7

COMPLEXIDADE DE TEMPO

Mesmo quando um problema é decidível e, portanto, computacionalmente solúvel em princípio, ele pode não ser solúvel na prática se a solução requer uma quantidade desordenada de tempo ou memória. Nesta parte final do livro introduzimos a teoria da complexidade computacional—uma investigação do tempo, memória ou outros recursos requeridos para resolver problemas computacionais. Começamos com tempo.

Nosso objetivo neste capítulo é apresentar o básico da teoria da complexidade de tempo. Primeiro introduzimos uma maneira de medir o tempo usado para resolver um problema. Então mostramos como classificar problemas de acordo com a quantidade de tempo necessária. Depois disso discutimos a possibilidade de que certos problemas decidíveis requerem quantidades enormes de tempo e como determinar quando você está diante de um problema desses.

7.1 MEDINDO COMPLEXIDADE

Vamos começar com um exemplo. Tome a linguagem $A = \{0^k1^k | \ k \geq 0\}$. Obviamente A é uma linguagem decidível. Quanto tempo uma máquina de Turing de uma única fita precisa para decidir A? Examinamos a seguinte MT de uma

única-fita M_1 para A. Damos a descrição da máquina de Turing num nível baixo, incluindo a própria movimentação da cabeça sobre a fita de modo que possamos contar o número de passos que M_1 usa quando ela roda.

M_1 = "Sobre a cadeia de entrada w:

1. Faça uma varredura na fita e *rejeite* se um `0` for encontrado à direita de um `1`.
2. Repita se ambos `0`s e `1`s permanecem sobre a fita:
3. Faça uma varredura na fita, cortando um único `0` e um único `1`.
4. Se `0`s ainda permanecerem após todos os `1`s tiverem sido cortados, ou se `1`s ainda permanecerem após todos os `0`s tiverem sido cortados, *rejeite*. Caso contrário, se nem `0`s nem `1`s permanecererem sobre a fita, *aceite*."

Analisamos o algoritmo para a MT M_1 decidindo A para determinar quanto tempo ela usa.

O número de passos que um algoritmo usa sobre uma entrada específica pode depender de vários parâmetros. Por exemplo, se a entrada for um grafo, o número de passos pode depender do número de nós, o número de arestas, e o grau máximo do grafo, ou alguma combinação desses e/ou de outros fatores. Para simplicidade computamos o tempo de execução de um algoritmo puramente como uma função do comprimento da cadeia representando a entrada e não consideramos quaisquer outros parâmetros. Na ***análise do pior-caso***, a forma que consideramos aqui, levamos em conta o tempo de execução mais longo de todas as entradas de um comprimento específico. Na ***análise do caso-médio***, consideramos a média dos tempos de execução de entradas de um comprimento específico.

DEFINIÇÃO 7.1

Seja M uma máquina de Turing determinística que pára sobre todas as entradas. O ***tempo de execução*** ou ***complexidade de tempo*** de M é a função $f: \mathcal{N} \longrightarrow \mathcal{N}$, onde $f(n)$ é o número máximo de passos que M usa sobre qualquer entrada de comprimento n. Se $f(n)$ for o tempo de execução de M, dizemos que M roda em tempo $f(n)$ e que M é uma máquina de Turing de tempo $f(n)$. Costumeiramente usamos n para representar o comprimento da entrada.

NOTAÇÃO O-GRANDE E O-PEQUENO

Em razão do fato de que o tempo exato de execução de um algoritmo freqüentemente é uma expressão complexa, usualmente apenas o estimamos. Em uma

forma mais conveniente de estimativa, chamada ***análise assintótica***, buscamos entender o tempo de execução do algoritmo quando ele é executado sobre entradas grandes. Fazemos isso considerando apenas o termo de mais alta ordem da expressão para o tempo de execução do algoritmo, desconsiderando tanto o coeficiente daquele termo quanto quaisquer termos de ordem mais baixa, porque o termo de mais alta ordem domina os outros termos sobre entradas grandes.

Por exemplo, a função $f(n) = 6n^3 + 2n^2 + 20n + 45$ tem quatro termos, e o termo de mais alta ordem é $6n^3$. Desconsiderando o coeficiente 6, dizemos que f é assintoticamente no máximo n^3. A ***notação assintótica*** ou ***notação O-grande*** para descrever esse relacionamento é $f(n) = O(n^3)$. Formalizamos essa noção na definição seguinte. Seja $\mathcal{R}^+$ o conjunto de números reais não-negativos.

DEFINIÇÃO 7.2

Sejam f e g be funções $f, g\colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Vamos dizer que $\boldsymbol{f(n) = O(g(n))}$ se inteiros positivos c e n_0 existem tais que para todo inteiro $n \geq n_0$

$$f(n) \leq c\, g(n).$$

Quando $f(n) = O(g(n))$ dizemos que $g(n)$ é um ***limitante superior*** para $f(n)$, ou mais precisamente, que $g(n)$ é um ***limitante superior assintótico*** para $f(n)$, para enfatizar que estamos suprimindo fatores constantes.

Intuitivamente, $f(n) = O(g(n))$ significa que f é menor ou igual a g se desconsiderarmos diferenças até um fator constante. Você pode pensar em O como representando uma constante suprimida. Na prática, a maioria das funções f que você tende a encontrar tem um termo óbvio de mais alta ordem h. Nesse caso escrevemos $f(n) = O(g(n))$, onde g é h sem seu coeficiente.

EXEMPLO 7.3

Seja $f_1(n)$ a função $5n^3 + 2n^2 + 22n + 6$. Então, selecionando o termo de mais alta ordem $5n^3$ e desconsiderando seu coeficiente 5 dá $f_1(n) = O(n^3)$.

Vamos verificar que esse resultado satisfaz a definição formal. Fazemos isso tornando c igual a 6 e n_0 igual a 10. Então, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ para todo $n \geq 10$.

Adicionalmente, $f_1(n) = O(n^4)$ porque n^4 é maior que n^3 e portanto é ainda um limitante assintótico superior sobre f_1.

Entretanto, $f_1(n)$ não é $O(n^2)$. Independentemente dos valores que atribuímos a c e n_0, a definição permanece insatisfeita nesse caso. ■

EXEMPLO 7.4

O O-grande interage com logaritmos de uma maneira particular. Normalmente quando usamos logaritmos temos que especificar a base, como em $x = \log_2 n$. A base 2 aqui indica que essa igualdade é equivalente à igualdade $2^x = n$. Mudando o valor da base b muda o valor de $\log_b n$ por um fator constante, devido à identidade $\log_b n = \log_2 n / \log_2 b$. Por conseguinte, quando escrevemos $f(n) = O(\log n)$, especificando a base não é mais necessária porque estamos de qualquer forma suprimindo fatores constantes.

Seja $f_2(n)$ a função $3n \log_2 n + 5n \log_2 \log_2 n + 2$. Nesse caso temos $f_2(n) = O(n \log n)$ porque $\log n$ domina $\log \log n$. ∎

A notação O-grande também aparece nas expressões aritméticas tais como a expressão $f(n) = O(n^2) + O(n)$. Nesse caso cada ocorrência do símbolo O representa uma constante suprimida diferente. Em razão do termo $O(n^2)$ dominar o termo $O(n)$, essa expressão é equivalente a $f(n) = O(n^2)$. Quando o símbolo O ocorre num expoente, como na expressão $f(n) = 2^{O(n)}$, a mesma idéia se aplica. Essa expressão representa um limitante superior de 2^{cn} para alguma constante c.

A expressão $f(n) = 2^{O(\log n)}$ ocorre em algumas análises. Usando a identidade $n = 2^{\log_2 n}$ e portanto que $n^c = 2^{c \log_2 n}$, vemos que $2^{O(\log n)}$ representa um limitante superior de n^c para alguma c. A expressão $n^{O(1)}$ representa o mesmo limitante de uma maneira diferente, porque a expressão $O(1)$ representa um valor que nunca é mais que uma constante fixa.

Freqüentemente derivamos limitantes da forma n^c para c maior que 0. Tais limitantes são chamados ***limitantes polinomiais***. Limitantes da forma $2^{(n^\delta)}$ são chamados ***limitantes exponenciais*** quando δ é um número real maior que 0.

A notação O-grande tem uma companheira chamada ***notação o-pequeno***. A notação O-grande diz que uma função é assintoticamente *não mais que* uma outra. Para dizer que uma função é asintoticamente *menor que* uma outra usamos a notação o-pequeno. A diferença entre as notações O-grande e o-pequeno é análoga à diferença entre $\leq$ e $<$.

DEFINIÇÃO 7.5

Sejam f e g be funções $f, g\colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Vamos dizer que $\boldsymbol{f(n) = o(g(n))}$ se

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Em outras palavras, $f(n) = o(g(n))$ significa que, para qualquer número real $c > 0$, um número n_0 existe, onde $f(n) < c\, g(n)$ para todo $n \geq n_0$.

EXEMPLO 7.6

O que segue é fácil de verificar.

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log \log n)$.
3. $n \log \log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.

Entretanto, $f(n)$ nunca é $o(f(n))$. ∎

ANALISANDO ALGORITMOS

Vamos analisar o algoritmo de MT que demos para a linguagem $A = \{0^k1^k \mid k \geq 0\}$. Repetimos o algoritmo aqui por conveniência.

M_1 = "Sobre a cadeia de entrada w:

1. Faça uma varredura na fita e *rejeite* se um 0 for encontrado à direita de um 1.
2. Repita se ambos 0s e 1s permanecem sobre a fita:
3. Faça uma varredura na fita, cortando um único 0 e um único 1.
4. Se 0s ainda permanecerem após todos os 1s tiverem sido cortados, ou se 1s ainda permanecerem após todos os 0s tiverem sido cortados, *rejeite*. Caso contrário, se nem 0s nem 1s permanecererem sobre a fita, *aceite*."

Para analisar M_1 consideramos cada um dos seus quatro estágios separadamente. No estágio 1, a máquina faz uma varredura na fita para verificar que a entrada é da forma 0^*1^*. Realizar essa varredura usa n passos. Como mencionamos anteriormente, tipicamente usamos n para representar o comprimento da entrada. Reposicionar a cabeça na extremidade esquerda da fita usa outros n passos. Assim, o total usado nesse estágio é $2n$ passos. Na notação O-grande dizemos que esse estágio usa $O(n)$ passos. Note que não mencionamos o reposicionamento da cabeça da fita na descrição da máquina. Usando uma notação assintótica nos permite omitir detalhes da descrição da máquina que afetam o tempo de execução por no máximo um fator constante.

Nos estágios 2 e 3, a máquina repetidamente faz uma varredura na fita e corta um 0 e um 1 em cada varredura. Cada varredura usa $O(n)$ passos. Em razão do fato de que cada varredura corta dois símbolos, no máximo $n/2$ varreduras podem ocorrer. Portanto, o tempo total tomado pelos estágios 2 e 3 é $(n/2)O(n) = O(n^2)$ passos.

No estágio 4 a máquina faz uma única varredura para decidir se aceita ou rejeita. O tempo tomado nesse estágio é no máximo $O(n)$.

Portanto o tempo total de M_1 sobre uma entrada de comprimento n é $O(n) + O(n^2) + O(n)$, ou $O(n^2)$. Em outras palavras, seu tempo de execução é $O(n^2)$, o que completa a análise de tempo dessa máquina.

Vamos fixar um pouco de notação para classificar linguagens conforme seus requisitos de tempo.

DEFINIÇÃO 7.7

Seja $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ uma função. Defina a ***classe de complexidade de tempo***, $\mathbf{TIME}(t(n))$, como sendo a coleção de todas as linguagens que são decidíveis por uma máquina de Turing de tempo $O(t(n))$.

Retomemos a linguagem $A = \{0^k1^k \mid k \geq 0\}$. A análise precedente mostra que $A \in \text{TIME}(n^2)$ because M_1 decide A em tempo $O(n^2)$ e $\text{TIME}(n^2)$ contém todas as linguagens que podem ser decididas em tempo $O(n^2)$.

Existe uma máquina que decide A assintoticamente mais rapidamente? Em outras palavras, A está em $\text{TIME}(t(n))$ para $t(n) = o(n^2)$? Podemos melhorar o tempo de execução cortando dois 0s e dois 1s em cada varredura ao invés de apenas um porque fazendo isso corta-se o número de varreduras pela metade. Mas isso melhora o tempo de execução apenas por um fator de 2 e não afeta o tempo de execução assintótico. A máquina seguinte, M_2, usa um método diferente para decidir A assintoticamente mais rápido. Ela mostra que $A \in \text{TIME}(n \log n)$.

M_2 = "Sobre a cadeia de entrada w:

1. Faça uma varredura na fita e *rejeite* se um 0 for encontrado à direita de um 1.
2. Repita enquanto alguns 0s e alguns 1s permanecerem sobre a fita:
3. Faça uma varredura na fita, verificando se o número total de 0s e 1s remanescentes é par ou ímpar. Se for ímpar, *rejeite*.
4. Faça uma varredura novamente na fita, cortando alternadamente um 0 não e outro sim começando com o primeiro 0, e então cortando alternadamente um 1 não e outro sim começando com o primeiro 1.
5. Se nenhum 0s e nenhum 1s permanecerem sobre a fita, *aceite*. Caso contrário, *rejeite*."

