arestas "paralelas" ao final da construção. Uma das duas arestas $(S_{i-1}, S_i)$ é chamada aresta *longa* e, a outra, aresta *curta*. Para $j = 1, \ldots, n$, entre os vértices $u_{j-1}$ e $u_j$, há *tantas arestas quanto conjuntos em $\mathcal{F}$ contendo o elemento $u_j$*. Desse modo, pode-se considerar que cada cópia da aresta $(u_{j-1}, u_j)$ corresponde a uma ocorrência de $u_j$ em um conjunto. Finalmente, adicionam-se as arestas $(u_n, S_0)$ e $(S_m, u_0)$ fechando, portanto, o circuito.

Note que a construção até aqui depende apenas do tamanho do universo e do número e tamanho dos conjuntos; ela não depende da informação precisa de qual conjunto contém cada elemento. Essa propriedade estrutural da instância irá influenciar nossa construção da seguinte maneira: como cada cópia da aresta $(u_{j-1}, u_j)$ corresponde a uma ocorrência do elemento $u_j$ em algum conjunto $S_i \in \mathcal{F}$, tal que $u_j \in S_i$, *unimos por um subgrafo de "ou exclusivo" essa cópia da aresta $(u_{j-1}, u_j)$ com a aresta longa $(S_{i-1}, S_i)$* (ver um exemplo na Figura 7-6). Isso completa a construção do grafo $\tau(U, \mathcal{F})$.



$S_1 = \{u_3, u_4\}$

$S_2 = \{u_2, u_3, u_4\}$

$S_3 = \{u_1, u_2\}$

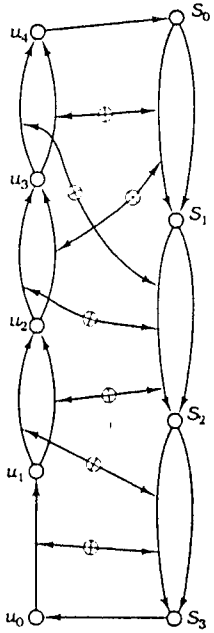



Figura 7-6

Vamos provar que o grafo $\tau(U, \mathcal{F})$ apresenta um circuito de Hamilton se e somente se $\tau(U, \mathcal{F})$ possui uma cobertura exata. Suponhamos que existe um circuito de Hamilton. Ele deve percorrer os vértices correspondentes aos conjuntos, na ordem $S_0, S_1, \ldots, S_m$, depois percorrer a aresta $(S_m, u_0)$, então, os vértices $u_0, u_1, \ldots, u_n$ e por fim terminar na aresta $(u_n, S_0)$ (ver a Figura 7-6). Pergunta-se se ele irá percorrer para cada conjunto $S_j$, as arestas curtas ou as longas $(S_{j-1}, S_j)$. Seja $\mathcal{C}$ o conjunto de todos os conjuntos $T_j$ tal que as arestas curtas $(S_{j-1}, S_j)$ sejam percorridas pelo circuito de Hamilton. Devemos mostrar que $\mathcal{C}$ é um legítimo conjunto de cobertura, isto é, que ele contém todos os elementos, sem repetições. Mas isso não é difícil de verificar: pela propriedade do subgrafo de "ou exclusivo", os elementos contidos nos conjuntos em $\mathcal{C}$ são precisamente as cópias das arestas da forma $(u_{i-1}, u_i)$ que são percorridas pelo circuito de Hamilton; e este percorre exatamente tal aresta para cada elemento $u_j \in U$. Segue-se que cada $u_j \in U$ está contido em exatamente um dos conjuntos pertencente a $\mathcal{C}$, logo $\mathcal{C}$ é uma cobertura exata.

Inversamente, suponhamos que uma *cobertura exata $C$* exista. Então um circuito de Hamilton no grafo $\tau(U, \mathcal{F})$ pode ser construído como segue: percorrem-se as cópias curtas de todas as arestas $(S_{j-1}, S_j)$ em que $S_j \in \mathcal{C}$, e as arestas longas para todos os outros conjuntos. Então, para cada elemento $u_i$, percorre-se a cópia da aresta $(u_{i-1}, u_i)$ que corresponde ao único conjunto em $\mathcal{C}$ que contém $u_i$. Completa-se o circuito de Hamilton com as arestas $(u_n, S_0)$ e $(S_m, u_0)$. ■

Uma vez mostrado que um problema é $\mathcal{NP}$-completo, a pesquisa muitas vezes concentra-se em resolver casos especiais, interessantes e tratáveis do mesmo. As provas da *completude* $\mathcal{NP}$, muitas vezes, produzem instâncias do problema-alvo que são complexas e "artificiais". Resta saber se as instâncias *de interesse prático*, sendo elas muito menos complexas, poderiam continuar insolúveis por qualquer algoritmo eficiente. Alternativamente, pode-se, muitas vezes, mostrar que até versões substancialmente restritas do problema continuam sendo $\mathcal{NP}$-completas. Já vimos ambas as situações ao estudar o caso de SATISFATIBILIDADE, cujo caso especial de 2-SATISFATIBILIDADE pode ser resolvido em tempo polinomial, mas cujo caso especial 3-SATISFATIBILIDADE é $\mathcal{NP}$-completo. Para apresentar um interessante caso especial do circuito de Hamilton, define-se o circuito de Hamilton não orientado como sendo o problema de circuito de Hamilton restrito a grafos não orientados, isto é, simétricos e sem laços.

**Teorema 7.3.3:** *O problema do* CIRCUITO DE HAMILTON NÃO-ORIENTADO *é $\mathcal{NP}$-completo.*

***Prova:*** Deve-se reduzir o problema normal do CIRCUITO DE HAMILTON a este. Dado um grafo $G \subseteq V \times V$, deve-se construir um grafo *simétrico* $G' \subseteq V' \times V'$, sem laços, tal que $G$ apresente um circuito de Hamilton se e somente se $G'$ possuir um. A construção, ilustrada na Figura 7-7, é a seguinte: seja, $V' = \{v_0, v_1, v_2: v \in V\}$, isto é, $G'$ apresenta três vértices $v_0$, $v_1$ e $v_2$ para cada vértice $v$ de $G$. Destes, $v_0$ é, informalmente, o vértice de *entrada*, para o qual as arestas que entram em $v$ são dirigidas e $v_2$ é o vértice de *saída*, a partir do qual as arestas que partem de $v$ irão emanar.

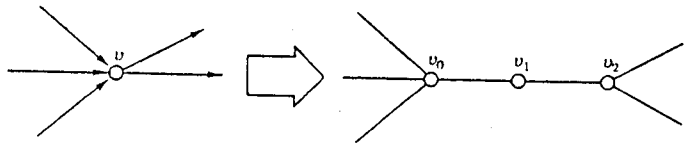



Figura 7-7

Assim, as arestas em $G'$ serão as seguintes (ver Figura 7-7; lembrar que grafos não orientados são melhor representados por linhas sem orientação conectando os vértices):

$$\{(u_2, v_0), (v_0, u_2): (u, v) \in G\} \cup \{(v_0, v_1), (v_1, v_0), (v_1, v_2), (v_2, v_1): v \in V\},$$

ou seja, os vértices $v_0$, $v_1$, $v_2$ são conectados por um caminho nessa ordem, e há uma aresta não orientada entre $u_2$ e $v_0$, sempre que $(u, v) \in G$. Isso completa a construção de $G'$.

Deve-se agora provar que $G'$ apresenta um circuito de Hamilton se, e somente se, $G$ possuir algum. Suponha-se que um circuito de Hamilton de $G'$ chega no vértice $v_0$ através uma aresta da forma $(u_2, v_0)$. Se o circuito de Hamilton deixa $v_0$ através de uma aresta outra que não $(v_0, v_1)$, então, ele não poderá atingir o vértice $v_1$ de qualquer outro modo e, portanto, ele foi (erroneamente) considerado como circuito de Hamilton. Portanto, a aresta $(v_0, v_1)$ deve ser uma parte do circuito, bem como $(v_1, v_2)$. Então, o circuito deve prosseguir por uma das arestas $(v_2, w_0)$, onde $(v, w) \in G$, daí para $w_1$, $w_2$, para algum $z_0$ onde $(w, z) \in G$ e assim por diante. Assim, as arestas da forma $(v_0, v_2)$ no circuito de Hamilton de $G'$ constituem, de fato, um circuito de Hamilton de $G$. Inversamente, qualquer circuito de Hamilton $(v^1, v^2, \ldots, v^{|V|})$ de $G$ pode ser convertido em um circuito de Hamilton de $G'$ como segue: $(v_0^1, v_1^1, v_2^1, v_0^2, v_1^2, v_2^2, \ldots, v_0^{|V|}, v_1^{|V|}, v_2^{|V|})$. Concluímos que $G$ tem um circuito de Hamilton se e somente se $G'$ tem um circuito de Hamilton, e a prova está completa. ■



Nosso próximo resultado diz respeito ao notório PROBLEMA DO CAIXEIRO VIAJANTE – sua versão "sim–não", na qual cada instância é fornecida com um orçamento $B$, como definido na Seção 6.2.

**Teorema 7.3.4:** *O* PROBLEMA DO CAIXEIRO-VIAJANTE *é $\mathcal{NP}$-completo.*

***Prova:*** Já sabemos que este problema está em $\mathcal{NP}$. Para mostrar a completude, vamos reduzir o CIRCUITO DE HAMILTON NÃO ORIENTADO AO PROBLEMA DO CAIXEIRO-VIAJANTE. Dado um grafo simétrico $G$, onde, sem perda de generalidade, $V = \{v_1, \ldots, v_{|V|}\}$, construiremos a seguinte instância do PROBLEMA DO CAIXEIRO-VIAJANTE: $n$, o número de cidades, é $|V|$, e a distância $d_{ij}$ entre quaisquer duas cidades $i$ e $j$ é

$$d_{ij} = \begin{cases} 0 \text{ se } i = j; \\ 1 \text{ se } (v_i, v_j) \in G; \\ 2 \text{ caso contrário.} \end{cases}$$

Como $G$ é um grafo simétrico sem laços, essa função de distância é, ela própria, simétrica, o que significa que $d_{ij} = d_{ji}$ para todas as cidades $i$ e $j$, como exigido. Por fim, o orçamento é $B = n$.

Obviamente, qualquer "roteiro" das cidades tem custo igual ao número $n$ acrescido do total das distâncias entre as cidades percorridas que não estão nas arestas de $G$. Portanto, uma viagem de custo $B$ ou menor existirá se e somente se o número de "não arestas" utilizado é zero, isto é, se e somente se a viagem constitui um circuito de Hamilton de $G$. ■

## Partições e Cliques

Uma COBERTURA EXATA também provê uma interessante prova da *completude* $\mathcal{NP}$ do problema da PARTIÇÃO. Parte-se do intimamente relacionado problema da MOCHILA, na qual é fornecida como alvo uma soma arbitrária $K$, a ser obtida (lembre-se de sua definição no Exemplo 7.1.2):

**Teorema 7.3.5:** *O* PROBLEMA DA MOCHILA *é $\mathcal{NP}$-completo.*

***Prova:*** Sabe-se que o problema da MOCHILA pertence à classe $\mathcal{NP}$: dada uma instância de MOCHILA, $a_1, \ldots, a_n$ e $K$, um subconjunto $P$ de $\{1, \ldots, n\}$, tal que $\sum_{i \in P} a_i = K$ pode servir como um certificado de que a resposta à

instância fornecida é "sim". Ele é polinomialmente sucinto, podendo ser confirmado em tempo polinomial, por adição binária.