Antes de analisar M_2, vamos verificar que ela realmente decide A. Em toda varredura realizada no estágio 4, o número total de 0s remanescentes é cortado pela metade e qualquer resto é descartado. Por conseguinte, se começarmos com 13 0s, após o estágio 4 ser executado uma única vez apenas 6 0s permanecem. Após

execuções subseqüentes desse estágio, 3, então 1, e depois 0 permanecem. Esse estágio tem o mesmo efeito sobre o número de 1s.

Agora examinamos a paridade par/ímpar do número de 0s e o número de 1s em cada execução do estágio 3. Considere novamente começar com 13 0s e 13 1s. A primeira execução do estágio 3 encontra um número ímpar de 0s (porque 13 é um número ímpar) e um número ímpar de 1s. Em execuções subseqüentes um número par (6) ocorre, então um número ímpar (3), e um número ímpar (1). Não executamos esse estágio sobre 0 0s ou 0 1s em razão da condição do laço de repetição especificada no estágio 2. Para a seqüência de paridades encontradas (ímpar, par, ímpar, ímpar) se substituirmos as pares por 0s e as ímpares por 1s e então revertermos a seqüência, obtemos 1101, a representação binária de 13, ou o número de 0s e 1s no início. A seqüência de paridades sempre dá o reverso da representação binária.

Quando o estágio 3 verifica para determinar que o número total de 0s e 1s remanescentes é par, ele na verdade está checando a concordância entre a paridade dos 0s com a paridade dos 1s. Se todas as paridades estão de acordo, as representações binárias dos números de 0s e de 1s concordam, e portanto os dois números são iguais.

Para analisar o tempo de execução de M_2, primeiro observamos que todo estágio leva um tempo $O(n)$. Então determinamos o número de vezes que cada um é executado. Os estágios 1 e 5 são executados uma vez, levando um total de tempo de $O(n)$. O estágio 4 corta pelo menos metade dos 0s e 1s cada vez que é executado, portanto no máximo $1 + \log_2 n$ iterações do laço de repetição ocorrem antes que todos sejam cortados. Por conseguinte, o tempo total dos estágios 2, 3 e 4 é $(1 + \log_2 n)O(n)$, ou $O(n \log n)$. O tempo de execução de M_2 é $O(n) + O(n \log n) = O(n \log n)$.

Anteriormente mostramos que $A \in \text{TIME}(n^2)$, mas agora temos um limitante melhor—a saber, $A \in \text{TIME}(n \log n)$. Esse resultado não pode ser melhorado ainda mais em máquinas de Turing de uma única fita. Na realidade, qualquer linguagem que pode ser decidida em tempo $o(n \log n)$ em uma máquina de Turing de uma única fita é regular, como o Problema 7.47 pede para você mostrar.

Podemos decidir a linguagem A em tempo $O(n)$ (também chamado ***tempo linear***) se a máquina de Turing tiver uma segunda fita. A seguinte MT de duas fitas M_3 decide A em tempo linear. A máquina M_3 opera diferentemente das máquinas anteriores para A. Ela simplesmente copia os 0s para sua segunda fita e então os confronta com os 1s.

M_3 = "Sobre a cadeia de entrada w:

1. Faça uma varredura na fita e *rejeite* se um 0 for encontrado à direita de um 1.
2. Faça uma varredura nos 0s sobre a fita 1 até o primeiro 1. Ao mesmo tempo, copie os 0s para a fita 2.
3. Faça uma varredura nos 1s sobre a fita 1 até o final da entrada. Para cada 1 lido sobre a fita 1, corte um 0 sobre a fita 2. Se todos os 0s estiverem cortados antes que todos os 1s sejam lidos,

rejeite.

4. Se todos os 0s tiverem agora sido cortados, *aceite*. Se algum 0 permanecer, *rejeite*."

Essa máquina é simples de analisar. Cada um dos quatro estágios usa $O(n)$ passos, portanto o tempo tota de execução é $O(n)$ e portanto é linear. Note que esse tempo de execução é o melhor possível porque n passos são necessários somente para ler a entrada.

Vamos resumir o que mostramos sobre a complexidade de tempo de A, a quantidade de tempo requerido para se decidir A. Produzimos uma MT de uma única fita M_1 que decide A em tempo $O(n^2)$ e uma MT de uma única fita mais rápida M_2 que decide A em tempo $O(n \log n)$. A solução para o Problema 7.47 implica que nenhuma MT de uma única fita pode fazê-lo mais rapidamente. Então exibimos uma MT de duas fitas M_3 que decide A em tempo $O(n)$. Logo, a complexidade de tempo de A numa MT de uma única fita é $O(n \log n)$ e numa MT de duas fitas é $O(n)$. Note que a complexidade de A depende do modelo de computação escolhido.

Essa discussão destaca uma importante diferença entre a teoria da complexidade e a teoria da computabilidade. Na teoria da computabilidade, a tese de Church–Turing implica que todos os modelos razoáveis de computação são are equivalentes—ou seja, todos eles decidem a mesma classe de linguagens. Na teoria da complexidade, a escolha do modelo afeta a complexidade de tempo de linguagens. Linguagens que decidíveis em, digamos, tempo linear em um modelo não são necessariamente decidíveis em tempo linear em um outro.

Na teoria da complexidade, classificamos problemas computacionais conforme sua complexidade de tempo. Mas com qual modelo medimos tempo? A mesma linguagem pode ter requisitos de tempo diferentes em modelos diferentes.

Felizmente, requisitos de tempo não diferem enormemente para modelos determinísticos típicos. Assim, se nosso sistema de classificação não for muito sensível a diferenças relativamente pequenas em complexidade, a escolha do modelo determinístico não é crucial. Discutimos essa idéia ainda mais nas próximas seções.

RELACIONAMENTOS DE COMPLEXIDADE ENTRE MODELOS

Aqui examinamos como a escolha do modelo computacional pode afetar a complexidade de tempo de linguagens. Consideramos três modelos: a máquina de Turing de uma única fita; a máquina de Turing multifita; e a máquina de Turing não-determinística.

TEOREMA 7.8

Seja $t(n)$ uma função, onde $t(n) \geq n$. Então toda máquina de Turing multifita de

tempo $t(n)$ tem uma máquina de Turing de uma única fita equivalente de tempo $O(t^2(n))$.

IDÉIA DA PROVA A idéia por trás da prova desse teorema é bastante simples. Lembre-se de que no Teorema 3.13 mostramos como converter qualquer MT multifita numa MT de uma única fita que a simula. Agora analisamos aquela simulação para determinar quanto tempo adicional ela requer. Mostramos que simular cada passo da máquina multifita usa no máximo $O(t(n))$ passos na máquina de uma única fita. Logo, o tempo total usado é $O(t^2(n))$ passos.

PROVA Seja M uma MT de k-fitas que roda em tempo $t(n)$. Construímos uma MT de uma única fita S que roda em tempo $O(t^2(n))$.

A máquina S opera simulando M, como descrito no Teorema 3.13. Para revisar aquela simulação, lembramos que S usa sua única fita para representar o conteúdo sobre todas as k fitas de M. As fitas são armazenadas consecutivamente, com as posiçcões das cabeças de M marcadas sobre as células apropriadas.

Inicialmente, S coloca sua fita no formato que representa todas as fitas de M e aí então simula os passos de M. Para simular um passo, S faz uma varredura em toda a informação armazenada na sua fita para determinar os símbolos sob as cabeças das fitas de M. Então S faz uma outra passagem sobre sua fita para atualizar o conteúdo da fita e das posições das cabeças. Se uma das cabeças de M move para a direita sobre a porção anteriormente não lida de sua fita, S tem que aumentar a quantidade de espaço alocado para sua fita. Ela faz isso deslocando uma porção de sua própria fita uma célula para a direita.

Agora analisamos essa simulação. Para cada passo de M, a máquina S faz duas passagens sobre a porção ativa de sua fita. A primeira obtém a informação necessária para determinar o próximo movimento e a segunda o realiza. O comprimento da porção ativa da fita de S determina quanto tempo S leva para varrê-la, por isso temos que determinar um limitante superior para esse comprimento. Para fazer isso tomamos a soma dos comprimentos das porções ativas das k fitas de M. Cada uma dessas porções ativas tem comprimento no máximo $t(n)$ porque M usa $t(n)$ células de fita em $t(n)$ passos se a cabeça move para a direita em todo passo e muito menos se uma cabeça em algum momento move para a esquerda. Por conseguinte, uma varredura da porção ativa da fita de S usa $O(t(n))$ passos.

Para simular cada um dos passos de M, S realiza duas varreduras e possivelmente até k deslocamentos para a direita. Cada uma usa um tempo $O(t(n))$, portanto o tempo total para S simular um dos passos de M é $O(t(n))$.

Agora limitamos o tempo total usado pela simulação. O estágio inicial, onde S coloca sua fita no formato apropriado, usa $O(n)$ passos. Depois disso, S simula cada um dos $t(n)$ passos de M, usando $O(t(n))$ passos, portanto essa parte da simulação usa $t(n) \times O(t(n)) = O(t^2(n))$ passos. Conseqüentemente, a simulação inteira de M usa $O(n) + O(t^2(n))$ passos.

Assumimos que $t(n) \geq n$ (uma suposição razoável porque M não poderia nem mesmo ler a entrada toda em menos tempo). Por conseguinte, o tempo de

execução de S é $O(t^2(n))$ e a prova está completa.

A seguir, consideramos o teorema análogo para máquinas de Turing não-determinísticas de uma-única-fita. Mostramos que qualquer linguagem que é decidível sobre uma dessas máquinas é decidível sobre uma máquina de Turing determinística de uma-única-fita que requer significativamente mais tempo. Antes de fazê-lo, temos que definir o tempo de execução de uma máquina de Turing não-determinística. Lembre-se de que uma máquina de Turing não-determinística é um decisor se todos os ramos de sua computação param sobre todas as entradas.

DEFINIÇÃO 7.9

Seja N uma máquina de Turing não-determinística que é um decisor. O ***tempo de execução*** de N é a função $f: \mathcal{N} \longrightarrow \mathcal{N}$, onde $f(n)$ é o número máximo de passos que N usa sobre qualquer ramo de sua computação sobre qualquer entrada de comprimento n, como mostrado na Figura 7.10.

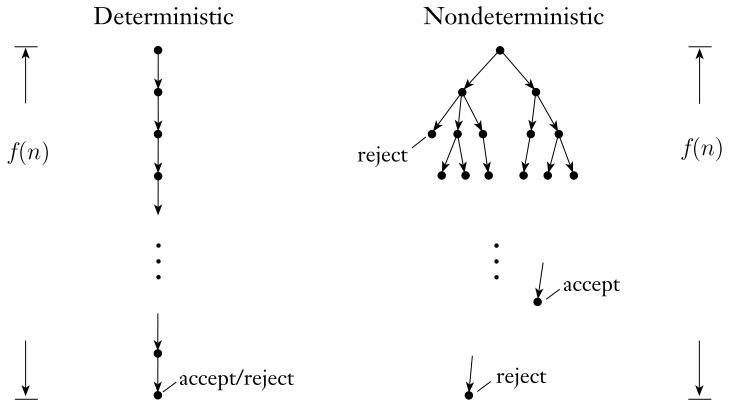


FIGURA 7.10
Medindo tempo determinístico e não-determinístico

A definição do tempo de execução de uma máquina de Turing não-determinística não tem o objetivo de corresponder a nenhum dispositivo de computação do mundo-real. Ao contrário, ela é uma definição matemática útil que assiste na caracterização da complexidade uma classe importante de problemas computacionais, como demonstramos em breve.

TEOREMA 7.11

Seja $t(n)$ uma função, onde $t(n) \geq n$. Então toda máquina de Turing não-determinística de uma-única-fita de tempo $t(n)$ tem uma máquina de Turing não-determinística de uma-única-fita de tempo $2^{O(t(n))}$.

PROVA Seja N uma MT não-determinística rodando em tempo $t(n)$. Construímos uma MT determinística D que simula N como na prova do Teorema 3.16 fazendo uma busca na árvore de computação não-determinísica de N. Agora analisamos essa simulação.

Sobre uma entrada de comprimento n, todo ramo da árvore de computação não-determinística de N tem um comprimento no máximo $t(n)$. Todo nó na árvore pode ter no máximo b filhos, onde b é o número máximo de escolhas legais dado pela funcção de transição de N. Portanto, o número total de folhas na árvore é no máximo $b^{t(n)}$.

A simulação procede explorando sua árvore na disciplina de busca por largura. Em outras palavras, ela visita todos os nós de profundidade d antes de continuar para quaisquer dos nós na profundidade $d + 1$. O algoritmo dado na prova do Teorema 3.16 ineficientemente começa na raiz e desce para um nó sempre que ele visita esse nó, mas eliminando essa ineficiência não altera o enunciado do teorema corrente, portanto deixamos como está. O número total de nós na árvore é menor que duas vezes o número máximo de folhas, portanto limitamo-lo por $O(b^{t(n)})$. O tempo para iniciar da raiz e descer a um nó é $O(t(n))$. Conseqüentemente, o tempo de execução de D é $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

Conforme descrito no Teorema 3.16, a MT D tem três fitas. Converter para uma MT de uma-única-fita no máximo eleva ao quadrado o tempo de execução, pelo Teorema 7.8. Por conseguinte, o tempo de execução do simulador de uma-única-fita é $\left(2^{O(t(n))}\right)^2 = 2^{O(2t(n))} = 2^{O(t(n))}$, e o teorema está provado.

7.2

A CLASSE P

Os Teoremas 7.8 e 7.11 ilustram uma importante distinção. Por um lado, demonstramos uma diferença de no máximo uma potência quadrática ou *polinomial* entre a complexidade de tempo de problemas medida em máquinas de Turing determinísticas de uma-única-fita e multifitas. Por outro lado, mostramos uma diferença no máximo *exponencial* entre a complexidade de tempo de problemas em máquinas de Turing determinístics e não-determinísticas.

TEMPO POLINOMIAL

Para nossos propósitos, diferenças polinomiais em tempo de execução são consideradas pequenas, enquanto que diferenças exponenciais são consideradas grandes. Vamos verpor que escolhemos fazer essa separação entre polinômios e exponenciais ao invés de entre algumas outras classes de funções.

Primeiro, note a dramática diferença entre a taxa de crescimento de polinômios que ocorrem tipicamente tais como n^3 e exponenciais típicas tais como 2^n. Por exemplo, suponha que n seja 1000, o tamanho de uma entrada razoável para um algoritmo. Nesse caso, n^3 é 1 bilhão, um número grande porém administrável, enquanto que 2^n é um número muito maior que o número de átomos no universo. Algoritmos de tempo polinomial são suficientemente rápidos para muitos propósitos, mas algoritmos de tempo exponencial raramente são úteis.

Algoritmos de tempo exponencial surgem tipicamente quando resolvemos problemas buscando exaustivamente dentro de um espaço de soluções, denominada ***busca pela força-bruta***. Por exemplo, uma maneira de fatorar um número em seus primos constituintes é buscar por todos os potenciais divisores. O tamanho do espaço de busca é exponencial, portanto essa busca usa tempo exponencial. Às vezes, a busca por força-bruta pode ser evitada através de um entendimento mais profundo de um problema, que pode revelar um algoritmo de tempo polinomial de utilidade maior.

Todos os modelos computacionais determinísticos razoáveis são ***polinomialmente equivalentes***. Ou seja, qualquer um deles pode simular um outro com apenas um aumento polinomial no tempo de execução. Quando dizemos que todos os modelos determinísticos razoáveis são polinomialmente equivalentes, não tentamos definir *razoável*. Entretanto, temos em mente uma noção suficientemente ampla para incluir modelos que aproximam de perto os tempos de execução em computadores reais. Por exemplo, o Teorema 7.8 mostra que os modelos de máquina de Turing determinística de uma-única-fita e multifita são polinomialmente equivalentes.

Daqui por diante focalizaremos em aspectos da teoria da complexidade de tempo que não são afetados por diferenças polinomiais em tempo de execução. Consideramos tais diferenças como sendo insignificantes e as ignoramos. Fazer isso nos permite desenvolver a teoria de uma maneira que não depende da escolha de um modelo específico de computação. Lembre-se de que nosso objetivo é apresentar as propriedades fundamentais da *computação*, ao invés das máquinas de Turing ou qualquer outro modelo especial.

Você pode achar que desconsiderar diferenças polinomiais em tempo de execução é absurdo. Programadores reais certamente se preocupam com tais diferenças e trabalham duro somente para fazer com que seus programas rodem duas vezes mais rápido. Entretanto, desconsideramos fatores constantes pouco tempo atrás quando introduzimos a notação assintótica. Agora propomos desconsiderar as diferenças muito maiores polinomiais, tais como aquela entre tempo n e tempo n^3.

Nossa decisão de desconsiderar diferenças polinomiais não implica que consideramos tais diferenças desimportantes. Ao contrário, certamente consideramos

a diferença entre tempo n e tempo n^3 como sendo uma importante diferença. Mas algumas questões, tais como a polinomialidade ou não-polinomialidade do problema da fatoração, não dependem das diferenças polinomiais e são importantes também. Meramente escolhemos focar nesse tipo de questão aqui. Ignorar as árvores para ver a floresta não significa que uma é mais importante que a outra—isso simplesmente dá uma perspectiva diferente.

Agora chegamos a uma importante definição em teoria da complexidade.

DEFINIÇÃO 7.12

P é a classe de linguagens que são decidíveis em tempo polinomial sobre uma máquina de Turing determinística de uma-única-fita. Em outras palavras,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

A classe P tem um papel central em nossa teoria e é importante porque

1. P é invariante para todos os modelos de computação que são polinomialmente equivalentes à máquina de Turing determinística de uma-única-fita, e
2. P aproximadamente corresponde à classe de problemas que são realisticamente solúveis num computador.

O item 1 indica que P é uma classe matematicamente robusta. Ela não é't afetada pelos particulares do modelo de computação que estamos usando.

O item 2 indica que P é relevante de um ponto de vista prático. Quando um problema está em P, temos um método de resolvê-lo que roda em tempo n^k para alguma constante k. Se esse tempo de execução é prático depende de k e da aplicação. É claro que um tempo de execução de n^{100} é improvável de ser de qualquer uso prático. Não obstante, chamar de tempo polinomial o limiar de solubilidade prática tem provado ser útil. Uma vez que um algoritmo de tempo polinomial tenha sido encontrado para um problema que anteriormente parecia requerer tempo exponencial, alguma percepção chave sobre ele foi obtida, e reduções adicionais na sua complexidade usuamente seguem, freqüentemente a ponto de real utilidade prática.

EXEMPLOS DE PROBLEMAS EM P

Quando apresentamos um algoritmo de tempo polinomial, damos uma descrição de alto-nível sem referência a características de um modelo computacional específico. Fazendo-se isso evita-se detalhes tediosos de movimentos de fitas e de cabeças. Precisamos seguir certas convenções ao descrever um algoritmo de modo que possamos analisá-lo com vistas à polynomialidade.

Descrevemos algoritmos com estágios numerados. A noção de um estágio de um algoritmo é análoga a um passo de uma máquina de Turing, embora é claro que implementar um estágio de um algoritmo numa máquina de Turing, em geral, vai requerer muitos passos de máquina de Turing.

Quando analisamos um algoritmo para mostrar que ele roda em tempo polinomial, precisamos fazer duas coisas. Primeiro, temos que dar um limitante superior polinomial (usualmente em notação O-grande) para o número de estágios que o algoritmo usa quando ele roda sobre uma entrada de comprimento n. Então, temos que examinar os estágios individuais na descrição do algoritmo para assegurar que cada um possa ser implementado em tempo polinomial num modelo determinístico razoável. Escolhemos os estágios quando descrevemos o algoritmo para tornar essa segunda parte da análise fácil de fazer. Quando ambas as tarefas tiverem sido completadas, podemos concluir que o algoritmo roda em tempo polinomial porque demonstramos que ele roda por um número polinomial de estágios, cada um dos quais pode ser feito em tempo polinomial, e a composição de polinômios é um polinômio.

Um ponto que requer atenção é o método de codificação usado para os problemas. Continuamos a usar a notação entre-colchetes $\langle \cdot \rangle$ para indicar uma codificação razoável de um ou mais objetos em uma cadeia, sem especificar qualquer método de codificação específico. Agora, um método razoável é aquele que permite codificação e decodificação de objetos em tempo polinomial em representações internas naturais ou em outras codificações razoáveis. Métodos de codificação familiares para grafos, autômatos e coisas do gênero são razoáveis. Mas note que notação unária para codificar números (como no número 17 codificado pela cadeia unária `11111111111111111`) não é razoável porque é exponencialmente maior que codificações verdadeiramente razoáveis, tais como notação na base k para qualquer $k \geq 2$.

Muitos problemas computacionais que você encontra neste capítulo contêm codificações de grafos. Uma codificação razoável de um grafo é uma lista de seus nós e arestas. Uma outra é a ***matriz de adjacência***, onde a (i, j)-ésima entrada é 1 se existe uma aresta do nó i para o nó j e 0 caso contrário. Quando analisamos algoritmos sobre grafos, o tempo de execução pode ser calculado em termos do número de nós ao invés do tamanho da representação do grafo. Em representações razoáveis de grafos, o tamanho da representação é um polinômio no número de nós. Por conseguinte, se analisamos um algoritm e mostramos que seu tempo de execução é polinomial (ou exponencial) no número de nós, sabemos que ele é polinomial (ou exponencial) no tamanho da entrada.

O primeiro problema concerne grafos direcionados. Um grafo direcionado G contém os nós s e t, como mostrado na Figura 7.13. O problema *CAMINH* é determinar se um caminho direcionado existe de s para t. Seja

CAMINH $= \{\langle G, s, t\rangle \mid G$ é um grafo direcionado que tem um caminho direcionado de s para $t\}$.

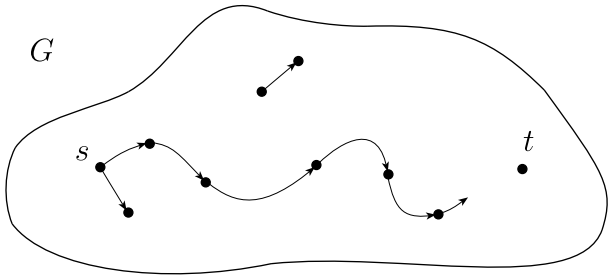


FIGURA 7.13
O problema *CAMINH*: Existe um caminho de s para t?

TEOREMA 7.14

CAMINH $\in$ P.

IDÉIA DA PROVA Provamos esse teorema apresentando um algoritmo de tempo polinomial que decide *CAMINH*. Antes de descrever esse algoritmo, vamos observar que um algoritmo de força-bruta para esse problema não é suficientemente rápido.

Um algoritmo de força-bruta para *CAMINH* procede examinando todos os caminhos potenciais em G e determinando se algum é um caminho direcionado de s para t. Um caminho potencial é uma seqüência de nós em G tendo um comprimento de no máximo m, onde m é o número de nós em G. (Se algum caminho direcionado existe de s para t, um tendo um comprimento de no máximo m existe porque repetir um nó nunca é necessário.) Mas o número de tais caminhos potenciais é aproximadamente m^m, o que é exponencial no número de nós em G. Por conseguinte, esse algoritmo de força-bruta usa tempo exponencial.

Para obter um algoritmo de tempo polinomial para *CAMINH* temos que fazer algo que evite a força bruta. Uma alternativa é usar um método de busca-em-grafo tal como busca-por-largura. Aqui, marcamos sucessivamente todos os nós em G que são atingíveis a partir de s por caminhos direcionados de comprimento 1, e então 2, e então 3, até m. Limitar o tempo de execução dessa estratégia por um polinômio é fácil.

PROVA Um algortimo de tempo polinomial M para *CAMINH* opera da seguinte forma.

$M =$ "Sobre a entrada $\langle G, s, t\rangle$ onde G é um grafo direcionado com nós s e t:

1. Ponha uma marca sobre o nó s.
2. Repita o seguinte até que nenhum nó adicional esteja marcado:
3. Faça uma varredura em todas as arestas de G. Se uma aresta (a, b) for encontrada indo de um nó marcado a para um nó

não marcado b, marque o nó b.

4. Se t estiver marcado, *aceite*. Caso contrário, *rejeite*."

Agora analisamos esse algoritmo para mostrar que ele roda em tempo polinomial. Obviamente, os estágios 1 e 4 são executados apenas uma vez. O estágio 3 roda no máximo m vezes porque cada vez exceto a última ele marca um nó adicional em G. Por conseguinte, o número total de estágios usados é no máximo $1 + 1 + m$, dando um tempo polinomial no tamanho de G.

Os estágios 1 e 4 de M são facilmente implementados em tempo polinomial em qualquer modelo determinístico razoável. O estágio 3 envolve uma varredura da entrada e um teste para ver se certos nós estão marcados, o que também é facilmente implementado em tempo polinomial. Logo, M é um algoritmo de tempo polinomial para *CAMINH*.

Vamos nos voltar para um outro exemplo de um algoritmo de tempo polinomial. Vamos dizer que dois números são ***primos entre si*** se 1 é o maior inteiro que divide ambos. Por exemplo, 10 e 21 são primos entre si, muito embora nenhum deles seja um número primo por si só, enquanto que 10 e 22 não são primos entre si porque ambos são divisíveis por 2. Seja *RELPRIME* o problema de se testar se dois números são primos entre si. Portanto

$$\textit{RELPRIME} = \{\langle x, y\rangle \mid x \text{ e } y \text{ são primos entre si}\}.$$

TEOREMA 7.15

RELPRIME $\in$ P.

IDÉIA DA PROVA Um algoritmo que resolve esse problema busca entre todos os possíveis divisores de ambos os números e aceita se nenhum deles é maior que 1. Entretanto, a magnitude de um número representado em binário, ou em qualquer outra notação na base k para $k \geq 2$, é exponencial no comprimento de sua representação. Conseqüentemente esse algoritmo de força-bruta busca entre um número exponencial de divisores potenciais e tem um tempo de execução exponencial.

Ao invés disso, resolvemos esse problema com um procedimento numérico antigo, chamado ***algoritmo euclideano***, para computar o máximo divisor comum. O ***máximo divisor comum*** de números naturais x e y, escrito $\gcd(x, y)$, é o maior inteiro que divide ambos x e y. Por exemplo, $\gcd(18, 24) = 6$. Obviamente que x e y são primos entre si sse $\gcd(x, y) = 1$. Descrevemos o algoritmo euclideano como algoritmo E na prova. Ele usa a função mod, onde $x \bmod y$ é o resto da divisão inteira de x por y.