Deve-se a seguir efetuar a redução da COBERTURA EXATA a MOCHILA. É dado um universo $U = \{u_1, \ldots, u_n\}$ e uma família $\mathcal{F} = \{S_1, \ldots, S_m\}$ de subconjuntos de $U$. Deve-se construir uma instância $r(U, \mathcal{F})$ de MOCHILA, isto é, uma seqüencia de inteiros não-negativos $a_1, \ldots, a_k$ e outro inteiro $K$, tais que haja um subconjunto $P \subseteq \{1, \ldots, k\}$ com $\Sigma_{i \in P} a_i = K$ se e somente se exista um conjunto de conjuntos $\mathcal{C} \subseteq \mathcal{F}$ que sejam disjuntos e que coletivamente cubram todos os elementos de $U$.

Essa construção é particularmente simples, porque se baseia em uma proximidade inesperada entre as operações de *união de conjuntos* e de *adição de inteiros*. Subconjuntos de um conjunto de $n$ elementos, tais como os contidos em $\mathcal{F}$, podem ser representados como cadeias sobre $\{0, 1\}^n$ (ver a Figura 7-8). Tais cadeias podem ser, por sua vez, interpretadas como inteiros compreendidos entre zero e $2^n - 1$, escritos em binário. A união de tais conjuntos, desde que eles sejam disjuntos, pode ser visto como sendo o mesmo que a adição dos inteiros correspondentes. Como na COBERTURA EXATA estamos perguntando se a união das partes disjuntas constituem realmente todo o universo $U$, isso parece ser o mesmo que perguntar se há uma seleção de inteiros entre os dados, cuja soma seja $K = 1 + 2 + 4 + \ldots + 2^{n-1}$ – os números binários representados por uma cadeia com $n$ 1's. E é muito próximo de uma instância de MOCHILA.

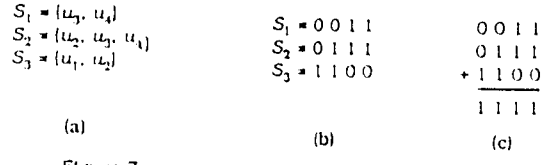



Figura 7-8: A partir dos conjuntos (a) para vetores de bit (b) para adição de inteiros na base $m$ (c)

Há um problema com essa redução simples: é rompida a correspondência de fechamento entre a união de conjuntos e adição de inteiros, porque na adição de inteiros pode ser necessário o "vai-um". Considere-se, por exemplo, a soma 11 + 13 + 15 + 24 = 63; em binário 001011 + 001101 + 001111 + 011000 = 111111. Se a traduzirmos de volta para a forma de subconjuntos de $\{u_1, \ldots, u_6\}$, os conjuntos $\{u_3, u_5, u_6\}$, $\{u_3, u_4, u_5\}$, $\{u_3, u_4, u_5, u_6\}$ e $\{u_2, u_3\}$ não são mais disjuntos, nem cobrem todos os elementos de $U$. Em outras palavras, a operação de "vai-um" torna falha a tradução entre as operações de união e adição.

Esse problema pode ser muito facilmente resolvido da seguinte maneira: em vez de considerar as cadeias em $\{0, 1\}^n$ como inteiros em binário, consideremo-los como inteiros *em m-ário*, onde $m$ é o número de conjuntos em $\mathcal{F}$. Dessa forma, teremos $M$ inteiros $a_1, \ldots, a_m$, onde $a_i = \sum_{u_j \in S_i} m^{j-1}$. Perguntamos se há um subconjunto que totalize $K = \sum_{j=1}^{n} m^{j-1}$. Esse modo de proceder não é problemático, uma vez que a adição de menos de $m$ dígitos em $m$-ário, sendo cada um dos dígitos apenas 0 ou 1, nunca pode levar a um "vai-um". Conclui-se, portanto, que a instância resultante de MOCHILA terá uma solução se e somente se a instância original de COBERTURA EXATA tiver uma solução. ■



---

**Corolário:** *Os problemas da* PARTIÇÃO E DO ESCALONAMENTO DE DUAS MÁQUINAS *são $\mathcal{NP}$-completos.*

---

*Prova:* Há reduções polinomiais de MOCHILA para ambos esses problemas (ver o Exemplo 7.1.2). ■

Concentrar-nos-emos a seguir em três problemas da teoria dos grafos, apresentados na Seção 6.2: CONJUNTOS INDEPENDENTES, CLIQUE E COBERTURA DE VÉRTICES.

---

**Teorema 7.3.6:** *O problema dos* CONJUNTOS INDEPENDENTES *é $\mathcal{NP}$-completo.*

---

*Prova:* Este problema está claramente na classe $\mathcal{NP}$; pretende-se reduzir a ele o problema da 3-SATISFATIBILIDADE.

Seja dada uma fórmula Booleana $\mathcal{F}$ com cláusulas $C_1, \ldots, C_m$, contendo cada uma, no máximo, três literais. De fato, devemos assumir que todos as cláusulas de $\mathcal{F}$ têm *exatamente* três literais; se uma cláusula tiver somente um ou dois literais, então permitimos que um literal seja repetido, a fim de elevar o número total para três. Vamos construir um grafo não-orientado $G$ e um inteiro $K$, tal que exista um conjunto de $K$ vértices em $G$ sem arestas entre elas se e somente se $\mathcal{F}$ for satisfazível.

Esta redução é ilustrada na Figura 7-9: para cada uma das cláusulas $C_1, \ldots, C_m$, de $\mathcal{F}$, tem-se três vértices em $G$, conectados por arestas, de maneira que formem um triângulo – chamados vértices dos triângulos correspondentes à cláusula $C_i$: $c_{i1}$, $c_{i2}$, $c_{i3}$. Estes são todos os vértices de $G$ – um total de $3m$ vértices. A meta é $K = m$, igual ao número de cláusulas. Para definir as arestas restantes de $G$, o vértice $c_{ij}$ é identificado com o $j$-ésimo literal da cláusula $C_i$. Por fim, dois vértices são unidos por uma aresta *se e somente se seus literais forem a negação um do outro*. Isso completa a descrição desta redução; ver a Figura 7-9 como um exemplo.

Suponhamos que exista um conjunto independente $I$ em $G$, com $K = m$ vértices. Como quaisquer dois vértices do mesmo triângulo estão conectados por uma aresta, evidentemente há exatamente um vértice em $I$ para cada triângulo. Convém lembrar que os vértices correspondem aos literais. Considere-se agora que o fato de que um vértice estar em $I$ significa que o literal correspondente é T. Como não há arestas ligando dois vértices em $I$, segue-se que não existem dois desses literais que sejam a negação um do outro e, portanto, eles podem ser a base de uma atribuição $T$. Note-se que $T$ pode não ser completamente definido em todas as variáveis, porque o conjunto de vértices em $I$ pode não envolver todas as variáveis; por exemplo, na Figura 7-9, o conjunto independente, indicado pelos círculos cheios, não determina o valor da variável $x_3$. $T$ pode assumir qualquer valor verdade em tais variáveis "ausentes"; o resultado da atribuição $T$ satisfaz todas as cláusulas, porque cada cláusula tem pelo menos um literal satisfeito por $T$. In-

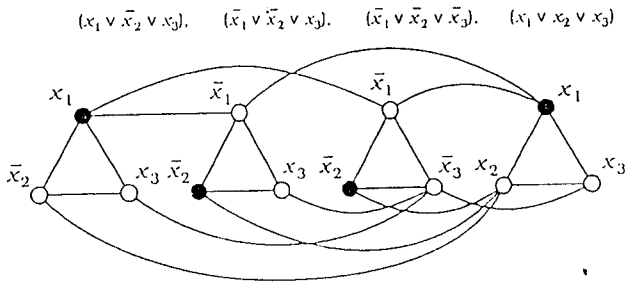



Figura 7-9

versamente, dada qualquer atribuição que satisfaça $\mathcal{F}$, pode-se obter um conjunto independente de tamanho $m$ escolhendo para cada cláusula um vértice correspondente a um literal satisfeito. ■

*A completude $\mathcal{NP}$* de dois outros problemas em teoria dos grafos é agora imediata:

**Teorema 7.3.7:** *Os* PROBLEMAS DA CLIQUE E DA COBERTURA *são $\mathcal{NP}$-completos.*

***Prova:*** Ambos os problemas estão claramente na classe $\mathcal{NP}$.

O problema da clique, ao exigir que *todas as* arestas entre quaisquer dois vértices do conjunto estejam presentes, é, de alguma forma, o *oposto exato* do problema dos CONJUNTOS INDEPENDENTES. A redução torna precisa esta relação: dada uma instância $(G, K)$ de CONJUNTOS INDEPENDENTES, na qual $G \subseteq V \times V$ é um grafo não-orientado e $K \geq 2$ é a meta, cria-se uma instância equivalente $(G', K')$ de clique simplesmente considerando $G' = V \times V - \{(i, i): i \in V\} - G$, e mantendo a mesma meta, $K' = K$. Isso funciona porque, como é relativamente fácil verificar, o conjunto independente máximo de $G$ é precisamente a clique máxima no *complemento* de $G$, o grafo que contém todas as arestas não-laços que não estão em $G$ (ver a Figura 7-10).

Por fim, COBERTURA DE VÉRTICES é o oposto exato de CONJUNTOS INDEPENDENTES, mas em um sentido diferente: uma vez que os vértices em uma COBERTURA DE VÉRTICE $N \subseteq V$ estão ligados a em todas as arestas[1], o conjunto $V - N$ não deve ter arestas entre seus elementos e é, portanto, um conjunto independente (ver a Figura 7-11). Portanto, $N \subseteq V$ é uma COBERTURA DE VÉRTICES de $G$ se e somente se $V - N$ é um conjunto independente de $G$. Portanto, o conjunto independente máximo de $G$ tem tamanho $K$ ou mais se, e somente se, a COBERTURA DE VÉRTICES mínima de $G$ tem tamanho $|V| - K$ ou menos. A redução de CONJUNTOS INDEPENDENTES para COBERTURA DE VÉRTICES mantém o grafo inalterado, e simplesmente substitui $K$ por $|V| - K$. ■

[1] N. de R. No sentido de que cada aresta do grafo tem ao menos um vértice pertencente a este conjunto de cobertura.



Figura 7-10

Figura 7-11

## Autômatos finitos

Nosso último resultado acerca da *completude $\mathcal{NP}$* diz respeito a alguns dos primeiros e aparentemente mais simples objetos matemáticos estudados neste livro: os autômatos finitos não-determinísticos e as expressões regulares.

Entre todos os problemas apresentados no último capítulo, há somente dois cujo caráter de membro da classe $\mathcal{NP}$ não é tão óbvio: a EQUIVALÊNCIA DE EXPRESSÕES REGULARES e o problema a ele muito fortemente relacionado, da EQUIVALÊNCIA DE AUTÔMATOS FINITOS NÃO-DETERMINÍSTICOS. Dadas duas expressões regulares, o que seria um certificado convincente de sua equivalência? Nada sucinto vem à mente.

Se, entretanto, definimos o *problema complementar*

INEQUIVALÊNCIA DE EXPRESSÕES REGULARES: dadas duas expressões regulares $R_1$ e $R_2$, é $L(R_1) \neq L(R_2)$?

então, os certificados parecem viabilizar-se: um certificado da inequivalência de duas expressões regulares é uma cadeia pertencente à linguagem gerada por um, mas não pelo outro, isto é, qualquer elemento de $(L(R_1) - L(R_2)) \cup (L(R_2) - L(R_1))$. De fato, esse conjunto será não-vazio se, e somente se, as expressões forem inequivalentes.