PROVA O algoritmo euclideano E é como segue.

E = "Sobre a entrada $\langle x, y \rangle$, onde x e y são números naturais em binário:

1. Repita até que $y = 0$:
2. Atribua $x \leftarrow x \bmod y$.
3. Intercambie x e y.
4. Dê como saída x."

O algoritmo R resolve *RELPRIME*, usando E como uma subrotina.

R = "Sobre a entrada $\langle x, y \rangle$, onde x e y são números naturais em binário:

1. Rode E sobre $\langle x, y \rangle$.
2. Se o resultado for 1, *aceite*. Caso contrário, *rejeite*."

Claramente, se E roda corretamente em tempo polinomial, assim faz R e portanto somente precisamos analisar E com relação a tempo e corretude. A corretude desse algoritmo é bem conhecida portanto não mais a discutiremos aqui.

Para analisar a complexidade de tempo de E, primeiro mostramos que toda execução do estágio 2 (exceto possivelmente a primeira), corta o valor de x por no mínimo a metade. Após o estágio 2 ser executado, $x < y$ devido à natureza da função $\bmod$. Após o estágio 3, $x > y$ porque os dois números foram intercambiados. Daí, quando o estágio 2 for subseqüentemente executado, $x > y$. Se $x/2 \geq y$, então $x \bmod y < y \leq x/2$ e x cai no mínimo pela metade. Se $x/2 < y$, então $x \bmod y = x - y < x/2$ e x cai no mínimo pela metade.

Os valores de x e y são intercambiados toda vez que o estágio 3 é executado, portanto cada um dos valores originais de x e y são reduzidos no mínimo pela metade uma vez não e outra sim através do laço. Conseqüentemente o número máximo de vezes que os estágios 2 e 3 são executados é o menor entre $2 \log_2 x$ e $2 \log_2 y$. Esses logaritmos são proporcionais aos comprimentos das representações, dando o número de estágios executados como $O(n)$. Cada estágio de E usa somente tempo polinomial, portanto o tempo de execução total é polinomial.

O exemplo final de um algoritmo de tempo polinomial mostra que toda linguagem livre-do-contexto é decidível em tempo polinomial.

TEOREMA 7.16

Toda linguagem livre-do-contexto é um membro de P.

IDÉIA DA PROVA No Teorema 4.9 provamos que toda LLC é decidível. Para fazer isso demos um algoritmo para cada LLC que a decide. Se esse algoritmo roda em tempo polinomial, o teorema corrente segue como um corolário. Vamos relembrar aquele algoritmo e descobrir se ele roda suficientemente rápido.

Seja L uma LLC gerada por GLC G que está na forma normal de Chomsky. Do Problema 2.26, qualquer derivação de uma cadeia w tem $2n - 1$ passos, onde n é o comprimento de w porque G está na forma normal de Chomsky. O decisor

para L funciona tentando todas as derivações possíveis com $2n-1$ passos quando sua entrada é uma cadeia de comprimento n. Se alguma dessas for uma derivação de w, o decisor aceita; caso contrário, ele rejeita.

Uma análise rápida desse algoritmo mostra que ele não roda em tempo polinomial. O número de derivações com k passos pode ser exponencial em k, portanto esse algoritmo pode requerer tempo exponencial.

Para obter um algoritmo de tempo polinomial introduzimos uma técnica a poderosa chamada ***programação dinâmica***. Essa técnica usa a acumulação de informação sobre subproblemas menores para resolver problemas maiores. Guardamos a solução para qualquer subproblema de modo que precisamos resolvê-lo somente uma vez. Fazemos isso montando uma tabela de todos os subproblemas e entrando com suas soluções sistematicamente à medida que as encontramos.

Nesse caso, consideramos os subproblemas de se determinar se cada variável em G gera cada subcadeia de w. O algoritmo entra com a solução para subproblema numa tabela $n \times n$. Para $i \leq j$ a (i, j)-ésima entrada da tabela contém a coleção de variáveis que geram a subcadeia $w_i w_{i+1} \cdots w_j$. Para $i > j$ as entradas na tabela não são usadas.

O algoritmo preenche as entradas na tabela para cada subcadeia de w. Primeiro ele preenche as entradas para as subcadeias de comprimento 1, então aquelas de comprimento 2, e assim por diante. Ele usa as entradas para comprimentos mais curtos para assistir na determinação das entradas para comprimentos mais longos.

Por exemplo, suponha que o algoritmo já tenha determinado quais variáveis geram todas as subcadeias até o comprimento k. Para determinar se uma variável A gera uma subcadeia específica de comprimento $k+1$ o algoritmo divide aquela subcadeia em duas partes não vazias nas k maneiras possíveis. Para cada divisão, o algoritmo examina cada regra $A \rightarrow BC$ para determinar se B gera a primeira parte e C gera a segunda parte, usando entradas previamente computadas na tabela. Se ambas B e C geram as partes respectivas, A gera a subcadeia e portanto é adicionada à entrada associada na tabela. O algoritmo inicia o processo com as cadeias de comprimento 1 examinando a tabela para as regras $A \rightarrow \mathtt{b}$.

PROVA O seguinte algoritmo D implementa a idéia da prova. Seja G uma GLC na forma normal de Chomsky gerando a LLC L. Assuma que S seja a variável inicial. (Lembre-se de que a cadeia vazia é trabalhada de forma especial numa gramática na forma normal de Chomsky. O algoritmo lida com o caso especial no qual $w = \varepsilon$ no estágio 1.) Os comentários aparecem dentro de parênteses duplos.

$D =$ "Sobre a entrada $w = w_1 \cdots w_n$:

1. Se $w = \varepsilon$ e $S \rightarrow \varepsilon$ for uma regra, *aceite*. [[handle $w = \varepsilon$ case]]
2. Para $i = 1$ até n: [[examine each substring of length 1]]
3. Para cada variável A:

4. Teste se $A \rightarrow \mathtt{b}$ é uma regra, onde $\mathtt{b} = w_i$.
5. Se for, coloque A em $table(i, i)$.
6. Para $l = 2$ até n: ⟦l is the length of the substring⟧
7. Para $i = 1$ até $n - l + 1$: ⟦i is the start position of the substring⟧
8. Faça $j = i + l - 1$, ⟦j is the end position of the substring⟧
9. Para $k = i$ até $j - 1$: ⟦k is the split position⟧
10. Para cada regra $A \rightarrow BC$:
11. Se $table(i, k)$ contém B e $table(k + 1, j)$ contém C, ponha A em $table(i, j)$.
12. Se S estiver em $table(1, n)$, *aceite*. Caso contrário, *rejeite*."

Agora analisamos D. Cada estágio é facilmente implementado para rodar em tempo polinomial. Os estágios 4 e 5 rodam no máximo nv vezes, onde v é o número de variáveis em G e é uma constante fixa independente de n; logo, esses estágios rodam $O(n)$ vezes. O estágio 6 roda no máximo n vezes. Cada vez que o estágio 6 roda, o estágio 7 roda no máximo n vezes. Cada vez que o estágio 7 roda, os estágios 8 e 9 rodam no máximo n vezes. Cada vez que o estágio 9 roda, o estágio 10 roda r vezes, onde r é o número de regras de G e é uma outra constante fixa. Portanto, o estágio 11, o laço mais interno do algoritmo, roda $O(n^3)$ vezes. Somando o total mostra que D executa $O(n^3)$ estágios.

7.3

A CLASSE NP

Como observamos na Seção 7.2, podemos evitar a busca de força-bruta em muitos problemas e obter soluções de tempo polinomial. Entretanto, tentativas de evitar a força bruta em alguns outros problemas, incluindo muitos interessantes e úteis, não têm sido bem sucedidas, e algoritmos de tempo polinomial que os resolvem não se sabe se existem.

Por que não temos tido sucesso em encontrar algoritmos de tempo polinomial para esses problemas? Não sabemos a resposta para essa importante questão. Talvez esses problemas tenham algoritmos de tempo polinomial que ainda não tenham sido descobertos, e que se baseiem em princípios desconhecidos. Ou possivelmente alguns desses problemas simplesmente *não podem* ser resolvidos em tempo polinomial. Eles podem ser intrinsecamente difíceis.

Uma descoberta notável concernente a essa questão mostra que as complexidades de muitos problemas estão interligadas. Um algoritmo de tempo polinomial para um desses problemas pode ser usado para resolver uma classe inteira de problemas. Para entender esse fenômeno, vamos começar com um exemplo.

Um ***caminho hamiltoniano*** em um grafo direcionado G é um caminho direci-

onado que passa por cada nó exatamente uma vez. Consideramos o problema de se testar se um grafo direcionado contém um caminho hamiltoniano conectando dois nós especificados, como mostrado na figura abaixo. Seja

$$\textit{HAMPATH} = \{\langle G, s, t\rangle \mid G \text{ é um grafo direcionado com um caminho hamiltoniano de } s \text{ para } t\}.$$

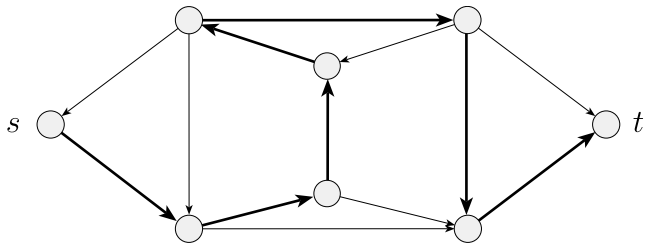


FIGURA 7.17
Um caminho hamiltoniano passa por todo nó exatamente uma vez

Podemos facilmente obter um algoritmo de tempo exponencial para o problema *HAMPATH* modificando o algoritmo de força-bruta para *CAMINH* dado no Teorema 7.14. Precisamos apenas adicionar um teste para verificar que o caminho potencial é hamiltoniano. Ninguém sabe se *HAMPATH* é solúvel em tempo polinomial.

O problema *HAMPATH* de fato tem uma característica chamada ***verificabilidade polinomial*** que é importante para entender sua complexidade. Muito embora não conheçamos uma forma rápida (i.e., de tempo polinomial) de determinar se um grafo contém um caminho hamiltoniano, se tal caminho fosse descoberto de alguma forma (talvez usando o algoritmo de tempo exponencial), poderímos facilmente convencer uma outra pessoa de sua existência, simplesmente apresentando-o. Em outras palavras, *verificar* a existência de um caminho hamiltoniano pode ser muito mais fácil que *determinar* sua existência.

Um outro problema polinomialmente verificável é compostura. Lembre-se de que um número natural é ***composto*** se ele é o produto de dois inteiros maiores que 1 (i.e., um número composto é aquele que não é um número primo). Seja

$$\textit{COMPOSITES} = \{x \mid x = pq, \text{ para inteiros } p, q > 1\}.$$

Podemos facilmente verificar que um número é composto—tudo o que é necessário é um divisor desse número. Recentemente, um algoritmo de tempo polinomial para testar se um númer é primo ou composto foi descoberto, mas ele é consideravelmente mais complicado que o método precedente para verificar compostura.

Alguns problemas podem não ser polinomialmente verificáveis. Por exem-

plo, tome $\overline{HAMPATH}$, o complemento do problema *HAMPATH*. Mesmo se pudéssemos determinar (de alguma forma) se um grafo realmente *não* tivesse um caminho hamiltoniano, não conhecemos uma maneira de permitir a uma outra pessoa verificar sua não existência sem usar o mesmo algoritmo de tempo exponencial para fazer a determinação primeiramente. Uma definição formal segue.

DEFINIÇÃO 7.18

Um ***verificador*** para uma linguagem A é um algoritmo V, onde

$$A = \{w | \ V \text{ aceita } \langle w, c\rangle \text{ para alguma cadeia } c\}.$$

Medimos o tempo de um verificador somente em termos do comprimento de w, portanto um ***verificador de tempo polinomial*** roda em tempo polinomial no comprimento de w. Uma linguagem A é ***polinomialmente verificável*** se ela tem um verificador de tempo polinomial.

Um verificador usa informaçã adicional, representada pelo símbolo c na Definição 7.18, para verificar que uma cadeia w é um membro de A. Essa informação é chamada ***certificado***, ou ***prova***, da pertinência a A. Observe que, para verificadores polinomiais, o certificado tem comprimento polinomial (no comprimento de w) porque isso é tudo que o verificador pode acessar no seu limitante de tempo. Vamoss aplicar essa definição às linguagens *HAMPATH* e *COMPOSITES*.

Para o problema *HAMPATH* problem, um certificado para uma cadeia $\langle G, s, t\rangle \in$ *HAMPATH* é simplesmente o caminho hamiltoniano de s a t. Para o problema *COMPOSITES* problem, um certificado para o número composto x é simplesmente um de seus divisores. Em ambos os casos o verificador pode checar em tempo polinomial que a entrada está na linguagem quando ela recebe o certificado.

DEFINIÇÃO 7.19

NP é a classe de linguagens que têm verificadores de tempo polinomial.

A classe NP é importante porque ela contém muitos problemas de interesse prático. Da discussão precedente, ambos *HAMPATH* e *COMPOSITES* são membros de NP. Como mencionamos, *COMPOSITES* é também um membro de P que é um subconjunto de NP, mas provar esse resultado mais forte é muito mais difícil. O termo NP vem de ***tempo polinomial não-determinístico*** e é derivado de uma caracterização alternativa usando máquinas de Turing não-determinísticas de tempo polinomial. Problemas em NP são às vezes chamados problemas NP.

A seguir está uma máquina de Turing não-determinística (MTN) que decide o problema *HAMPATH* problem em tempo polinomial não-determinístico. Lembre-se de que na Definição 7.9 especificamos o tempo de uma máquina não-determinística como sendo o tempo usado pelo ramo de computação mais longo.

N_1 = "Sobre a entrada $\langle G, s, t\rangle$, onde G é um grafo direcionado com nós s e t:

1. Escreva uma lista de m números, $p_1, \ldots, p_m$, onde m é o número de nós em G. Cada número na lista é selecionado não-deterministicamente sendo entre 1 e m.
2. Verifique se há repetições na lista. Se alguma for encontrada, *rejeite*.
3. Verifique se $s = p_1$ e $t = p_m$. Se algum falhar, *rejeite*.
4. Para cada i entre 1 e $m - 1$, verifique se (p_i, p_{i+1}) é uma aresta de G. Se alguma não for, *rejeite*. Caso contrário, todos os testes foram positivos, portanto *aceite*."

Para analisar esse algoritmo e verificar que ele roda em tempo polinomial não-determinístico, examinamos cada um de seus estágios. No estágio 1, a escolha não-determinística claramente roda em tempo polinomial. Nos estágios 2 e 3, cada parte é uma simples verificação, portanto juntos eles rodam em tempo polinomial. Finalmente, o estágio 4 também claramente roda em tempo polinomial. Por conseguinte, esse algoritmo roda em tempo polinomial não-determinístico.

TEOREMA 7.20

Uma linguagem está em NP sse ela é decidida por alguma máquina de Turing não-determinística de tempo polinomial.

IDÉIA DA PROVA Mostramos como converter um verificador de tempo polinomial para uma MTN de tempo polinomial equivalente e vice versa. A MTN simula o verificador adivinhando o certificado. O verificador simula a MTN usando o ramo de computação de aceitação como o certificado.

PROVA Para a direção para a frente desse teorema, suponha que $A \in$ NP e mostre que A é decidida por uma MTN de tempo polinomial N. Seja V o verificador de tempo polinomial para A que existe pela definição de NP. Assuma que V seja uma MT que roda em tempo n^k e construa N da seguinte maneira.

N = "Sobre a entrada w de comprimento n:

1. Não-deterministicamente selecione uma cadeia c de comprimento no máximo n^k.
2. Rode V sobre a entrada $\langle w, c\rangle$.
3. Se V aceita, *aceite*; caso contrário, *rejeite*."

Para provar a outra direção do teorema, assuma que A seja decidida por uma MTN de tempo polinomial N e construa um verificador de tempo polinomial V da seguinte maneira.

V = "Sobre a entrada $\langle w, c \rangle$, onde w e c são cadeias:

1. Simule N sobre a entrada w, tratar cada símbolo de c como uma descrição da escolha não-determinística a fazer a cada passo (como na prova do Teorema 3.16).
2. Se esse ramo da computação de N aceita, *aceite*; caso contrário, *rejeite*."

Definimos a classe de complexidade de tempo não-determinístico $\mathrm{NTIME}(t(n))$ como análoga à classe de complexidade de tempo determinístico $\mathrm{TIME}(t(n))$.

DEFINIÇÃO 7.21

NTIME*(t(n))* $= \{L \mid L$ é uma linguagem decidida por uma máquina de Turing não-determinística de tempo $O(t(n))\}$.

COROLÁRIO 7.22

$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$.

A classe NP é insensível à escolha do modelo computacional não-determinístico razoável porque todos esses modelos são polinomialmente equivalentes. Ao descrever e analisar algoritmos de tempo polinomial não-determinísticos, seguimos as convenções precedentes para algoritmos determinísticos de tempo polinomial. Cada estágio de um algoritmo não-determinístico de tempo polinomial deve ter uma implementação óbvia em tempo polinomial não-determinístico em um modelo computacional não-determinístico razoável. Analisamos o algoritmo para mostrar que todo ramo usa no máximo uma quantidade polinomial de estágios.

EXEMPLOS DE PROBLEMAS EM NP

Um ***clique*** em um grafo não-direcionado é um subgrafo, no qual todo par de nós está conectado por uma aresta. Um ***k-clique*** é um clique que contém k nós. A Figura 7.23 ilustra um grafo tendo um 5-clique.

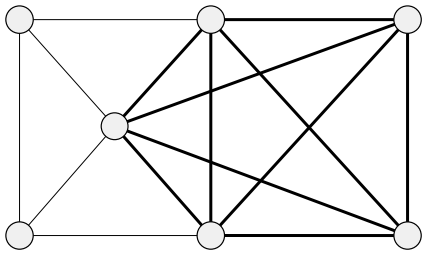


FIGURA 7.23
Um grafo com um 5-clique

O problema do clique é determinar se um grafo contém um clique de um tamanho especificado. Seja

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ é um grafo não-direcionado com um } k\text{-clique}\}.$$

TEOREMA 7.24

CLIQUE está em NP.

IDÉIA DA PROVA O clique é o certificado.

PROVA Aqui está um verificador V para *CLIQUE*.

V = "Sobre a entrada $\langle\langle G, k\rangle, c\rangle$:

1. Teste se c é um conjunto de k nós em G
2. Teste se G contém todas as arestas conectando nós em c.
3. Se ambos os testes retornam positivo, *aceite*; caso contrário, *rejeite*."

PROVA ALTERNATIVA Se você preferir pensar em NP em termos de máquinas de Turing não-determinísticas de tempo polinomial, você pode provar esse teorema fornecendo uma que decida *CLIQUE*. Observe a similaridade entre as duas provas.

N = "Sobre a entrada $\langle G, k\rangle$, onde G é um grafo:

1. Não-deterministicamente selecione um subconjunto c de k nós de G.
2. Teste se G contém todas as arestas conectando nós em c.
3. Se sim, *aceite*; caso contrário, *rejeite*."

A seguir consideramos o problema *SUBSET-SUM* concernente a aritmética

de inteiros. Nesse problema temos uma coleção de números $x_1, \ldots, x_k$ e um número alvo t. Desejamos determinar se a coleção contém uma subcoleção que soma t. Por conseguinte,

$$\textit{SUBSET-SUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ e para algum } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ temos } \Sigma y_i = t\}.$$

Por exemplo, $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ *SUBSET-SUM* porque $4 + 21 = 25$. Note que $\{x_1, \ldots, x_k\}$ e $\{y_1, \ldots, y_l\}$ são considerados como ***multiconjuntos*** e portanto permitem repetição de elementos.

TEOREMA 7.25

SUBSET-SUM está em NP.

IDÉIA DA PROVA O subconjunto é o certificado.

PROVA O que segue é um verificador V para *SUBSET-SUM*.

$V =$ "Sobre a entrada $\langle\langle S, t\rangle, c\rangle$:

1. Teste se c é uma coleção de números que somam t.
2. Teste se S contém todos os números em c.
3. Se ambos os testes retornem positivo, *aceite*; caso contrário, *rejeite*."

PROVA ALTERNATIVA Podemos também provar esse teorema dando uma máquina de Turing não-determinística de tempo polinomial para *SUBSET-SUM* da seguinte forma.

$N =$ "Sobre a entrada $\langle S, t\rangle$:

1. Não-deterministicamente selecione um subconjunto c dos números em S.
2. Teste se c é uma coleção de números que somam t.
3. Se o teste der positivo, *aceite*; caso contrário, *rejeite*."

Observe que os complementos desses conjntos, $\overline{\textit{CLIQUE}}$ e $\overline{\textit{SUBSET-SUM}}$, não são obviamente membros de NP. Verificar que algo *não* está presente parece ser mais difícil que verificar que *está* presente. Fazemos uma classe de complexidade separada, chamada coNP, que contém as linguagens que são complementos de linguagens em NP. Não sabemos se coNP é diferente de NP.

A QUESTÃ P VERSUS NP

Como temos insistido, NP é a classe de linguagens que são solúveis em tempo polinomial numa máquina de Turing não-determinística, ou, equivalentemente,

ela é a classe de linguagens nas quais pertinência na linguagem pode ser verificada em tempo polinomial. P é a classe de linguagens onde pertinência pode testada em tempo polinomial. Resuma essa informação da seguinte forma, onde nos referimos frouxamente a solúvel em tempo polinomial como solúvel "rapidamente."

P = a classe de linguagens para as quais pertinência pode ser *decidida* rapidamente.

NP = a classe de linguagens para as quais pertinência pode ser *verificada* rapidamente.

Apresentamos exemplos de linguagens, tais como *HAMPATH* e *CLIQUE*, que são membros de NP mas que não se sabe se estão em P. O poder de verificabilidade polinomial parece ser muito maior que aquele da decidibilidade polinomial. Mas, por mais difícil que seja de imaginar, P e NP poderiam ser iguais. Somos incapazes de *provar* a existência de uma única linguagem em NP que não esteja em P.

A questão de se P = NP é um dos maiores problemas não resolvidos em ciência da computação teórica e matemática contemporânea. Se essas classes fossem iguais, qualquer problema polinomiamente verificável ser polinomialmente decidível. A maioria dos pesquisadores acreditam que as duas classes não são iguais porque as pessoas investiram esforços enormes para encontrar algoritmos de tempo polinomial para certos problemas em NP, se sucesso. Pesquisadores também têm tentado provar que as classes são diferentes, mas isso acarretaria mostrar que nenhum algoritmo rápido existe para substituir a força-bruta. Fazer isso está atualmente além do alcance científico. A seguinte figura mostra as duas possibilidades.

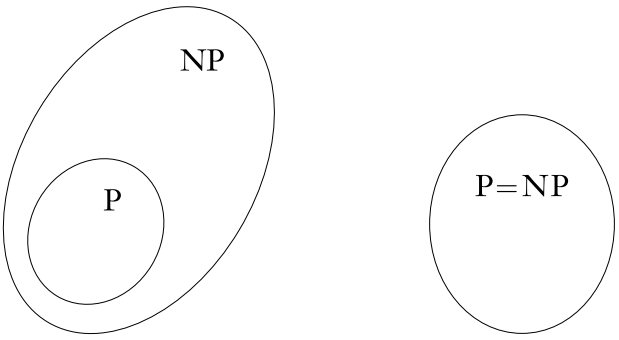


FIGURA 7.26
Uma dessas possibilidades é correta

O melhor método conhecido para resolver linguagens em NP deterministicamente usa tempo exponencial. Em outras palavras, podemos provar que

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k}),$$

mas não sabemos se NP está contida em uma classe de complexidade de tempo determinístico menor.

7.4 NP-COMPLETETUDE

Um avanço importante na questão P versus NP veio no início dos anos 1970s com o trabalho de Stephen Cook e Leonid Levin. Eles descobriram certos problemas em NP cuja complexidade individual está relacionada àquela da classe inteira. Se um algoritmo de tempo polinomial existe para quaisquer desses problemas, todos os problemas em NP seriam solúveis em tempo polinomial. Esses problemas são chamados ***NP-completos***. O fenômeno da NP-completude é importante por razões tanto teóricas quanto práticas.

No lado teóretico, um pesquisador tentando mostrar que P é diferente de NP pode focar sobre um problema NP-completo. Se algum problema em NP requer mais que tempo polinomial, um NP-completo também requer. Além disso, um pesquisador tentando provar que P é igual a NP somente precisa encontrar um algoritmo de tempo polinomial para um problema NP-completo para atingir seu objetivo.

No lado prático, o fenômeno da NP-completude pode evitar que se desperdice tempo buscando por um algoritmo de tempo polinomial não existente para resolver um problema específico. Muito embora possamos não ter a matemática necessária para provar que o problema é insolúvel em tempo polinomial, acreditamos que P é diferente de NP, portanto provar que um problema é NP-completo é forte evidência de sua não-polinomialidade.

O primeiro problema NP-completo que apresentamos é chamado ***problema da satisfatibilidade***. Lembre-se de que variáveis que podem tomar os valores VERDADEIRO FALSO são chamadas ***variáveis booleanas*** (veja a Seção 0.2). Usualmente, representamos VERDADEIRO por 1 e FALSO por 0. As ***operações booleanas*** E, OUe NÃO, representadas pelos símbolos $\wedge$, $\vee$e $\neg$, respectivamente, são descritos na lista seguinte. Usamos a barra superior como uma abreviação para o símbolo $\neg$, portanto $\overline{x}$ significa $\neg\, x$.

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \overline{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

Uma ***fórmula booleana*** é uma expressão envolvendo variáveis booleanas e operações. Por exemplo,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

é uma fórmula booleana. Uma fórmula booleana é ***satisfatível*** se alguma atribuição de 0s e 1s às variáveis faz a fórmula ter valor 1. A fórmula prece-

dente é satisfatível porque a atribuição $x = 0$, $y = 1$ e $z = 0$ faz ϕ ter valor 1. Dizemos que a atribuição *satisfaz* ϕ. O ***problema da satisfatibilidade*** é testar se uma fórmula booleana é satisfatível. Seja

$$SAT = \{\langle\phi\rangle \mid \phi \text{ é uma fórmula booleana satisfatível}\}.$$

Agora enunciamos o teorema de Cook–Levin, que relaciona a complexidade do problema *SAT* às complexidades de todos os problemas em NP.