Mas, agora, surge a real dificuldade: esses certificados são legítimos em todos os aspectos, exceto quanto à essencial propriedade de *concisão polinomial* (*polynomial succinctness*). Dadas duas expressões regulares $R_1$ e $R_2$, não há um limite superior óbvio que seja polinomial em relação ao comprimento da cadeia mais curta que pertence a $(L(R_1) - L(R_2)) \cup (L(R_2) - L(R_1))$. Para que isso possa ocorrer, ele deve ser polinomial em relação ao compri-

mento total das duas expressões, $|R_1| + |R_2|$ – isto é, ao número total de símbolos, tais como $a$, $b$, $\cup$, $^*$ e parênteses, necessários para representá-las. De fato, há famílias de pares de expressões regulares inequivalentes que diferem somente em cadeias que são exponencialmente longas em relação ao tamanho das expressões!

Para obter um problema em $\mathcal{NP}$, é preciso observar um caso restrito especial: **expressões regulares isentas de ***, que são expressões regulares sobre união e concatenação, não contendo quaisquer ocorrências de estrela de Kleene. Considere uma expressão regular isenta de $^*$, tal como

$$R = (0 \cup 1)00(0 \cup 1) \cup 010(0 \cup 1)0.$$

Aqui é fácil verificar que, se $x$ é uma cadeia da linguagem gerada pela expressão (por exemplo, $x = 1001$ para a expressão $R$ acima), então $|x| \leq |R|$. Como resultado dessa observação, o seguinte problema está em $\mathcal{NP}$:

> INEQUIVALÊNCIA DE EXPRESSÕES REGULARES ISENTAS DE $^*$: dadas duas expressões regulares isentas de $^*$ $R_1$ e $R_2$, é $L(R_1) \neq L(R_2)$?

Um certificado válido pode ser qualquer cadeia em $(L(R_1) - L(R_2)) \cup (L(R_2) - L(R_1))$, *desde que* todas as tais cadeias sejam sucintas, mais curtas do que $|R_1| + |R_2|$. Para um problema análogo no domínio de AUTÔMATOS FINITOS NÃO-DETERMINÍSTICOS, veja o Problema 7.3.7.

De fato, podemos provar esse resultado:

---

**Teorema 7.3.8:** *A* INEQUIVALÊNCIA DE EXPRESSÕES REGULARES ISENTAS DE $^*$ *é* $\mathcal{NP}$*-completa.*

---

***Prova:*** Já argumentamos que este problema pertence à classe $\mathcal{NP}$. Vamos mostrar que o problema da SATISFATIBILIDADE pode ser reduzido para o da INEQUIVALÊNCIA DE EXPRESSÕES REGULARES ISENTAS DE $^*$.

Dada uma fórmula booleana arbitrária com as variáveis booleanas $x_1, \ldots, x_n$ e as cláusulas $C_1, \ldots, C_m$, devemos produzir duas expressões regulares $R_1$ e $R_2$ sobre o alfabeto $\Sigma = \{0, 1\}$, tais que em nenhuma delas ocorre uma estrela de Kleene, e que $L(R_1) \neq L(R_2)$ se e somente se a fórmula booleana em questão é satisfazível.

A segunda expressão regular, $R_2$, é muito simples:

$$(0 \cup 1)(0 \cup 1) \ldots (0 \cup 1),$$

com a expressão $(0 \cup 1)$ repetida $n$ vezes. A linguagem gerada por $R_2$ é obviamente o conjunto de todas as cadeias binárias de comprimento $n$, o que significa dizer, $L(R_2) = \{0, 1\}^n$.

Pode-se então passar à construção de $R_1$. Ao contrário de $R_2$, $R_1$ depende fortemente da fórmula booleana fornecida. Em particular, $R_1$ é a união de $m$ expressões regulares

$$R_1 = \alpha_1 \cup \alpha_2 \cup \ldots \cup \alpha_m,$$



onde a expressão regular $\alpha_i$ depende da cláusula $C_i$. Cada $\alpha_i$ é a concatenação de $n$ expressões regulares:

$$\alpha_i = \alpha_{i1}\, \alpha_{i2} \ldots \alpha_{in},$$

onde

$$\alpha_{ij} = \begin{cases} 0, & \text{se } x_j \text{ é um literal de } C_i; \\ 1, & \text{se } \overline{x_j} \text{ é um literal de } C_i; \\ (0 \cup 1), & \text{caso contrário.} \end{cases}$$

Se desprezarmos por um momento a distinção entre 0 e 1 e entre $\top$ e $\bot$, então cadeias em $\{0, 1\}^n$ podem ser pensadas como atribuições de valores verdade para as variáveis booleanas $\{x_1, \ldots, x_n\}$. Nessa interpretação, $L(\alpha_i)$ é precisamente o conjunto de todas as atribuições de valores verdade que *não conseguem* satisfazer $C_i$. Portanto, $L(R_1)$ é o conjunto de todas as atribuições que deixam de satisfazer pelo menos uma das cláusulas da fórmula booleana fornecida. Portanto, a fórmula Booleana fornecida é satisfeita se e somente se $L(R_1)$ for diferente de $\{0, 1\}^n$ – que é precisamente $L(R_1)$, completando a prova. ■

O problema de equivalência, no caso de expressões regulares gerais e dos autômatos finitos não-determinísticos pode, naturalmente, ser somente mais difícil (consulte as referências para obter informações sobre sua complexidade precisa), o mesmo ocorrendo com os problemas de minimização de estados para autômatos finitos não-determinísticos. Nenhum desses problemas, mesmo quando formulados sobre os mais primitivos modelos computacionais, pode ser resolvido eficientemente a não ser que $\mathcal{P} = \mathcal{NP}$.

Mas, considerando o seguinte caso mais ambicioso: deseja-se, dado um autômato finito não-determinístico, determinar o equivalente *determinístico*, com um número mínimo de estados. Sabemos que qualquer algoritmo desse tipo deve ser exponencial no pior caso porque a saída pode ter de ser exponencialmente longa em relação ao comprimento de entrada (lembrar do Exemplo 2.5.4). Existiria algum algoritmo que possa ser executado em tempo polinomial, em relação tanto ao tamanho da entrada quanto ao *da saída*? Tal algoritmo gastaria um tempo exponencial quando a saída fosse grande, mas rapidamente forneceria a saída para pequenos autômatos. (O algoritmo óbvio, que implementa inicialmente a construção proposta e em seguida minimiza o autômato determinístico resultante não será satisfatório, porque a construção do subconjunto pode produzir um resultado intermediário exponencialmente grande, mesmo que, ao final, a saída – o autômato determinístico mínimo equivalente – possa ser polinomial).

Infelizmente, é possível mostrar que mesmo a existência de tal algoritmo é improvável:

---

***Corolário:*** *A não ser que $\mathcal{P} = \mathcal{NP}$, não há algoritmo tal que, dada uma expressão regular ou um autômato finito não-determinístico, seja capaz de construir o autômato finito determinístico equivalente com o número mínimo de estados, em um tempo que seja polinomial em relação à entrada e à saída.*

---

*Prova:* Suponhamos que $M_n$ denote o autômato finito simples de $n+1$ estados que aceita $\{0, 1\}^n$. Na redução utilizada na prova do teorema, a fórmula booleana fornecida, com $n$ variáveis, é insatisfatível se e somente se o autômato finito determinístico, com número mínimo de estados, equivalente a $R_1$, é exatamente $M_n$.

Suponhamos que exista um algoritmo como o descrito no enunciado do corolário, que apresente um tempo limite na forma $p(|x| + |y|)$, onde $p$ é um polinômio, $x$ é a entrada do algoritmo e $y$ é sua saída. Então, podemos resolver a SATISFATIBILIDADE do seguinte modo:

Dada qualquer fórmula booleana $F$ com $n$ variáveis, primeiro realizamos a redução, descrita na prova do teorema, para obter uma expressão regular $R_1$. Depois executamos, sobre a entrada $R_1$, o algoritmo pretendido por $p(|R_1| + |M_n|)$ passos, onde $|M_n|$ é o comprimento da codificação de $M_n$. Se o algoritmo terminar dentro do tempo alocado, respondemos "não" à questão original de SATISFATIBILIDADE caso a saída seja $M_n$, e respondemos "sim" no caso de qualquer outra saída. Se, entretanto, o algoritmo não parar após $p(|R_1| + |M_n|)$ passos, e como sabemos que ele sempre pára após $p(|x| + |y|)$ passos, podemos concluir que a saída desse algoritmo seria maior que $|M_n|$. Portanto, o algoritmo de saída não é $M_n$, e podemos seguramente responder "sim" à questão original de SATISFATIBILIDADE.

O que foi descrito no parágrafo anterior é um algoritmo de tempo polinomial para SATISFATIBILIDADE – o tempo limite $p(|R_1| + |M|)$ é polinomial em relação ao tamanho da fórmula booleana $F$. Como SATISFATIBILIDADE é um problema $\mathcal{NP}$-*completo*, devemos concluir, pelo Teorema 7.1.1, que $\mathcal{P} = \mathcal{NP}$, completando a prova. ■

**Problemas para a Seção 7.3**

**7.3.1** (a) Mostre que o problema da COBERTURA EXATA permanece $\mathcal{NP}$-completo mesmo que todos os conjuntos envolvidos não tenham mais que três elementos e que cada elemento apareça em, no máximo, três conjuntos. (b) O que acontece se um dos números for dois?

**7.3.2** Forneça o grafo *completo* (sem a abreviação proporcionada pelo dispositivo "ou exclusivo") que resultaria da redução DE COBERTURA EXATA para CIRCUITO DE HAMILTON se a instância DE COBERTURA EXATA consistisse do universo $\{u_1, u_2\}$ e da família de conjuntos $\mathcal{F} = \{\{u_1\}, \{u_1, u_2\}\}$.

**7.3.3** (a) Mostre que o problema do CAMINHO DE HAMILTON é $\mathcal{NP}$-completo, (1) reduzindo a ele o problema de CIRCUITO DE HAMILTON; (2) modificando ligeiramente a construção apresentada na prova de Teorema 7.3.2.

(b) Repetir para o problema do CAMINHO DE HAMILTON ENTRE DOIS VÉRTICES ESPECÍFICOS (a definição é óbvia).

**7.3.4** Cada um dos seguintes problemas é uma **generalização** de um problema $\mathcal{NP}$-completo e é, portanto, $\mathcal{NP}$-completo. Em outras palavras, se certos parâmetros do problema forem fixados de algum modo, então o problema em mãos torna-se um problema $\mathcal{NP}$-completo conhecido (ver a prova do Teorema 7.2.4). Pode-se reduzir qualquer problema à sua generalização simplesmente introduzindo um novo parâmetro e, caso contrário, deixando a instância da forma que estiver. Para cada um dos problemas abaixo, pode-se provar que é $\mathcal{NP}$-completo, mostrando que o mesmo é a generalização de um problema $\mathcal{NP}$-completo. Fornecer o vínculo do parâmetro apropriado em cada caso.



(a) CIRCUITO MAIS LONGO: dado um grafo e um número inteiro $K$, existe um circuito, sem vértices repetidos, de comprimento pelo menos $K$? (*sugestão*: o que acontece com esse problema se $K$ se restringir a ser igual ao número de vértices do grafo?)