TEOREMA 7.27

Teorema de Cook–Levin $SAT \in \mathrm{P}$ sse $\mathrm{P} = \mathrm{NP}$.

A seguir, desenvolvemos o método que é central para a prova do teorema de Cook–Levin.

REDUTIBILIDADE EM TEMPO POLINOMIAL

No Capítulo 5 definimos o conceito de reduzir um problema a um outro. Quando o problema A se reduz ao problema B, uma solução para B pode ser usada para resolver A. Agora definimos uma versão da redutibilidade que leva em consideração a eficiência da computação. Quando o problema A é *eficientemente* redutível ao problema B, uma solução eficiente para B pode ser usada para resolver A eficientemente.

DEFINIÇÃO 7.28

Uma função $f\colon \Sigma^* \longrightarrow \Sigma^*$ é uma ***função computável em tempo polinomial*** se alguma máquina de Turing de tempo polinomial M existe que pára com exatamente $f(w)$ na sua fita, quando iniciada sobre qualquer entrada w.

DEFINIÇÃO 7.29

A linguagem A é ***redutível por mapeamento em tempo polinomial***,[1] ou simplesmente ***redutível em tempo polinomial***, à linguagem B, em símbolos $A \leq_{\mathrm{P}} B$, se uma função computável em tempo polinomial $f\colon \Sigma^* \longrightarrow \Sigma^*$ existe, onde para toda w,

$$w \in A \iff f(w) \in B.$$

A função f é chamada ***redução de tempo polinomial*** de A para B.

A redutibilidade de tempo polinomial é a análoga eficiente à redutibilidade por mapeamento como definida na Seção 5.3. Outras formas de redutibilidade eficiente estão disponíveis, mas redutibilidade de tempo polinomial é uma forma simples que é adequada para nossos propósitos portanto não discutiremos os outros aqui. A figura abaixo ilustra a redutibilidade de tempo polynomial.

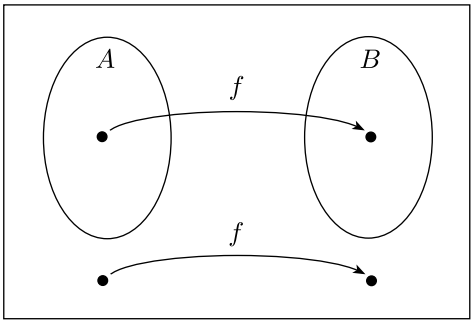


FIGURA 7.30
A função de tempo polinomial f reduzindo A a B

Da mesma forma que com uma redução por mapeamento comum, uma redução de tempo polinomial de A para B provê uma maneira converter o teste de pertinência em A para o teste de pertinência em B, mas agora a conversão é feita eficientemente. Para testar se $w \in A$, usamos a redução f para mapear w para $f(w)$ e testar se $f(w) \in B$.

Se uma linguagem for redutível em tempo polinomial a uma linguagem já sabidamente solúvel em tempo polinomial, obtemos uma solução polinomial para a linguagem original, como no teorema seguinte.

TEOREMA 7.31

Se $A \leq_{\mathrm{P}} B$ e $B \in \mathrm{P}$, então $A \in \mathrm{P}$.

PROVA Seja M o algoritmo de tempo polinomial que decide B e f a redução de tempo polinomial de A para B. Descrevemos um algoritmo de tempo polinomial N que decide A da seguinte forma.

$N =$ "Sobre a entrada w:

1. Compute $f(w)$.

[1]Ela é chamada ***redutibilidade de tempo polinomial muitos–para–um*** em alguns outros livros-texto.

2. Rode M sobre a entrada $f(w)$ e dê como saída o que quer que M dê como saída."

Temos $w \in A$ sempre que $f(w) \in B$ porque f é uma redução de A para B. Por conseguinte, M aceita $f(w)$ sempre que $w \in A$. Além do mais, N roda em tempo polinomial porque cada um de seus dois estágios roda em tempo polinomial. Note que o estágio 2 roda em tempo polinomial porque a composição de polinômios é um polinômio.

Antes de exibir uma redução de tempo polinomial introduzimos *3SAT*, um caso especial do problema da satisfatibilidade no qual todas as fórmulas estão numa forma especial. Um ***literal*** é uma variável booleana ou uma variável booleana negada, como em x ou $\overline{x}$. Uma ***cláusula*** é uma fórmula composta de vários literais conectados por $\vee$s, como em $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. Uma fórmula booleana está na ***forma normal conjuntiva***, chamada ***fnc-fórmula***, se ela compreende várias cláusulas conectadas por $\wedge$s, como em

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

Ela é uma ***3fnc-fórmula*** se todas as cláusulas tiverem três literais, como em

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Seja $\mathit{3SAT} = \{\langle\phi\rangle \mid \phi$ é uma 3fnc-fórmula satisfatível$\}$. Em uma fnc-fórmula satisfatível, cada cláusula deve conter pelo menos um literal que é atribuído o valor 1.

O teorema seguinte apresenta uma redução de tempo polinomial do problema *3SAT* para o problema *CLIQUE*.

TEOREMA 7.32

3SAT é redutível em tempo polinomial a *CLIQUE*.

IDÉIA DA PROVA A redução de tempo polinomial f que mostramos de *3SAT* para *CLIQUE* converte fórmulas para grafos. Nos grafos construídos, cliques de um dado tamanho correspondem a atribuições satisfeitoras da fórmula. Estruturas dentro do grafo são projetadas para imitar o comportamento das variáveis e cláusulas.

PROVA Seja ϕ uma fórmula com k cláusulas tal como

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \quad \cdots \quad \wedge (a_k \vee b_k \vee c_k).$$

A redução f gera a cadeia $\langle G, k\rangle$, onde G é um grafo não-direcionado definido da seguinte forma.

Os nós em G são organizados em k grupos de três nós cada um chamado de ***tripla***, $t_1, \ldots, t_k$. Cada tripla corresponde a uma das cláusulas em ϕ, e cada nó em uma tripla corresponda a um literal na cláusula associada. Rotule cada nó de G com seu literal correspondente em ϕ.

As arestas de G conectam todos exceto dois tipos de pares de nós em G. Nenhuma aresta está presente entre nós na mesma tripla e nenhuma aresta está presente entre dois nós com rótulos contraditórios, como em x_2 e $\overline{x_2}$. A figura abaixo ilustra essa construção quando $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

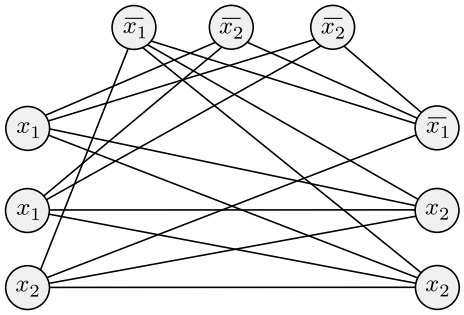


FIGURA 7.33
O grafo que a redução produz de
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Agora demonstramos por que essa construção funciona. Mostramos que ϕ é satisfatível sse G tem um k-clique.

Suponha que ϕ tem uma atribuição satisfeitora. Nessa atribuição satisfeitora, pelo menos um literal é verdadeiro em toda cláusula. Em cada tripla de G, selecionamos um nó correspondendo a um literal verdadeiro na atribuição satisfeitora. Se mais que um literal for verdadeiro em uma cláusula específica, escolhemos um dos literais arbitrariamente. Os nós que acabam de ser selecionados formam um k-clique. O número de nós selecionado é k, porque escolhemos um para cada uma das k triplas. Cada par de nós selecionados é ligado por uma aresta porque nenhum par se encaixa em uma das exceções descritas anteriormente. Eles não poderiam ser da mesma tripla porque selecionamos somente um nó por tripla. Eles não poderiam ter rótulos contraditórios porque os literais associados eram ambos verdadeiros na atribuição satisfeitora. Por conseguinte, G contém um k-clique.

Suponha que G tenha um k-clique. Nenhum par de nós do clique ocorre na mesma tripla porque nós na mesma tripla não são conectados por arestas. Conseqüentemente, cada uma das k triplas contém exatamente um dos k nós do clique. Atribuímos valores-verdade às variáveis de ϕ de modo que cada literal

que rotula um nó do clique torna-se verdadeiro. Fazer isso é sempre possível porque dois nós rotulados de uma maneira contraditória não são conectados por uma aresta e portanto ambos não podem estar no clique. Essa atribuição às variáveis satisfaz ϕ porque cada tripla contém um nó do clique e portanto cada cláusula contém um literal ao qual é atribuído VERDADEIRO. Por conseguinte, ϕ é satisfatível.

Os Teoremas 7.31 e 7.32 nos dizem que, se *CLIQUE* for solúvel em tempo polinomial, o mesmo acontece com *3SAT*. À primeira vista, essa conexão entre esses dois problemas parece um tanto impressionante porque, superficialmente, eles são bastante diferentes. Mas a redutibilidade de tempo polinomial nos permite relacionar suas complexidades. Agora nos voltamos para uma definição que nos permitirá similarmente relacionar as complexidades de um classe inteira de problemas.

DEFINIÇÃO DE NP-COMPLETUDE

DEFINIÇÃO 7.34

Uma linguagem B é ***NP-completa*** se ela satisfaz duas condições:

1. B está em NP, e
2. toda A em NP é redutível em tempo polinomial a B.

TEOREMA 7.35

Se B for NP-completa e $B \in \mathrm{P}$, então $\mathrm{P} = \mathrm{NP}$.

PROVA Esse teorema segue diretamente da definição de redutibilidade de tempo polinomial.

TEOREMA 7.36

Se B for NP-completa e $B \leq_{\mathrm{P}} C$ para C in NP, então C é NP-completa.

PROVA Já sabemos que C está em NP, portanto devemos mostrar que toda A em NP é redutível em tempo polinomial a C. Em razão de B ser NP-completa, toda linguagem em NP é redutível em tempo polinomial a B, e B por sua vez é redutível em tempo polinomial a C. Reduções em tempo polinomial se compõem; ou seja, se A for redutível em tempo polinomial a B e B for redutível

em tempo polinomial a C, então A é redutível em tempo polinomial a C. Logo toda linguagem em NP é redutível em tempo polinomial a C.

O TEOREMA DE COOK–LEVIN

Uma vez que temos um problema NP-completo, podemos obter outros por redução de tempo polinomial a partir dele. Entretanto, estabelecer o primeiro problema NP-completo é mais difícil. Agora fazemos isso provando que *SAT* é NP-completo.

TEOREMA 7.37

SAT é NP-completo.[2]

Esse teorema re-enuncia o Teorema 7.27, o teorema de Cook–Levin, em uma outra forma.

IDÉIA DA PROVA Mostrar que *SAT* está em NP é fácil, e fazemos isso em breve. A parte difícil da prova é mostrar que qualquer linguagem em NP é redutível em polinomial time a *SAT*.

Para fazer isso construimos uma redução de tempo polinomial para cada linguagem A em NP para *SAT*. A redução para A toma uma cadeia w e produz uma fórmula boolean ϕ que simula a máquina NP para A sobre a entrada w. Se a máquina aceita, ϕ tem uma atribuição satisfeitora que corresponde à computação de aceitação. Se a máquina não aceita, nenhuma atribuição satisfaz ϕ. Conseqüentemente, w está em A se e somente se ϕ é satisfatível.

Construir verdadeiramente a redução para funcionar dessa maneira é uma tarefa conceitualmente simples, embora devamos ser capazes de lidar com muitos detalhes. Uma fórmula booleana pode conter as operações booleanas E, OU e NÃO, e essas operações formam a base para a circuitaria usada em computadores eletrônicos. Logo, o fato de que podemos projetar uma fórmula booleana para simular uma máquina de Turing não é surpreendente. Os detalhes estão na implementação dessa idéia.

PROVA Primeiro, mostramos que *SAT* está em NP. Uma máquina de tempo polinomial não-determinístico pode adivinhar uma atribuição para uma dada fórmula ϕ e aceitar se a atribuição satisfaz ϕ.

A seguir, tomamos qualquer linguagem A em NP e mostramos que A é redutível em tempo polinomial a *SAT*. Seja N uma máquina de Turing não-determinística que decide A em tempo n^k para alguma constante k. (Por conveniência assumimos, na verdade, que N roda em tempo $n^k - 3$, mas apenas aque-

[2]Uma prova alternativa desse teorema aparece na Seção 9.3 na página 375.

les leitores interessados em detalhes deveriam se preocupar com essa questão menor.) A seguinte noção ajuda a descrever a redução.

Um ***tableau*** para N sobre w é uma tabela $n^k \times n^k$ cujas linhas são as configurações de um ramo da computação de N sobre a entrada w, como mostado na Figura 7.38.