(b) ISOMORFISMO DE SUBGRAFOS: dados dois grafos não-orientados $G$ e $H$, seria $G$ um subgrafo de $H$? (Em outras palavras, se $G$ tem os vértices $v_1, \ldots, v_n$, deseja-se encontrar vértices distintos $u_1, \ldots, u_n$ em $H$, de tal modo que $\{u_i, u_j\}$ seja uma aresta de $H$, sempre que $\{v_i, v_j\}$ for uma aresta em $G$)

(c) ISOMORFISMO DE SUBGRAFO INDUZIDO: dados dois grafos não-orientados $G$ e $H$, seria $G$ um subgrafo induzido de $H$? (Em outras palavras, se $G$ tem os vértices $v_1, \ldots, v_n$, deseja-se encontrar vértices distintos $v_1, \ldots, u_n$ em $H$ de tal modo que $\{u_i, u_j\}$ seja uma aresta de $H$ *se e somente se* $\{v_i, v_j\}$ é uma aresta em $G$)

(d) GRAFO CONFIÁVEL: dado um grafo não-orientado $G$ com vértices $v_1, \ldots, v_n$, uma matriz $R_{ij}$ simétrica, $n \times n$, de números naturais e um inteiro $B$, existe algum conjunto $S$ com $B$ arestas de $G$ com a seguinte propriedade: entre os vértices $v_i \neq v_j$ há, pelo menos, $R_{ij}$ caminhos disjuntos (que são caminhos que não compartilham nenhum vértice, além dos finais) com arestas em $S$? (*Sugestão*: Imaginar o que acontece se $B = n$ e $R_{ij} = 2$ para todos os $i,j$)

(e) PROGRAMAÇÃO INTEIRA: dadas $m$ equações

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, \ldots, m$$

sobre $n$ variáveis, com os coeficientes inteiros $a_{ij}$ e $b_i$. Têm elas uma solução em que todos os $x_j$'s são ou zero ou um? (Essa é uma generalização comum de *muitos* dos problemas $\mathcal{NP}$-completos estudados. Quais seriam eles?)

(f) TARIFA DE TAXI: dado um grafo orientado $G$ com os comprimentos positivos $d_{ij}$ em suas arestas, dois vértices 1 e $n$ e um inteiro $K$, existiria um caminho de 1 para $n$, não repetindo nenhum vértice, e com comprimento total $K$ ou mais?

(g) CONJUNTO DE COBERTURA ("HITTING SET"): dada uma família de conjuntos $\{S_1, S_2, \ldots, S_n\}$ e um inteiro $B$, existiria um conjunto $H$ com $B$ ou menos elementos, tal que a interseção de $H$ com cada conjunto da família não seja vazia.

(h) EMPACOTAMENTO ("BIN PACKING"): dado um conjunto de inteiros positivos $A = \{a_1, \ldots, a_n\}$ e dois inteiros, $B$ e $K$, podem os inteiros contidos em $A$ ser particionados em $B$ subconjuntos ("pacotes") de tal modo que os números em cada pacote totalizem $K$ ou menos?

(i) COBERTURA DE CONJUNTO: dada uma família $\mathcal{F}$ de subconjuntos de um universo $U$ e um inteiro $K$, existiriam $K$ conjuntos em $\mathcal{F}$ cuja união seja igual a $U$?

**7.3.5** Mostre que o problema dos CONJUNTOS INDEPENDENTES permanece $\mathcal{NP}$-completo, mesmo que o tamanho $K$ do CONJUNTO INDEPENDENTE procurado seja igual a $\lfloor n/2 \rfloor$, onde $n$ é o número de vértices.

**7.3.6** Mostre que o problema CONJUNTO DOMINANTE descrito a seguir é $\mathcal{NP}$-completo: dado um grafo orientado $G$ e um inteiro $B$, existiria algum conjunto $S$ com $B$ vértices de $G$, tal que, para cada vértice $u \notin S$ de $G$, haja um vértice $v \in S$, tal que $(v, u)$ seja uma aresta de $G$.

**7.3.7** Diz-se *acíclico* um autômato finito não-determinístico $M = (K, \Sigma, \Delta, s, F)$, para o qual não exista um estado $q$ e uma cadeia $w \neq \varepsilon$, tal que $(q, w) \vdash^*_M (q, \varepsilon)$. Mostre que o problema de determinar se dois autômatos finitos não-determinísticos acíclicos são inequivalentes é $\mathcal{NP}$-completo.

## 7.4 LIDANDO COM A COMPLETUDE NP

Os problemas não desaparecem quando se prova que eles são $\mathcal{NP}$-completos. Contudo, sabendo que os problemas em que estamos interessados fazem parte da classe dos problemas $\mathcal{NP}$-completos, tendemos a reduzir nossas expectativas, adotando soluções aproximadas, baseadas em algoritmos que nem sempre são polinomiais ou que não se apliquem obrigatoriamente em todas as possíveis instâncias. Nesta seção, revemos algumas das mais úteis manobras desse tipo.

### Casos particulares

Uma vez que nosso problema tenha sido identificado como $\mathcal{NP}$-completo, a primeira dúvida que surge é esta: precisamos *realmente* resolver esse problema em toda a generalidade na qual ele foi formulado – e identificado como $\mathcal{NP}$-completo? Reduções de *completude* $\mathcal{NP}$ muitas vezes produzem instâncias do problema que são artificialmente complexas. Talvez o problema que realmente precisamos resolver seja algum *caso particular* mais tratável do problema.

Por exemplo, já vimos um importante caso especial de SATISFATIBILIDADE que pode ser facilmente resolvido com eficiência: o problema da 2-SATISFATIBILIDADE (ver Seção 6.3). Se todas as instâncias de SATISFATIBILIDADE que necessitamos resolver apresentam cláusulas desse tipo, então o fato de ser o problema geral $\mathcal{NP}$-completo torna-se relativamente irrelevante. Todavia, com freqüência um caso particular de interesse revela-se, *ele próprio*, $\mathcal{NP}$-completo – por exemplo, 3-SATISFATIBILIDADE é um desses casos (ver Teorema 7.2.3). A seguir apresentamos outro exemplo.

**Exemplo 7.4.1:** A maioria dos problemas envolvendo grafos não-orientados torna-se menos complexo quando o grafo for uma árvore, ou seja, um grafo acíclico (ver Figura 7-12). Lembrando da nossa coleção de problemas



$\mathcal{NP}$-completos sobre grafos, o CIRCUITO DE HAMILTON apresenta uma execução trivial no caso de árvores (nenhuma árvore apresenta circuito de Hamilton, ou qualquer outro). Assim também se comporta o CAMINHO DE HAMILTON – a árvore tem um caminho de Hamilton somente se *ela for* um caminho de Hamilton. O problema da CLIQUE também se torna trivial, pois nenhuma árvore pode ter uma clique com mais de dois vértices.

O problema dos CONJUNTOS INDEPENDENTES também tem sua complexidade reduzida sempre que o grafo for uma árvore. O método utilizado em sua solução tira vantagem da estrutura hierárquica das árvores. É freqüentemente útil, dada uma árvore, selecionar um vértice arbitrário e designá-lo como **raiz** (ver a Figura 7-12); feito isso, cada vértice $u$ na árvore torna-se ele próprio a raiz de uma *subárvore* $T(u)$ – o conjunto de todos os vértices $v$, tal que o (único) caminho de $v$ para a raiz passa por $u$; ver a Figura 7-12. Então, esses problemas podem ser resolvidos de baixo para cima, partindo das folhas (subárvores com um vértice) em direção a subárvores cada vez maiores, até que toda a árvore (a subárvore da raiz) tenha sido abrangida. Para cada vértice $u$ podemos definir o conjunto de seus filhos $C(u)$ – os vértices, em sua subárvore, que são adjacentes a ele, excluindo o próprio $u$ – e seu conjunto de *netos* $G(u)$ – os filhos de seus filhos. Naturalmente, cada um desses conjuntos pode ser vazio. Por exemplo, na Figura 7-12, a raiz, denotada $r$, tem dois filhos e cinco netos. Vértices sem filhos são denominados *folhas* da árvore.

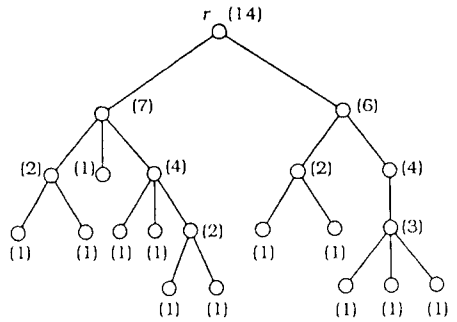



Figura 7-12

O tamanho do conjunto independente máximo da árvore pode agora ser encontrado computando, para cada vértice $u$, o número $I(u)$, definido como sendo o tamanho do conjunto independente máximo de $T(u)$. É fácil constatar a igualdade seguinte:

$$I(u) = \max\{ \sum_{v \in C(u)} I(v),\ 1 + \sum_{v \in G(u)} I(v) \} \qquad (2)$$

Essa equação informa que, ao designar o maior conjunto independente de $T(u)$, tem-se duas escolhas: ou (este é o primeiro argumento de max) não colocamos $u$ no conjunto independente, caso em que podemos juntar todos

os conjuntos máximos independentes contidos nas subárvores de seus filhos, ou (este é o segundo argumento) inserimos $u$ no conjunto independente, caso em que devemos omitir todos os seus filhos e juntar os conjuntos independentes máximos das subárvores de todos os seus netos.

Agora pode-se constatar que um algoritmo de *programação dinâmica* pode resolver o problema dos CONJUNTOS INDEPENDENTES, no seu caso particular de árvores, em tempo polinomial: o algoritmo se inicia nas folhas (onde $I(u)$ obviamente vale um) e computa $I(u)$ para subárvores cada vez maiores. O valor de $I$ na raiz é o tamanho do conjunto independente máximo da árvore. O algoritmo é polinomial, porque para cada vértice $u$, tudo o que temos a fazer é computar a expressão em (2), e essa tarefa é executada em tempo linear. Por exemplo, na árvore da Figura 7-12, os valores de $I(u)$ são mostrados entre parênteses. O conjunto independente máximo dessa árvore tem tamanho 14.

Obviamente, o problema mais intimamente relacionado, da COBERTURA DE VÉRTICES, também pode ser resolvido do mesmo modo (lembre-se das reduções entre COBERTURA DE VÉRTICES e CONJUNTOS INDEPENDENTES). Assim, se os grafos em que estamos interessados forem árvores, o fato de COBERTURA DE VÉRTICES e CONJUNTOS INDEPENDENTES serem $\mathcal{NP}$-completos é irrelevante. Muitos outros problemas $\mathcal{NP}$-completos sobre grafos são resolvidos por algoritmos similares quando especializados para árvores. Ver, por exemplo, o Problema 7.4.1. ◊

### Algoritmos de aproximação

Quando se encontra um problema de otimização $\mathcal{NP}$-completo, pode-se querer construir para ele algoritmos que não produzem soluções ótimas, mas soluções *que se garantam próximas do ótimo*. Suponha-se que se deseja obter tais soluções para um problema de otimização, maximização ou minimização. Para cada instância $x$ de tal problema, há uma solução ótima com valor opt($x$); admitindo que opt($x$) seja sempre um inteiro positivo (isso é o que ocorre com todos os problemas de otimização aqui estudados; pode-se facilmente indicar e resolver instâncias para as quais opt é zero).

Suponha-se agora que se dispõe de um algoritmo polinomial $A$ o qual, quando apresentado à instância $x$ do problema de otimização, produz alguma solução com valor $A(x)$. Como o problema é $\mathcal{NP}$-completo e $A$ é polinomial, não se pode realisticamente esperar que $A(x)$ seja sempre o valor ótimo. Suponhamos entretanto que se saiba que a seguinte desigualdade é sempre válida:

$$\frac{|\mathrm{opt}(x)-A(x)|}{\mathrm{opt}(x)} \leq \varepsilon,$$

onde $\varepsilon$ é algum número positivo real, preferencialmente muito pequeno, que limita superiormente o erro relativo no pior caso do algoritmo $A$. (Os valores absolutos, nessa desigualdade, nos permitem tratar tanto o problema da minimização como o da maximização usando a mesma estrutura.) Se o algoritmo $A$ satisfaz essa desigualdade para todas as instâncias $x$ do problema, então ele é chamado um **algoritmo de $\varepsilon$-aproximação**.

Constatado que o problema de otimização é $\mathcal{NP}$-completo, a seguinte questão torna-se importante: existem algoritmos de $\varepsilon$-aproximação para esse



problema? Se existem, quão pequeno $\varepsilon$ pode ser feito? Observe-se inicialmente que tais questões são significativas somente se admitirmos que $\mathcal{P} \neq \mathcal{NP}$, porque, se $\mathcal{P} = \mathcal{NP}$, então o problema pode ser resolvido de forma exata com $\varepsilon = 0$.

Todos os problemas de otimização $\mathcal{NP}$-completos podem, assim, ser subdivididos em três grandes categorias:

(a) Problemas **completamente aproximáveis**, no sentido de que existe um algoritmo de $\varepsilon$-aproximação, de tempo polinomial, para eles e para *todos os* $\varepsilon > 0$, embora pequenos. Entre os problemas de otimização $\mathcal{NP}$-completos que vimos, somente AGENDAMENTO DE DUAS MÁQUINAS (no qual desejamos minimizar o tempo de término $D$) caem nessa categoria mais privilegiada.

(b) Problemas **parcialmente aproximáveis**, no sentido de que há um algoritmo de $\varepsilon$-aproximação de tempo polinomial para eles em algum intervalo de $\varepsilon$'s, mas – exceto se $\mathcal{P} = \mathcal{NP}$ – esse intervalo não abrange todo o intervalo até zero, como os problemas completamente aproximáveis. Entre os problemas de otimização $\mathcal{NP}$-completos que vimos, COBERTURA DE VÉRTICES e MAX SAT se enquadram nessa classe intermediária.

(c) Problemas **não-aproximáveis**, isto é, para os quais não há algoritmo de $\varepsilon$-aproximação por maior que se escolha o valor de $\varepsilon$ – exceto se $\mathcal{P} = \mathcal{NP}$. Entre os problemas de otimização $\mathcal{NP}$-completos que vimos nesse capítulo, infelizmente muitos se classificam nessa categoria: o PROBLEMA DO CAIXEIRO-VIAJANTE, CLIQUE, CONJUNTOS INDEPENDENTES, bem como o problema de minimizar o número de estados de um autômato determinístico equivalente a uma dada expressão regular em tempo polinomial de saída (ver o corolário do Teorema 7.3.8).

**Exemplo 7.4.2:** Descreve-se a seguir um algoritmo de 1-aproximação para COBERTURA DE VÉRTICES – ou seja, um algoritmo geral que, para qualquer grafo, retorna uma COBERTURA DE VÉRTICES que tem no máximo *duas vezes* o tamanho ótimo. O algoritmo é muito simples:

```
C := ∅
enquanto existir uma aresta [u, v] em G faça
    acrescentar u e v a C e eliminá-los de G
```

Por exemplo, no grafo na Figura 7-13, o algoritmo poderia começar escolhendo a aresta $[a, b]$ e inserindo seus dois extremos em $C$; ambos os vértices (e suas arestas adjacentes, naturalmente) são então excluídos de $G$. A seguir $[e, f]$ pode ser escolhido e, por fim, $[g, h]$. O resultado do conjunto $C$ é uma COBERTURA DE VÉRTICES, porque cada aresta de $G$ deve tocar um de seus vértices (seja porque ele foi escolhido pelo algoritmo, seja porque foi por ele excluído). No presente exemplo, $C = \{a, b, e, f, g, h\}$, há seis vértices, os quais têm, no máximo, duas vezes o valor ótimo – neste caso, quatro.

Para provar a garantia de "no máximo duas vezes", considere-se a cobertura retornada pelo algoritmo e que $\hat{C}$ seja a cobertura de vértice ótima. $|C|$ corresponde exatamente ao dobro do número de arestas escolhido pelo algoritmo. Entretanto, essas arestas, exatamente pelo modo como foram escolhidas, não têm vértices em comum e, para cada uma delas, pelo menos

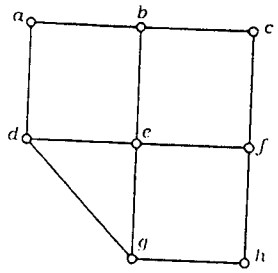



Figura 7-13

um de seus pontos extremos deve estar em $\hat{C}$ – porque $\hat{C}$ é uma cobertura de vértices. Portanto, o número de arestas escolhido pelo algoritmo não é maior que o conjunto de cobertura ótimo, e, portanto, $|C| \leq 2 \cdot |\hat{C}|$, e este é de fato um algoritmo de 1-aproximação.

Infelizmente, esse algoritmo simples de aproximação é o melhor que se conhece para o problema da COBERTURA DE VÉRTICES. E só muito recentemente foi possível provar que, a não ser que $\mathcal{P} = \mathcal{NP}$, não existe algoritmo de $\varepsilon$-aproximação para a COBERTURA DE VÉRTICES para qualquer $\varepsilon < \frac{1}{6}$. ✧

**Exemplo 7.4.3:** Entretanto, para o problema do ESCALONAMENTO DE DUAS MÁQUINAS, não há limite para o quanto podemos nos aproximar da solução ótima: para qualquer $\varepsilon > 0$ há um algoritmo de $\varepsilon$-aproximação para esse problema.

Essa família de algoritmos é baseada em uma idéia já apresentada: observe-se que o problema da PARTIÇÃO pode ser resolvido em tempo $O(nS)$ (onde $n$ é o número de inteiros e $S$ é sua soma; ver Seção 6.2). É fácil notar que esse algoritmo pode ser adaptado, de maneira relativamente trivial, para resolver o ESCALONAMENTO DE DUAS MÁQUINAS (encontrar o menor $D$): os conjuntos $B(i)$ são estendidos para incluir somas até $S$ (não apenas até $H = \frac{1}{2}S$). A menor soma em $B(n)$ que é $\geq \frac{1}{2} S$ identifica o $D$ mínimo desejado.

Uma idéia a mais é necessária para se chegar ao algoritmo desejado de aproximação: considere-se uma instância de ESCALONAMENTO DE DUAS MÁQUINAS com os seguintes comprimentos de tarefas:

45362, 134537, 85879, 56390, 145627, 197342, 83625, 126789, 38562, 75402,

com $n = 10$ e $S \approx 10^6$. Resolvê-lo por nosso algoritmo exato $O(nS)$ iria custar um nada atraente número de $10^7$ passos. Mas se, em vez disso, o comprimento das tarefas for arredondado para a próxima centena, obteríamos os números:

45400, 134600, 85900, 56400, 145700, 197400, 83700, 126800, 38600, 75500,

o que é praticamente o mesmo que (normalizando por 100):

454, 1346, 859, 564, 1457, 1974, 837, 1268, 386, 755,



Agora, podemos resolver essa instância em cerca de $10^5$ passos. Sacrificando um pouco a precisão (o ótimo do novo problema claramente estará não muito distante do original), reduzimos as exigências de tempo de uma centena de vezes!

É fácil provar que, se arredondarmos para a próxima $k$-ésima potência de dez, a diferença entre os dois valores ótimos obtidos não é superior a $n10^k$. Para calcular o erro relativo, essa diferença deve ser dividida pelo ótimo, o qual, obviamente, não pode ser menor que $\frac{S}{2}$. Tem-se, portanto, um algoritmo de $\frac{2n10^k}{S}$-aproximação, cujo tempo de execução é $O(\frac{nS}{10^k})$. Configurando $\frac{2n10^k}{S}$ igual a qualquer $\varepsilon > 0$ desejado chega-se a um algoritmo cujo tempo de execução é $O(\frac{n^2}{\varepsilon})$ – que é, certamente, um polinômio. ◊

**Exemplo 7.4.4:** Como provar que um problema é não-aproximável (ou não completamente aproximável)? Para a maioria dos problemas de otimização de maior interesse, essa questão tem sido uma das mais obstinadas em levantar novos problemas e exigir o desenvolvimento de novas idéias e técnicas matemáticas (ver referências ao final deste capítulo). O caso a seguir apresenta uma prova relativamente simples, e é o PROBLEMA DO CAIXEIRO-VIAJANTE.

Seja dado algum número $\varepsilon$ grande. Deseja-se provar que, a não ser que $\mathcal{P} = \mathcal{NP}$, não existe qualquer algoritmo de $\varepsilon$-aproximação para o PROBLEMA DO CAIXEIRO-VIAJANTE. Sabe-se que o problema do CIRCUITO DE HAMILTON é $\mathcal{NP}$-completo; deve-se mostrar que, se existir um algoritmo de $\varepsilon$-aproximação para o PROBLEMA DO CAIXEIRO-VIAJANTE, então existirá um algoritmo, de tempo polinomial, para o problema de CIRCUITO DE HAMILTON. Inicia-se com qualquer instância $G$ do problema de CIRCUITO DE HAMILTON, com $n$ vértices. Aplica-se a ele a redução simples do CIRCUITO DE HAMILTON para o PROBLEMA DO CAIXEIRO-VIAJANTE (ver a prova do Teorema 7.3.4), mas com uma alteração: as distâncias $d_{ij}$ são agora as seguintes (compare-se com as da prova do Teorema 7.3.4):

$$d_{ij} = \begin{cases} 0 & \text{se } i = j; \\ 1 & \text{se } (v_i, v_j) \in G; \\ 2 + n\varepsilon & \text{caso contrário.} \end{cases}$$

A instância construída apresenta a seguinte propriedade interessante: se $G$ tem um circuito de Hamilton, então o custo ótimo de uma viagem será $n$; se, entretanto, não apresentar qualquer circuito de Hamilton, então o custo ótimo será *maior que* $n(1 + \varepsilon)$ – porque ao menos uma distância $2 + n\varepsilon$ deve ser percorrida, além de, pelo menos $n - 1$ outros de custo pelo menos 1.

Considere-se a hipótese de existir um algoritmo $A$ de $\varepsilon$-aproximação de tempo polinomial para o PROBLEMA DO CAIXEIRO-VIAJANTE. Nesse caso, seria possível responder se $G$ tem ou não um circuito de Hamilton, da seguinte maneira: executa-se o algoritmo $A$ sobre a instância dada do PROBLEMA DO CAIXEIRO-VIAJANTE. Então, dois casos são possíveis:

(a) Se a solução retornada pelo algoritmo tiver um custo $\geq n(1 + \varepsilon) + 1$, pode-se afirmar que o ótimo não pode ser $n$, porque, neste caso, o erro relativo de $A$ teria sido, pelo menos, igual a

$$\frac{|n(1+\varepsilon) + 1 - n|}{n} > \varepsilon,$$

o que contradiz a hipótese de que $A$ é um algoritmo de $\varepsilon$-aproximação. Como a solução ótima é maior do que $n$, conclui-se que $G$ não apresenta circuitos de Hamilton.

(b) Se, ao contrário, a solução retornada por $A$ apresentar um custo $\leq n(1+\varepsilon)$, então a solução ótima deve ser $n$, porque a instância proposta foi concebida de maneira que não possa existir uma viagem com custo entre $n+1$ e $n(1+\varepsilon)$. Portanto, nesse caso, $G$ apresenta um circuito de Hamilton.