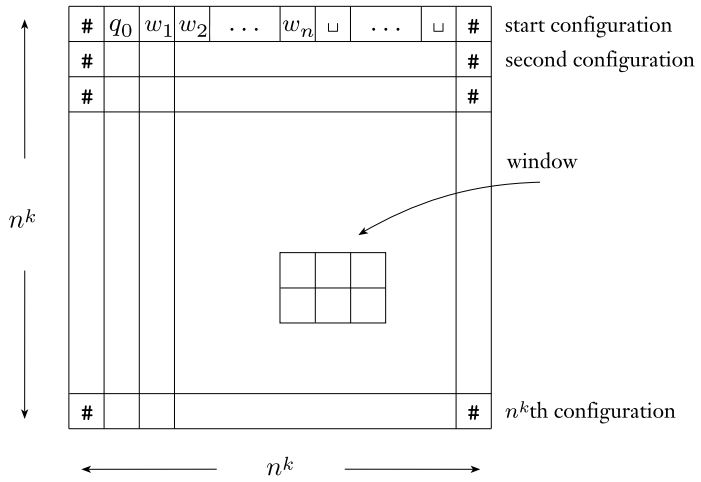


FIGURA 7.38
Um tableau é uma tabela $n^k \times n^k$ de configurações

Por conveniência mais adiante assumimos que cada configuração começa e termina com um símbolo #, de modo que a primeira e a última colunas de um tableau são todas de #s. A primeira linha do tableau é a configuração inicial de N sobre w, e cada linha segue da anterior conforme a função de transição de N. Um tableau é de ***aceitação*** se qualquer linha do tableau for uma configuração de aceitação.

Todo tableau de aceitação para N sobre w corresponde a um ramo de computação de aceitação de N sobre w. Portanto, o problema de se determinar se N aceita w é equivalente ao problema de se determinar se um tableau de aceitação para N sobre w existe.

Agora chegamos à descrição da redução em tempo polinomial f de A para *SAT*. Sobre a entrada w, a redução produz uma fórmula ϕ. Começamos descrevendo as variáveis de ϕ. Digamos que Q e Γ sejam o conjunto de estados e o alfabeto de fita de N. Seja $C = Q \cup \Gamma \cup \{\#\}$. Para cada i e j entre 1 e n^k e para cada s em C temos uma variável, $x_{i,j,s}$.

Cada uma das $(n^k)^2$ entradas de um tableau é chamada ***célula***. A célula na linha i e coluna j é chamada $cell[i, j]$ e contém um símbolo de C. Representamos o conteúdo das células com as variáveis de ϕ. Se $x_{i,j,s}$ toma o valor 1, isso significa que $cell[i, j]$ contém um s.

Agora projetamos ϕ de modo que uma atribuição satisfeitora às variáveis re-

almente corresponda a um tableau de aceitação para N sobre w. A fórmula ϕ é o E de quatro partes $\phi_{\text{celula}} \wedge \phi_{\text{inicio}} \wedge \phi_{\text{movemento}} \wedge \phi_{\text{aceita}}$. Descrevemos cada parte por vez.

Como mencionamos anteriormente, ligar a varivél $x_{i,j,s}$ corresponde a colocar o símbolo s na $cell[i,j]$. A primeira coisa que devemos garantir de modo a obter uma correspondência entre uma atribuição e um tableau é que a atribuição liga exatamente uma variável para cada célula. A fórmula ϕ_{celula} garante esse requisito expressando-o em termos de operações booleanas:

$$\phi_{\text{celula}} = \bigwedge_{1 \le i,j \le n^k} \left[\Big(\bigvee_{s \in C} x_{i,j,s} \Big) \wedge \Big(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \Big) \right].$$

Os símbolos $\bigwedge$ e $\bigvee$ significam E e OU iterados. Por exemplo, a expressão na fórmula precedente

$$\bigvee_{s \in C} x_{i,j,s}$$

é uma abreviação para

$$x_{i,j,s_1} \vee x_{i,j,s_2} \vee \cdots \vee x_{i,j,s_l}$$

onde $C = \{s_1, s_2, \ldots, s_l\}$. Logo, ϕ_{celula} é na realidade uma expressão grande que contém um fragmento para cada célula no tableau porque i e j variam de 1 a n^k. A primeira parte de cada fragmento diz que pelo menos uma variável é ligada na célula correspondente. A segunda parte de cada fragmento diz que não mais que uma variável é ligada (literalmente, ela diz que em cada par de variáveis, pelo menos uma é desligada) na célula correspondente. Esses fragmentos são conectados por operações $\wedge$.

A primeira parte de ϕ_{celula} dentro dos parênteses estipula que pelo menos uma variável que está associada a cada célula está ligada, enquanto que a segunda parte estipula que não mais que uma variável está ligada para cada célula. Qualquer atribuição às variáveis que satisfaz ϕ (e portanto ϕ_{celula}) deve ter exatamente uma variável ligada para toda célula. Por conseguinte, qualquer atribuição satisfeitora especifica um símbolo em cada célula da tabela. As partes ϕ_{inicio}, $\phi_{\text{movemento}}$ e ϕ_{aceita} garantem que esses símbolos realmente correspondam a um tableau de aceitação da seguinte forma.

A fórmula ϕ_{inicio} garante que a primeira linha da tabela é a configuração inicial de N sobre w estipulando explicitamente que as variáveis correspondentes estão ligadas:

$$\begin{aligned} \phi_{\text{inicio}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \,. \end{aligned}$$

A fórmula ϕ_{aceita} garante que uma configuração de aceitação ocorre no tableau. Ela garante que q_{aceita}, o símbolo para o estado de aceitação, aparece em uma das células do tableau, estipulando que uma das variáveis correspondentes está

ligada:

$$\phi_{\text{aceita}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{aceita}}} .$$

Finalmente, a fórmula $\phi_{\text{movemento}}$ garante que cada linha da tabela corresponde a uma configuração que segue legalmente da configuração da linha precedente conforme as regras de N. Ela faz isso assegurando que cada janela 2×3 de células seja legal. Dizemos que uma janela 2×3 é ***legal*** se essa janela não viola as ações especificadas pela função de transição de N. Em outras palavras, uma janela é legal se ela pode aparecer quando uma configuração corretamente segue uma outra.[3]

Por exemplo, digamos que a, b e c sejam membros do alfabeto de fita e que q_1 e q_2 sejam estados de N. Assuma que, quando no estado q_1 com a cabeça lendo um a, N escreva um b, permaneça no estado q_1 e mova para a direita, e que quando no estado q_1 com a cabeça lendo um b, N não-deterministicamente

1. escreva um c, entre em q_2 e mova para a esquerda, ou
2. escreva um a, entre em q_2 e mova para a direita.

Expresso formalmente, $\delta(q_1, \texttt{a}) = \{(q_1,\texttt{b},\text{D})\}$ e $\delta(q_1, \texttt{b}) = \{(q_2,\texttt{c},\text{E}), (q_2,\texttt{a},\text{D})\}$. Exemplos de janelas legais para essa máquina são mostradas na Figura 7.39.

(a)
a	q_1	b
q_2	a	c

(b)
a	q_1	b
a	a	q_2

(c)
a	a	q_1
a	a	b

(d)
#	b	a
#	b	a

(e)
a	b	a
a	b	q_2

(f)
b	b	b
c	b	b

FIGURA 7.39
Exemplos de janelas legais

Na Figura 7.39, as janelas (a) e (b) são legais porque a função de transição permite a N mover da maneira indicada. A janela (c) é porque, com q_1 aparecendo no lado direito da linha superior, não sabemos sobre que símbolo a cabeça está. Esse símbolo poderia ser um a, e q_1 pode modificá-lo para um b e mover para a direita. Essa possibilidade daria origem a essa janela, portanto ela não viola as regras de N. A janela (d) é obviamente legal porque as partes superior e inferior

[3]Poderíamos dar uma definição precisa de ***janela legal*** aqui, em termos da função de transição. Mas fazer isso também é bastante cansativo e nos desviaria da principal linha do argumento de prova. Qualquer um que deseje mais precisão deve se reportar à análise relacionada na prova do Teorema 5.15, a indecidibilidade do problema da Correspondência de Post.

são idênticas, o que ocorreria se a cabeça não estivesse adjacente à localização da janela. Note que # pode aparecer à esquerda ou à direita das linhas superior e inferior em uma janela legal. A janela (e) é legal porque o estado q_1 lendo um b poderia ter estado imediatamente à direita da linha superior, e teria então movido para a esquerda no estado q_2 para aparecer no lado direito da linha inferior. Finalmente, a janela (f) é legal porque o estado q_1 poderia ter estado imediatamente à esquerda da linha superior e poderia ter modificado o b para um c e movido para a esquerda.

As janelas mostradas na Figura 7.39 não são legais para a máquina N.

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

FIGURA 7.40
Exemplos de janelas ilegais

Na janela (a) o símbolo central na linha superior não pode mudar porque um estado não estava adjacente a ele. A janela (b) não é legal porque a função de transição especifica que o b é modificado para um c mas não para um a. A janela (c) não é legal porque dois estados aparecem na linha inferior.

AFIRMAÇÃO 7.41

Se a linha superior da tabela for a configuração inicial e toda janela na tabela for legal, cada linha da tabela é uma configuração que segue legalmente da precedente.

Provamos essa afirmação considerando quaisquer duas configurações adjacentes na tabela, chamadas configuração superior e configuração inferior. Na configuração superior, toda célula que não é adjacente a um símbolo de estado e que não contém o símbolo de fronteira #, é a célula central superior em uma janela cuja linha superior não contém nenhum estado. Por conseguinte, esse símbolo deve aparecer imutável na posição central inferior da janela. Logo, ele aparece na mesma posição na configuração inferior.

A janela contendo o símbolo de estado na célula centra superior garante que as três posições correspondentes sejam atualizadas consistentemente com a função de transição. Conseqüentemente, se a configuração superior for uma configuração legal, o mesmo acontece com a configuração inferior, e a inferior segue a superior conforme as regras de N. Note que essa prova, embora fácil, depende crucialmente de nossa escolha de um tamanho de janela de 2×3, como mostra o Exercício 7.39.

Agora voltamos à construção de $\phi_{\text{movemento}}$. Ela estipula que todas as janelas no tableau são legais. Cada janela contém seis células, que podem ser inicializadas de

um número fixo de maneiras para originar uma janela legal. A fórmula $\phi_{\text{movemento}}$ diz que as inicializações daquelas seis células devem ser uma dessas maneiras, ou

$$\phi_{\text{movemento}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} (\text{the } (i,j) \text{ window is legal})$$

Substituimos o texto "a janela (i, j) é legal" nessa fórmula com a fórmula seguinte. Escrevemos o conteúdo de seis células de uma janela como $a_1, \ldots, a_6$.

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{é uma janela legal}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

A seguir analisamos a complexidade da redução para mostrar que ela opera em tempo polinomial. Para fazer isso examinamos o tamanho de ϕ. Primeiro, estimamos o número de variáveis que ela tem. Lembre-se de que o tableau é uma tabela $n^k \times n^k$, portanto ela contém n^{2k} células. Cada célula tem l variáveis associadas a ela, onde l é o número de símbolos em C. Em razão do fato de que l depende somente da MT N e não do comprimento da entrada n, o número total de variáveis é $O(n^{2k})$.

Estimamos o tamanho de cada uma das partes de ϕ. A fórmula ϕ_{celula} contém um fragmento de tamanho-fixo da fórmula para cada célula do tableau, portanto seu tamanho é $O(n^{2k})$. A fórmula ϕ_{inicio} tem um fragmento para cada célula na linha superior, portanto seu tamanho é $O(n^k)$. As fórmulas $\phi_{\text{movemento}}$ e ϕ_{aceita} cada uma contém um fragmento de tamanho-fixo da fórmula para cada célula do tableau, portanto seu tamanho é $O(n^{2k})$. Conseqüentemente, o tamanho total de ϕ é $O(n^{2k})$. Esse limitante é suficiente para nossos propósitos porque ele mostra que o tamanho de ϕ é polinomial em n. Se fosse mais que polinomial, a redução não teria nenhuma chance de gerá-la em tempo polinomial. (Na verdade, nossas estimativas são baixas por um fator de $O(\log n)$ porque cada variável tem índices que podem ir até n^k e portanto podem requerer $O(\log n)$ símbolos para escrever na fórmula, mas esse fator adicional não modifica a polinomialidade do resultado.)

Para ver que podemos gerar a fórmula em tempo polinomial, observe sua natureza altamente repetitiva. Cada componente da fórmula é composto de muitos fragmentos quase idênticos, que diferem apenas nos índices de uma maneira simples. Conseqüentemente, podemos facilmente construir uma redução que produz ϕ em tempo polinomial a partir da entrada w.

Por conseguinte, concluimos a prova do teorema de Cook–Levin, mostrando que *SAT* é NP-completa. Mostrar a NP-completude de outras linguagens geralmente não requer uma prova tão longa. Ao contrário, a NP-completude pode ser provada com uma redução de tempo polinomial a partir de uma linguagem que já se sabe que é NP-completa. Podemos usar *SAT* para esse propósito, mas usar *3SAT*, o caso especial de *SAT* que definimos na página 292, é usualmente mais fácil. Lembre-se de que as fórmulas em *3SAT* estão na forma normal conjuntiva

(fnc) com três literais por cláusula. Primeiro, temos que mostrar que *3SAT* propriamente dita é NP-completa. Provamos essa asserção como um corolário do Teorema 7.37.

COROLÁRIO 7.42

3SAT é NP-completa.

PROVA Obviamente *3SAT* está em NP, portanto somente precisamos provar que todas as linguagens em NP se reduzem a *3SAT* em tempo polinomial. Uma maneira de fazê-lo é mostrar que *SAT* reduz em tempo polinomial a *3SAT*. Ao invés disso, modificamos a prova do Teorema 7.37 de tal forma que ele produza diretamente uma fórmula na forma normal conjuntiva com três literais por cláusula.