Assim, aplicando o algoritmo polinomial $A$ à instância do PROBLEMA DO CAIXEIRO-VIAJANTE que foi construída a partir de $G$, em tempo polinomial, pode-se afirmar se $G$ tem ou não um circuito de Hamilton – o que implica que $\mathcal{P} = \mathcal{NP}$. Como esse argumento pode ser executado para qualquer $\varepsilon > 0$, mesmo que seja grande, deve-se concluir que o PROBLEMA DO CAIXEIRO-VIAJANTE é não-aproximável. ◊

Maneiras diferentes de conviver com a *completude* $\mathcal{NP}$ muitas vezes podem ser combinadas: sabendo que o PROBLEMA DO CAIXEIRO-VIAJANTE é não-aproximável, pode-se desejar *aproximações para casos particulares* do problema. Consideremos o caso particular em que as distâncias $d_{ij}$ satisfazem a *desigualdade triângular*

$$d_{ij} \leq d_{ik} + d_{kj} \text{ para cada } i, j, k,$$

uma suposição relativamente natural em matrizes de distância, que corresponde à maioria das instâncias do PROBLEMA DO CAIXEIRO-VIAJANTE que surgem na prática. Esse caso particular se mostra parcialmente aproximável, e para ele o melhor limite conhecido de erro é $\frac{1}{2}$. Adicionalmente, quando as cidades se restringem a pontos no plano, guardando entre si distâncias euclidianas usuais – outro caso particular de óbvio interesse e relevância – então o problema torna-se até completamente aproximável! Os dois casos particulares são conhecidos como $\mathcal{NP}$-completos (ver Problema 7.4.3 para a prova do caso da desigualdade triângular).

### "*Backtracking*"[*] e ramificação limitada ("*branch-and-bound*")

Todos os problemas $\mathcal{NP}$-completos são, por definição, solúveis por máquinas de Turing não-determinísticas polinomialmente limitadas; infelizmente, dispõe-se apenas de métodos exponenciais para simular tais máquinas. Examina-se a seguir uma classe de algoritmos que se propõe a aperfeiçoar tais procedimentos exponenciais aplicando artifícios engenhosos, de acordo com o problema particular. Essa prática geralmente leva a algoritmos que são exponenciais no pior caso, mas com freqüência fazem muito melhor que isso.

Em problemas $\mathcal{NP}$-completos pergunta-se, tipicamente, se qualquer membro de um grande conjunto $S_0$ de propostas de certificados ou de testemunhas (atribuições, conjuntos de vértices, permutações de vértices e

[*] N. de R. Nesta versão, a palavra "*Backtracking*" foi mantida sem tradução, mas é aqui usada no sentido de "Método exaustivo que opera por tentativa e erro".



assim por diante – ver Seção 6.4) satisfazem certas limitações especificadas pela instância (se satisfaz todas as cláusulas, se é uma clique de tamanho $K$, se é um caminho de Hamilton). Para todos os problemas interessantes, os tamanhos do conjunto $S_0$, para todas as possíveis soluções, são, em geral, exponencialmente grandes, e dependem apenas da dada instância $x$ (seu tamanho depende exponencialmente do número de variáveis na fórmula, do número de vértices no grafo, e assim por diante).

Uma máquina de Turing não-determinística que resolve uma instância desse problema $\mathcal{NP}$-completo produz uma árvore de configurações (ver Figura 6-3). Cada uma dessas configurações corresponde a um subconjunto do conjunto $S_0$ de potenciais soluções, a que chamaremos genericamente $S$, e a "tarefa" enfrentada nessa configuração é determinar se existe alguma solução em $S$ satisfazendo as limitações de $x$. Portanto, $S_0$ corresponde ao conjunto que representa a configuração inicial. Responder se $S$ contém ou não uma solução é, muitas vezes, um problema não muito diferente do original. Portanto, é possível verificar cada uma das configurações na árvore como um *subproblema* do mesmo tipo que o original (essa útil propriedade de auto-similaridade de problemas $\mathcal{NP}$-completos é chamada *auto-redutibilidade*). Fazer uma escolha não-determinística a partir de uma configuração, para as $r$ próximas configurações possíveis, corresponde a substituir $S$ por $r$ conjuntos, $S_1, \ldots, S_r$, cuja união deve ser $S$, de modo tal que nenhuma solução candidata caia entre as lacunas.

Isso sugere o seguinte gênero de algoritmos para resolver problemas $\mathcal{NP}$-completos: manter-se permanentemente um conjunto $A$ de *subproblemas ativos*. Inicialmente, $A$ contém somente o problema original $S_0$, isto é, $A = \{S_0\}$. Em cada ponto, escolhe-se um subproblema de $A$ (talvez o que aparente ser o mais promissor). Remove-se então de $A$ esse subproblema, substituindo-o por vários subproblemas mais simples (cuja união de soluções propostas deve cobrir apenas o que foi removido). A essa prática dá-se o nome de *ramificação*.

Em seguida, cada subproblema recém-gerado é submetido a um rápido *teste heurístico*. Esse teste analisa um subproblema e gera para ele uma das respostas seguintes:

(a) Uma resposta "vazia", o que significa que o subproblema considerado não tem soluções que satisfazem os vínculos da instância, e, portanto, pode ser omitido. Essa prática denomina-se "***backtracking***".

(b) Uma solução real do problema original contido no subproblema atual (uma atribuição satisfatória para a fórmula original, um circuito de Hamilton do grafo original, etc.), e nesse caso o algoritmo termina de forma bem-sucedida.

(c) Sendo o problema $\mathcal{NP}$-completo, não se pode esperar que haja um teste heurístico rápido que sempre ofereça uma das respostas acima (caso contrário, submeteríamos o subproblema original $S_0$ a ele). Portanto, o teste irá muitas vezes responder "?", significando que ele é incapaz tanto de provar que o subproblema é vazio quanto de localizar rapidamente uma solução para ele; nesse caso, inserimos o subproblema em questão no conjunto $A$ de subproblemas ativos. A esperança é que o teste ofereça uma das duas outras respostas com suficiente freqüência, reduzindo assim substancialmente o número de

subproblemas a serem examinados – e, portanto, o tempo de execução do algoritmo.

Pode-se assim formular inteiramente o **algoritmo de *backtracking***:

$\mathcal{A} := \{S_0\}$
enquanto $A$ não estiver vazio faça
  escolher um subproblema $S$, e elimine-o de $A$
  escolher um modo de ramificar $S$ em subproblemas, digamos, $S_1, \ldots, S_r$
  Para cada subproblema $S_i$ dessa lista faça
    se teste($S_i$) retornar "solução encontrada", então terminar
    senão se teste($S_i$) retornar "?", então acrescentar $S_i$ em $A$
retornar "sem solução"

O **algoritmo de "*backtracking*"** sempre termina porque, ao final de sua execução, os subproblemas acabam por tornar-se tão pequenos e especializados que conterão apenas uma solução candidata (correspondendo às folhas da árvore de computação não-determinística); neste caso, o teste será capaz de decidir rapidamente se essa solução satisfaz ou não as exigências impostas à instância.

A eficácia de um **algoritmo de "*backtracking*"** depende de três importantes decisões de projeto:

(1) A maneira de escolher o próximo subproblema a partir do qual efetuar a ramificação.
(2) A maneira de refinar o subproblema escolhido em outros subproblemas mais simples.
(3) O tipo de teste a ser utilizado.

**Exemplo 7.4.5:** Para projetar um **algoritmo de "*backtracking*"** para o problema da SATISFATIBILIDADE, deve-se instanciar as decisões de projeto (1) a (3) acima.

No caso da SATISFATIBILIDADE, o modo mais intuitivo de refinar um subproblema é escolher uma variável $x$, e então criar dois novos subproblemas: um para o qual $x = \top$, e outro para o qual $x = \bot$. Conforme foi comentado, cada subproblema é do mesmo tipo que o problema original: um conjunto de cláusulas com menos variáveis (mais uma atribuição fixa para cada uma das variáveis originais que não aparecem no subproblema em questão). No subproblema $x = \top$, as cláusulas em que $x$ aparece são omitidas, e $\bar{x}$ é omitido das cláusulas em que ele aparece, enquanto o inverso ocorre no subproblema $x = \bot$.

A questão relativa à decisão de projeto (2) consiste em determinar a variável $x$ a partir da qual se deve efetuar a ramificação, e para isso pode-se adotar como regra *escolher aquela variável que aparece na menor cláusula* (se houver empates, a decição pode ser arbitrária). Essa é uma estratégia sensível, porque cláusulas menores representam vínculos mais exigentes, podendo conduzir mais cedo ao *"backtracking"*. Em particular, uma cláusula vazia é um inconfundível sinal de não-satisfatilidade.

Para a decisão de projeto (1) (como escolher o próximo subproblema), conforme a estratégia adotada para (2), pode-se escolher o subproblema que contiver a menor cláusula (novamente, decidindo empates arbitrariamente).



Por fim, quanto ao teste a ser adotado (decisão de projeto (3)) pode-se proceder da seguinte forma:

se existir uma cláusula vazia, retornar "o subproblema é vazio";
se não existirem cláusulas, retornar "solução encontrada";
caso contrário, retornar "?"

Observe-se a Figura 7-14 para uma aplicação do **algoritmo de "*backtracking*"** descrito acima para a instância seguinte:

$$(x \vee y \vee z), (\bar{x} \vee y), (\bar{y} \vee z), (\bar{z} \vee x), (\bar{x} \vee \bar{y} \vee \bar{z}),$$

a qual já sabemos que é insatisfazível (ver Exemplo 6.3.3). Esse algoritmo é uma variante de um conhecido algoritmo para o problema da SATISFATIBILIDADE, denominado **procedimento de Davis-Putnam**. Em particular, quando a instância tem no máximo dois literais por cláusula, o **algoritmo de "*backtracking*"** *torna-se exatamente o algoritmo polinomial de eliminação* da seção *6.3*. ◊

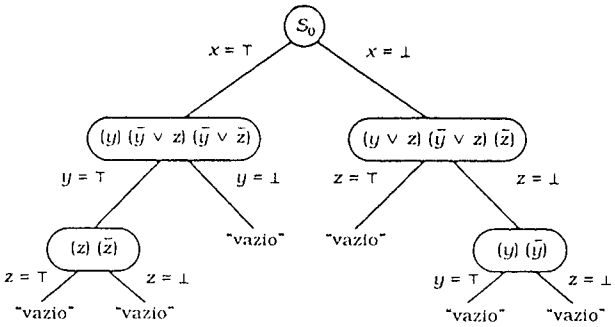



Figura 7-14

**Exemplo 7.4.6:** Neste exemplo, projeta-se um **algoritmo de "*backtracking*"** para o problema do CIRCUITO DE HAMILTON. Em cada subproblema já existe um caminho com pontos finais $a$ e $b$, o qual percorre um conjunto de vértices $T \subseteq V - \{a, b\}$. Estamos procurando um caminho de Hamilton de $a$ para $b$ pelos vértices restantes em $V$, para fechar o circuito de Hamilton. Inicialmente $a = b$ é um vértice arbitrário e $T = \emptyset$.

A ramificação é simples – escolhe-se como estender o caminho por uma nova aresta $\{a, c\}$, que leva de $a$ para um vértice $c \notin T$. Esse vértice $c$ torna-se o novo valor de $a$ no subproblema (o vértice $b$ é sempre fixo ao longo de todo o processamento). Deixa-se não-especificado o subproblema da escolha do vértice a partir da qual ramificar (escolhe-se um subproblema qualquer de $A$). Por fim, o teste escolhido é o seguinte (lembrar que, em um subproblema, procura-se um caminho de $a$ para $b$ em um grafo $G - T$, o grafo original com os vértices em $T$ excluídos).

se $G - T - \{a, b\}$ é desconexo ou se $G - T$ possui um vértice de grau um,
além de $a$ ou $b$, retornar "o subproblema é vazio";
se $G - T$ é um caminho de $a$ para $b$, retornar "solução encontrada";
caso contrário retornar "?"

A aplicação desse algoritmo a um grafo simples é mostrada na Figura 7-15. Apesar de o número de soluções parciais construídas parecer grande (dezenove), ele é insignificante quando comparado ao número de soluções examinadas pelo algoritmo não-determinístico completo para a mesma instância (esse número seria $(n-1)! = 5.040$). Obviamente, é possível conceber regras e testes de ramificação mais sofisticados e eficazes que os aqui utilizados. ◊

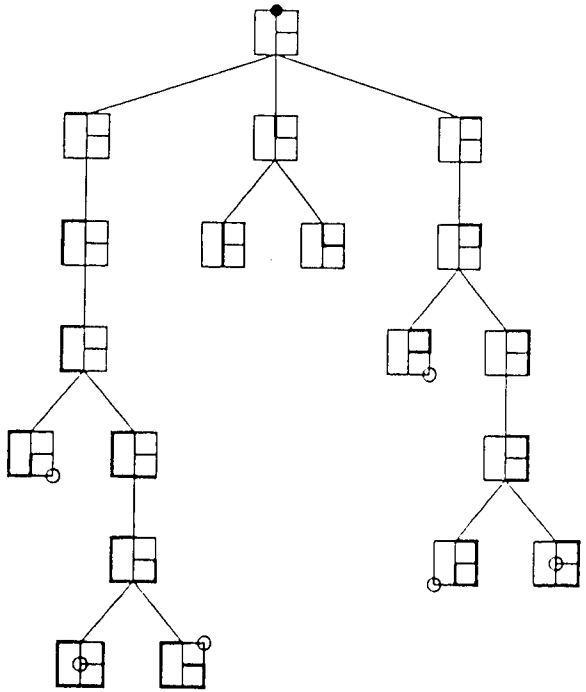


Figura 7-15 Execução do algoritmo de "*backtracking*" para o problema do CIRCUITO DE HAMILTON no grafo mostrado na raiz. Inicialmente $a$ e $b$ coincidem com o vértice tracejado. Na folha (reversão) os vértices de grau um são circulados (no meio das folhas há muitas escolhas). Um total de dezenove subproblemas é considerado.

A determinação das melhores decisões de projeto (1) a (3) depende não somente do problema, mas também dos tipos de instâncias de interesse, e geralmente requer *experimentação exaustiva*.

Algoritmos de "*backtracking*" são de grande interesse quando se propõem a resolver problemas com respostas do tipo "sim-não". Para problemas de otimização, muitas vezes utiliza-se uma variação interessante do "backtracking" chamada ramificação limitada. Em problemas de otimização, pode-se também considerar a existência de um conjunto exponencialmente grande de propostas de soluções. Entretanto, desta vez, cada solução apresenta um custo† associado, e se deseja localizar a proposta da solução que apresente o menor custo. O algoritmo de ramificação-limitada é apresentado a seguir (o algoritmo apresentado somente retorna o custo ótimo, mas pode ser facilmente modificado, para retornar a solução ótima).



$\mathcal{A} := \{S_0\}$; melhorateagora:= $\infty$;
enquanto $A$ não estiver vazio faça
  escolher um subproblema $S$; removê-lo de $\mathcal{A}$
  escolher um modo de refinar $S$, em subproblemas $S_1, \ldots, S_r$
  para cada subproblema $S_i$ desta lista faça
    se $|S_i| = 1$ (isto é, $S_i$ é uma solução completa)então atualizar melhorateagora
    senão se limiteinferior($S_i$) <melhorateagora então incluir $S_i$ em $\mathcal{A}$
retornar melhorateagora

O algoritmo sempre memoriza o menor custo dentre as soluções analizadas, até então, inicialmente $\infty$ (o desempenho muitas vezes melhora bastante, se melhorateagora for iniciado com o custo de alguma solução obtida por outra heurística). Cada vez que é encontrada uma solução completa para o problema original, melhorateagora é atualizado. O ingrediente-chave de um algoritmo do tipo ramificação-limitada (além das decisões de projeto (1) e (2), que ele compartilha com o "*backtracking*") é alguma forma de obtenção de um *limite mais baixo* para o custo de qualquer solução em um subproblema $S$, isto é, uma função limiteinferior($S$) que retorne um número que seja garantidamente menor ou igual ao custo mais baixo de qualquer das soluções em $S$. O algoritmo de ramificação-limitada irá terminar sempre com a solução ótima. Isso ocorre porque os únicos subproblemas ainda não considerados são exatamente aqueles para os quais limiteinferior($S_i$) ≥ melhorateagora – ou seja, aqueles cuja solução ótima é, comprovadamente, não melhor que a melhor solução obtida até o momento.

Há muitas maneiras de obter limites mais baixos que os obtidos, como por exemplo, fazendo melhorateagora($S$) = 0. No entanto, se limiteinferior ($S$) for um algoritmo sofisticado que retorna um valor geralmente muito próximo da solução ótima em $S$, então é provável que o método de ramificação limitada também apresente bom desempenho, isto é, termine de maneira razoavelmente rápida.

**Exemplo 7.4.7:** Vamos adaptar o **algoritmo de "*backtracking*"**, desenvolvido para problema do CIRCUITO DE HAMILTON, para obter um algoritmo de ramificação limitada para o PROBLEMA DO CAIXEIRO-VIAJANTE. Analogamente ao que foi visto, um subproblema $S$ é caracterizado por um caminho de $a$ para $b$ através de um conjunto $T$ de cidades. Uma idéia para um limite inferior razoável é a seguinte: para cada cidade fora de $T \cup \{a, b\}$, calcula-se a soma das *duas distâncias* mais *curtas* para outra cidade fora de $T$. Para $a$ e $b$,

† Admite-se neste texto que o problema de otimização que está sendo estudado seja um problema de minimização; os problemas de maximização podem ser tratados de maneira muito semelhante.

calcula-se sua distância mais curta para outra cidade fora de $T$. Não é difícil provar (ver Problema 7.4.4) que a *metade* da soma desses números mais o custo do caminho já definido de $a$ para $b$ através de $T$, é um limite inferior válido ao custo de qualquer viagem no subproblema $S$. O algoritmo de ramificação-limitada ficou agora completamente especificado.

Há muitos limites inferiores mais sofisticados para o PROBLEMA DO CAIXEIRO-VIAJANTE. ✧

### Melhora local

A última família de algoritmos que estamos apresentando é inspirada pela evolução: vamos investigar o que ocorre permitindo uma pequena mudança na solução de um problema de otimização e adotando essa nova solução mesmo que ela possa aumentar o custo. Seja $S_0$ o conjunto das soluções candidatas em uma instância de um problema de otimização (trata-se novamente de um problema de minimização). Define-se uma **relação de vizinhança** $N$ no conjunto de soluções $N \subseteq S_0 \times S_0$ – ela concretiza a noção intuitiva de "pequena mudança". Para $s \in S_0$, o conjunto $\{s' : (s, s') \in N\}$ é dito **vizinhança** de $s$.

O algoritmo é simplesmente este (ver Figura 7-16, onde se encontra uma representação da operação dos algoritmos de melhora local):

```
s :=soluçãoinicial
enquanto existir uma solução s', tal que
    N(s, s') e custo(s') <custo(s) faça: s:= s'
retornar s
```

Esse algoritmo continua melhorando $s$, substituindo-o por um vizinho $s'$ que exiba melhor custo, até que não haja $s'$, na vizinhança de $s$, que apresente um custo melhor; neste último caso, dize-se que *$s$ é um ótimo local*. Obviamente, não se garante que um ótimo local seja uma solução ótima – a não ser que $N = S_0 \times S_0$, pois a qualidade do ótimo local obtido e o tempo de execução do algoritmo dependam fortemente de $N$: quanto maiores as vizinhanças, melhor o ótimo local; por outro lado, amplas vizinhanças implicam que a iteração do algoritmo (uma execução do loop enquanto e a conseqüente busca na vizinhança da solução $s$ atual) será lenta. Algoritmos de melhora local buscam um equilíbrio nessa negociação. Como de costume, não há princípios gerais para nos guiar ao designar-mos uma boa vizinhança; a escolha parece muito dependente do problema, e até mesmo da instância, sendo melhor realizada por experimentação.

Outra questão que afeta o desempenho de um algoritmo de melhora local é o método utilizado para encontrar $s'$. Adota-se a primeira melhor solução encontrada na vizinhança de $s$ ou se deve procurar a melhor de todas. Seria justificável um custo maior em troca da maior velocidade de descida assim obtida? Seria a alta velocidade realmente desejável? Por fim, o desempenho de um algoritmo de melhora local também depende do



procedimento da soluçãoinicial. Não é claro, afinal, que as melhores soluções iniciais resultarão em melhor desempenho – muitas vezes um ponto de partida medíocre é preferível, porque ele deixa o algoritmo mais livre para explorar o espaço das soluções (ver Figura 7-16). Assim, o procedimento soluçãoinicial deve ser melhor *aleatorizado* – isto é, capaz de gerar diferentes soluções iniciais quando solicitado diversas vezes. Isso nos permite *reiniciar* repetidamente o algoritmo de melhora local acima para obter diversos ótimos locais.

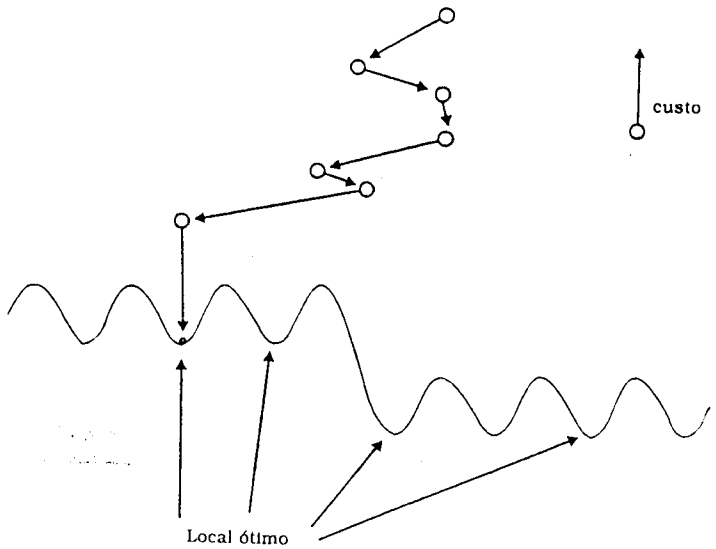



Figura 7-16 Uma vez fixada a relação de vizinhança, as soluções para um problema de otimização podem ser representadas como um *gráfico de energia*, em que os ótimos locais são representados como vales. A heurística da melhora local percorre solução por solução, até encontrar um ótimo local.

**Exemplo 7.4.8:** Considere-se novamente o PROBLEMA DO CAIXEIRO-VIAJANTE. Em quais situações se deve considerar duas viagens como vizinhas? Como uma viagem pode ser considerada como um conjunto de $n$ "conexões" não-orientadas entre as cidades, uma resposta plausível é: *quando todas compartilham as mesmas poucas conexões*. Dois é o número mínimo possível de conexões em que duas viagens podem diferir, e isso sugere uma conhecida relação de vizinhança com o PROBLEMA DO CAIXEIRO-VIAJANTE, denominada **2-mudanças** (ver Figura 7-17). Duas viagens são relacionadas por $N$ se e somente se diferem em apenas duas conexões. O algoritmo de melhora local que utiliza a vizinhança de **2-mudanças** funciona razoavelmente bem na prática. Entretanto, resultados muito melhores podem ser

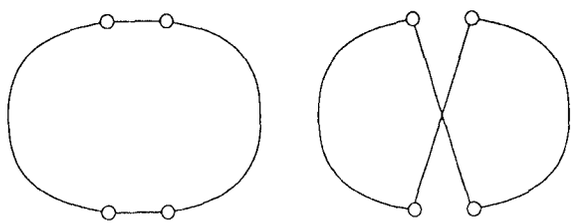


Figura 7-17

obtidos adotando-se a vizinhança de **3-mudanças**; além disso, encontra-se na literatura que **4-mudanças** não retorna viagens que se mostrem suficientemente melhores para justificar uma redução no tempo de execução da iteração.

Provavelmente o melhor algoritmo heurístico atualmente conhecido para o PROBLEMA DO CAIXEIRO-VIAJANTE, o **algoritmo de Lin-Kernighan**, se baseia em **λ–mudanças**, uma vizinhança tão sofisticada e complexa que nem sequer se adapta à nossa estrutura (determinar se duas soluções são vizinhas depende da distância). Como o próprio nome sugere, λ-mudanças permite muitas trocas arbitrária de conexão em um passo (mas naturalmente, nem todas as possíveis mudanças desse tipo são exploradas, pois isso tornaria a iteração exponencialmente lenta). ✧

**Exemplo 7.4.9:** Para desenvolver um algoritmo de melhora local para o problema MAX SAT (versão de SATISFATIBILIDADE que se propõe a satisfazer o máximo possível de cláusulas – ver Teorema 7.2.4), pode-se considerar que duas atribuições estão relacionadas por $N$ no caso de elas diferirem somente no valor de uma única variável. Isso define um interessante e empiricamente bem-sucedido algoritmo de melhora local para MAX SAT. É aparentemente vantajoso, nesse caso, adotar como $s'$ o melhor vizinho de $s$, em lugar do primeiro encontrado que seja melhor que $s$. Além disso, conclui-se que vale a pena efetuar "movimentos laterais" (aceitar uma proposta de solução mesmo que a desigualdade da terceira linha do algoritmo não seja estrita).

Essa heurística é considerada um modo muito eficaz de obter boas soluções para o problema MAX SAT, e é muitas vezes utilizada para resolver a SATISFATIBILIDADE (quando empregada dessa maneira, espera-se que, ao final do processamento, o algoritmo retorne uma atribuição que satisfaça *todas as cláusulas*). ✧

Uma interessante modificação nos algoritmos de melhora local é um método chamado ***simulated annealing.*** Como o nome sugere, a inspiração vem da física do resfriamento dos sólidos. O *simulated annealing* permite que o algoritmo "escape" de seus ótimos locais falsos (ver Figura 7-18 e comparar com a Figura 7-17) executando ocasionais mudanças que aumentam o custo.



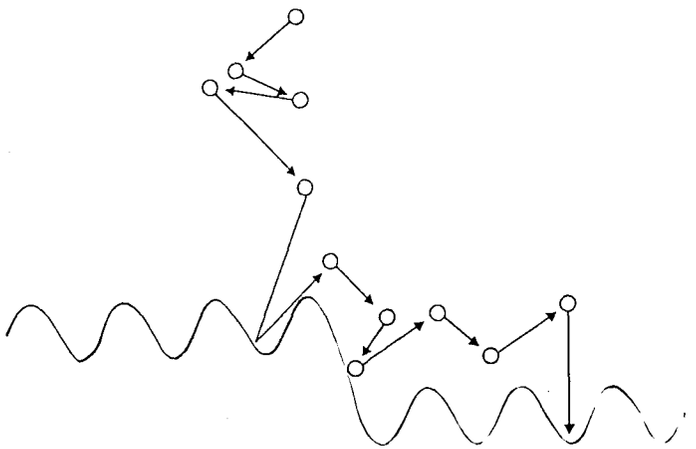


Figura 7-18 O método do recozimento simulado tem uma vantagem sobre o algoritmo básico de melhora local, porque seu eventual aumento de custo ajuda a evitar a convergência prematura para um ótimo local falso. Isso, muitas vezes, acarreta uma grande perda de eficiência.

```
s :=soluçãoinicial; T:= T0;
repita
    gerar uma solução aleatória s' tal que N(s, s'),
        e fazer Δ:=custo(s') – custo(s)
    se Δ ≤ 0 então s:= s', senão
        s:= s' com probabilidade e^(-Δ/T)
    atualizar(T)
até T = 0
retornar a melhor solução observada.
```

Intuitivamente, a probabilidade de que seja adotada alguma mudança que aumente o custo é determinada pelo grau de redução no custo Δ, bem como por um importante parâmetro, a **temperatura** $T$. Quanto mais alta a temperatura, mais agressivamente são procuradas soluções dispendiosas. O modo como $T$ é atualizado na penúltima linha do algoritmo – o agendamento do algoritmo do *simulated annealing*, como é chamado – talvez seja a mais crucial decisão de projeto nesses algoritmos – além, naturalmente, da escolha da vizinhança.

Há vários outros tipos de métodos de melhora local, muitos deles baseados, como os anteriormente descritos, em alguma analogia com *sistemas* físicos ou biológicos (*algoritmos genéticos*, *redes neurais*, etc. – ver as referências).

Do ponto de vista do critério formal desenvolvido neste livro, os algoritmos de melhora local e suas muitas variantes não são nada atraentes: em geral não retornam a solução ótima, tendem a ter complexidade exponencial

de pior caso, e nem mesmo garantem a obtenção de soluções que sejam, em qualquer sentido bem definido, próximas da ótima. Apesar disso, para muitos problemas $\mathcal{NP}$-completos, na prática eles muitas vezes se revelam como tendo melhor desempenho! A explicação e previsão do impressionante sucesso empírico de alguns desses algoritmos é atualmente uma das mais desafiantes fronteiras da teoria da computação.

**Problemas para a Seção 7.4**

**7.4.1** Apresentar um algoritmo polinomial para resolver o problema do CONJUNTO DOMINANTE (ver Problema 7.3.6) no caso especial das árvores (consideradas como grafos simétricos orientados).

**7.4.2** Suponha-se que todas as cláusulas em uma instância de satisfatibilidade contenham, no máximo, *um literal positivo*; tais cláusulas são chamadas **cláusulas de Horn.** Mostre que, se todas as cláusulas de uma fórmula booleana forem cláusulas de Horn, então a questão da satisfatibilidade para essa fórmula pode ser estabelecida em tempo polinomial. (*Sugestão:* Estudar em que situações é necessário atribuir T a uma variável em uma fórmula de Horn.)

**7.4.3** Mostrar que o PROBLEMA DO CAIXEIRO-VIAJANTE permanece $\mathcal{NP}$-completo mesmo que se exija que as distâncias obedeçam à desigualdade triangular. (*Sugestão:* Consulte a prova, apresentada anteriormente, de que o problema do caixeiro-viajante é $\mathcal{NP}$-completo.)

**7.4.4** Suponha-se que, em uma instância do problema do caixeiro-viajante com cidades 1, 2, ..., $n$ e matriz de distâncias $d_{ij}$, são consideradas apenas viagens que começam em $a$, percorrendo algum caminho de comprimento $L$ envolvendo as cidades em um conjunto $T \subseteq \{1, 2, ..., n\}$, e terminando em outra cidade $b$, e, então, visitando as cidades restantes e retornando para $a$. A esse conjunto de viagens denominamos $S$.
(a) Para cada cidade $i \in \{1, 2, ..., n\} - T - \{a, b\}$, seja $m_i$ a soma das duas menores distâncias de $i$ para outra cidade em $\{1, 2, ..., n\} - T - \{a, b\}$. Seja $s$ a distância mais curta de $a$ para qualquer cidade em $\{1, 2, ..., n\} - T - \{a, b\}$, mais a distância mais curta a partir de $b$. Mostrar que qualquer viagem em $S$ tem custo de, pelo menos,

$$L + \frac{1}{2}\left[\sum_{i \in \{1,2,...,n\} - T - \{a,b\}} m_i + s\right]$$

A fórmula acima é um limite inferior válido para $S$.
(b) A **árvore geradora mínima** das $n$ cidades é a menor árvore que tem as cidades como um conjunto de vértices; ela pode ser computada de maneira muito eficiente. Deduza para $S$ um limite inferior melhor a partir dessa informação.

**7.4.5** Quantos vizinhos de **2-mudanças** tem uma viagem de $n$ cidades? Quantos de **3-mudanças**? Quantos de **4-mudanças**?



**7.4.6** (a) Suponha-se que, no algoritmo *simulated annealing*, a temperatura seja mantida em zero. Mostrar que esse é o algoritmo básico de melhora local.
(b) Qual é o algoritmo *simulated annealing* cuja temperatura seja mantida no infinito?
(c) Suponha-se agora que a temperatura seja zero para algumas iterações, então infinita para outras, então zero novamente, etc. De que modo o algoritmo resultante se relaciona com a versão básica da melhora local?

## REFERÊNCIAS


*Stephen A. Cook foi o primeiro a mostrar uma linguagem $\mathcal{NP}$-completa em seu trabalho:*

- S. A. Cook "The Complexity of Theorem-Proving Procedures", *Proceedings of the Third Annual ACM Symposium on the Theory of Computing* pp. 151-158). New York: Association for Computing Machinery, 1971.

*Richard M. Karp estabeleceu a extensão e a importância da completude $\mathcal{NP}$ em sua obra:*

- R. M. Karp "*Reducibility among Combinatorial Problems*", in Complexity of Computer Computations, (pp. 85-104), ed. R. E. Miller and J. W. Thatcher. New York: Plenum Press, 1972.

*onde, entre vários outros resultados, os Teoremas 7.3.1 – 7.3.7 e os enunciados dos problemas 7.3.4 e 7.3.6, são provados. A completude $\mathcal{NP}$ foi independentemente descoberta por Leonid Levin em:*

- L. A. Levin "Universal Sorting Problems", *Problemi Peredachi Informatsii*, 9, 3, pp. 265-266 (em russo), 1973.

*O seguinte livro contém um útil catálogo de mais de 300 problemas $\mathcal{NP}$-completos de diversas áreas: muitos outros problemas foram provados como $\mathcal{NP}$-completos desde seu aparecimento.*

- M. R. Garey and D. S. Johnson *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$*-completeness, New York: Freeman, 1979.

*Este livro também é uma excelente fonte de informação sobre o modo como a complexidade se aplica a problemas concretos, bem como sobre algoritmos de aproximação. Para tratamentos muito mais recentes e extensos deste último assunto, veja-se:*

- D. Hochbaum (ed.) *Approximation Algorithms for $\mathcal{NP}$*-hard Problems, Boston, Mass: PWS Publishers, 1996.

*E, para maiores informações sobre outras maneiras de tratar a completude $\mathcal{NP}$, veja-se por exemplo:*

- C. R. Reeves, (ed.) *Modern Heuristic Techniques for Combinatorial Problems*, New York: John Wiley, 1993, e
- C. H. Papadimitriou and K. Steiglitz *Combinatorial Optimization: Algorithms and Complexity* Englewood Cliffs, N.J.: Prentice-Hall, 1982; Second edition, New York: Daven 1997.