O Teorema 7.37 produz uma fórmula que já está quase na forma normal conjuntiva. A fórmula ϕ_{celula} é um grand E de subfórmulas, cada uma das quais contém um grande OU e um grande E de OUs. Por conseguinte, ϕ_{celula} é um E de cláusulas e por isso já está na fnc. A fórmula ϕ_{inicio} é um grande E de variáveis. Tomando cada uma dessas variáveis como sendo uma cláusula de tamanho 1 vemos que ϕ_{inicio} está na fnc. A fórmula ϕ_{aceita} é um grande OU de variáveis e é portanto uma única cláusula. A fórmula $\phi_{\text{movemento}}$ é a única que ainda não está na fnc, mas podemos facilmente convertê-la numa fórmula está na fnc da seguinte forma.

Lembre-se de que $\phi_{\text{movemento}}$ é um grand E de subfórmulas, cada uma das quais é um OU de Es que descreve todas as janelas legais possíveis. As leis distributivas, como descritas no Capítulo 0, afirmam que podemos substituir um OU de Es por um E de OUs equivalente. Fazer isso pode aumentar significativamente o tamanho de cada subfórmula, mas pode aumentar o tamanho total de $\phi_{\text{movemento}}$ somente de um fator constante porque o tamanho de cada subfórmula depende apenas de N. O resultado é uma fórmula que está na forma normal conjuntiva.

Agora que escrevemos a fórmula na fnc, convertemo-la para uma fórmula com três literais por cláusula. Em cada cláusula que correntemente tem um ou dois literais, replicamos um dos literais até que o número total seja três. Em cada cláusula que tem mais de três literais, dividimo-la em várias cláusulas e acrescentamos variáveis extras para preservar a satisfiabilidade ou não-satisfiabilidade da original.

Por exemplo, substituimos a cláusula $(a_1 \vee a_2 \vee a_3 \vee a_4)$, na qual cada a_i é um literal, pela expressão de duas-cláusulas $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$, na qual z é uma nova variável. Se alguma valoração de a_i's satisfaz a cláusula original, podemos encontrar alguma valoração de z de modo que as duas novas cláusulas sejam satisfeitas. Em geral, se a cláusula contém l literais,

$$(a_1 \vee a_2 \vee \cdots \vee a_l),$$

podemos substituí-la pelas $l - 2$ cláusulas

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l).$$

Podemos facilmente verificar que a nova fórmula é satisfatível sse a fórmula original o era, portanto a prova está completa.

7.5 PROBLEMAS NP-COMPLETOS ADICIONAIS

O fenômeno da NP-completude está espalhado. Problemas NP-completos aparecem em muitas áreas. Por razões que não são bem entendidas, a maioria dos problemas NP que ocorrem naturalmente sabe-se que estão em P ou que são NP-completos. Se você busca um algoritmo de tempo polinomial para um novo problema NP, gastar parte de seu esforço tentando provar que ele é NP-completo é sensato porque fazer isso pode lhe evitar de trabalhar para encontrar um algoritmo de tempo polinomial que não existe.

Nesta seção apresentamos teoremas adicionais mostrando que várias linguagens são NP-completas. Esses teoremas provêm exemplos das técnicas que são usadas em provas desse tipo. Nossa estratégia geral é exibir uma redução de tempo polinomial a partir de *3SAT* para a linguagem em questão, embora às vezes reduzimos a partir de outras linguagens NP-completas quando isso é mais conveniente.

Quando construimos uma redução de tempo polinomial a partir de *3SAT* para uma linguagem, procuramos por estruturas naquela linguagem que possam simular as variáveis e cláusulas nas fórmulas booleanas. Tais estruturas são às vezes chamadas ***engrenagem***. Por exemplo, na redução de *3SAT* para *CLIQUE* apresentada no Teorema 7.32, os nós individuais simulam variáveis e triplas de nós simulam cláusulas. Um nó individual pode ou não ser um membro do clique, o que corresponde a uma variável que pode ou não ser verdadeira em uma atribuição satisfeitora. Cada cláusula tem que conter um literal que é atribuído VERDADEIRO e que corresponde à forma pela qual cada tripla tem que conter um nó no clique se o tamanho alvo é para ser atingido. O seguinte corolário do Teorema 7.32 afirma que *CLIQUE* é NP-completa.

COROLÁRIO 7.43

CLIQUE is NP-completa.

O PROBLEMA DA COBERTURA DE VÉRTICES

Se G é um grafo não-direcionado, uma ***cobertura de vértices*** de G é um subconjunto dos nós onde toda aresta de G toca um daqueles nós. O problema da cobertura de vértices pergunta se um grafo contém uma cobertura de vértices de um tamanho especificado:

$$\textit{VERTEX-COVER} = \{\langle G, k\rangle \mid G \text{ é um grafo não-direcionado que tem uma cobertura de vértices de } k\text{-nós}\}.$$

TEOREMA 7.44

VERTEX-COVER é NP-completo.

IDÉIA DA PROVA Para mostrar que *VERTEX-COVER* é NP-completo temos que mostrar que ele está em NP e que todos os problemas NP são redutíveis em tempo polinomial a ele. A primeira parte é fácil; um certificado é simplesmente uma cobertura de vértices de tamanho k. Para provar a segunda parte mostramos que *3SAT* é redutível em tempo polinomial a *VERTEX-COVER*. A redução converte uma 3fnc-fórmula ϕ num grafo G e um número k, de modo que ϕ é satisfatível sempre que G tem uma cobertura de vértices com k nós. A conversão é feita sem saber se ϕ é satisfatível. Com efeito, G simula ϕ. O grafo contém engrenagens que imitam as variáveis e cláusulas da fórmula. Projetar essas engrenagens requer um pouco de engenhosidade.

Para a engrenagem das variáveis, procuramos por uma estrutura em G que possa participar da cobertura de vértices em uma das duas maneiras possíveis, correspondendo às duas possíveis atribuições de verdade à variável. Dois nós conectados por uma aresta é uma estrutura que funciona, porque um desses nós tem que aparecer na cobertura de vértices. Arbitrariamente atribuimos VERDADEIRO e FALSO a esses dois nós.

Para a engrenagem das cláusulas, buscamos uma estrutura que induza a cobertura de vértices para incluir nós nas engrenagens de variáveis correspondendo a pelo menos um literal verdadeiro na cláusula. A engrenagem contém três nós e arestas adicionais de modo que que qualquer cobertura de vértices tem que incluir pelo menos dois dos nós, ou possivelmente todos os três. Somente dois nós seriam necessários se um dos nós da engrenagem de vértices ajuda cobrindo uma aresta, como aconteceria se o literal associado satisfaz essa cláusula. Caso contrário três nós seriam necessários. Finalmente, escolhemos k de modo que a cobertura de vértices procurada tem um nó por engrenagem de variáveis e dois nós por engrenagem de cláusulas.

PROVA Aqui estão os detalhes de uma redução de *3SAT* para *VERTEX-COVER* que opera em tempo polinomial. A redução mapeia uma fórmula booleana ϕ para um grafo G e um valor k. Para cada variável x em ϕ, produzimos uma aresta conectando dois nós. Rotulamos os dois nós nessa engrenagem x e $\overline{x}$. Fazer x VERDADEIRO corresponde a selecionar o nó esquerdo para a cobertura de vértices, enquanto que FALSO corresponde ao nó direito.

As engrenagens para as cláusulas são um pouco mais complexas. Cada engrenagem de cláusulas é uma tripla de três nós que são rotulados com três literais da

cláusula. Esses três nós são conectados um ao outro e aos nós nas engrenagens de variáveis que têm os rótulos idênticos. Por conseguinte, o número total de nós que aparecem em G é $2m+3l$, onde ϕ tem m variáveis e l cláusulas. Suponha que k seja $m + 2l$.

Por exemplo, se $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$, a redução produz $\langle G, k\rangle$ a partir de ϕ, onde $k = 8$ e G toma a forma mostrada na Figura 7.45.

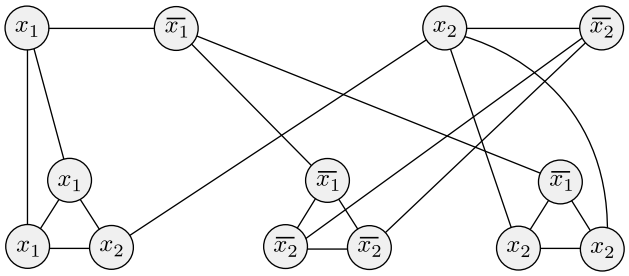


FIGURA 7.45
O grafo que a redução produz a partir de
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Para provar que essa redução funciona, precisamos mostrar que ϕ é satisfatível se e somente se G tem uma cobertura de vértices com k nós. Começamos com uma atribuição satisfeitora. Primeiro colocamos os nós das engrenagens de variáveis que correspondem aos literais verdadeiros na atribuição na cobertura de vértices. Então, selecionamos um literal verdadeiro em toda cláusula e colocamos os dois nós remanescentes de toda engrenagem de cláusulas na cobertura de vértices. Agora, temos um total de k nós. Eles cobrem todas as arestas porque toda engrenagem de variáveis é claramente coberta, todas as três arestas dentro de toda engrenagem de cláusulas são cobertas, e todas as arestas entre as engrenagens de variáveis e de cláusulas são cobertas. Logo, G tem uma cobertura de vértices com k nós.

Segundo, se G tem uma cobertura de vértices com k nós, mostramos que ϕ é satisfatível construindo a atribuição satisfeitora. A cobertura de vértices tem que conter um nó em cada engrenagem de variáveis e dois em toda engrenagem de cláusulas de forma a cobrir as arestas das engrenagens de variáveis e as três arestas dentro das engrenagens de cláusulas. Isso dá conta de todos os nós, portanto nenhum resta. Tomamos os nós das engrenagens de variáveis que estão na cobertura de vértices e atribuimos VERDADEIRO aos literais correspondentes. Essa atribuição satisfaz ϕ porque cada uma das três arestas conectando as engrenagens de variáveis com cada engrenagem de cláusulas é coberta e somente dois nós da engrenagem de cláusulas estão na cobertura de vértices. Conseqüentemente, uma das arestas tem que ser coberta por um nó de uma engrenagem de

variáveis e portanto essa atribuição satisfaz a cláusula correspondente.

O PROBLEMA DO CAMINHO HAMILTONIANO

Lembre-se de que o problema do caminho hamiltoniano pergunta se o grafo de entrada contém um caminho de s para t que passa por todo nó exatamente uma vez.

TEOREMA 7.46

HAMPATH é NP-completo.

IDÉIA DA PROVA Mostramos que *HAMPATH* está em NP na Seção 7.3. Para mostrar que todo problema NP é redutível em tempo polinomial a *HAMPATH*, mostramos que *3SAT* é redutível em tempo polinomial a *HAMPATH*. Damos uma maneira de converter 3fnc-fórmulas para grafos na qual caminhos hamiltonianos correspondem a atribuições satisfeitoras da fórmula. Os grafos contêm engrenagens que imitam variáveis e cláusulas. A engrenagem de variáveis é uma estrutura em formato de diamante que pode ser percorrida em duas das seguintes maneiras, correspondendo a duas atribuições satisfeitoras. A engrenagem de cláusulas é um nó. Assegurar que o caminho passa por cada engrenagem de cláusulas corresponde a assegurar que cada cláusula seja satisfeita na atribuição satisfeitora.

PROVA Anteriormente demonstramos que *HAMPATH* está em NP, portanto tudo o que resta a ser feito é mostrar que *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH*. Para cada 3fnc-fórmula ϕ mostramos como construir um grafo direcionado G com dois nós, s e t, onde um caminho hamiltoniano existe entre s e t sse ϕ é satisfatível.

Começamos a construção com uma 3fnc-fórmula ϕ contendo k cláusulas:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k),$$

onde cada a, b e c é um literal x_i ou $\overline{x_i}$. Sejam $x_1, \ldots, x_l$ as l variáveis de ϕ.

Agora mostramos como converter ϕ num grafo G. O grafo G que construimos tem várias partes para representar as variáveis e cláusulas que aparecem em ϕ.

Represente cada variável x_i com uma estrutura num formato de diamante que contém uma linha horizontal de nós, como mostrado na Figura 7.47. Mais adiante especificamos o número de nós que aparecem na linha horizontal.

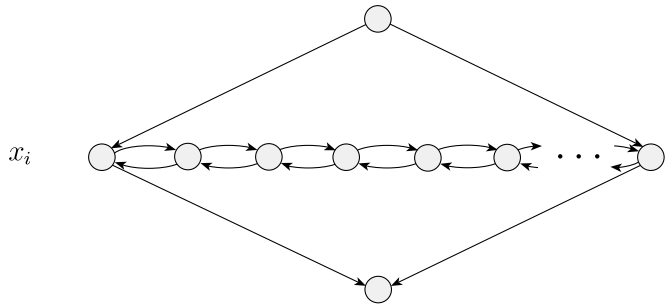


FIGURA 7.47
Representando a variável x_i como uma estrutura no formato de um diamante

Pepresentamos cada cláusula de ϕ como um único nó, da seguinte forma.

FIGURA 7.48
Representando a cláusula c_j como um nó

A Figura 7.49 exibe a estrutura global de G. Ela mostra todos os elementos de G e seus relacionamentos, exceto as arestas que representam o relacionamento das variáveis com as cláusulas que as contêm.

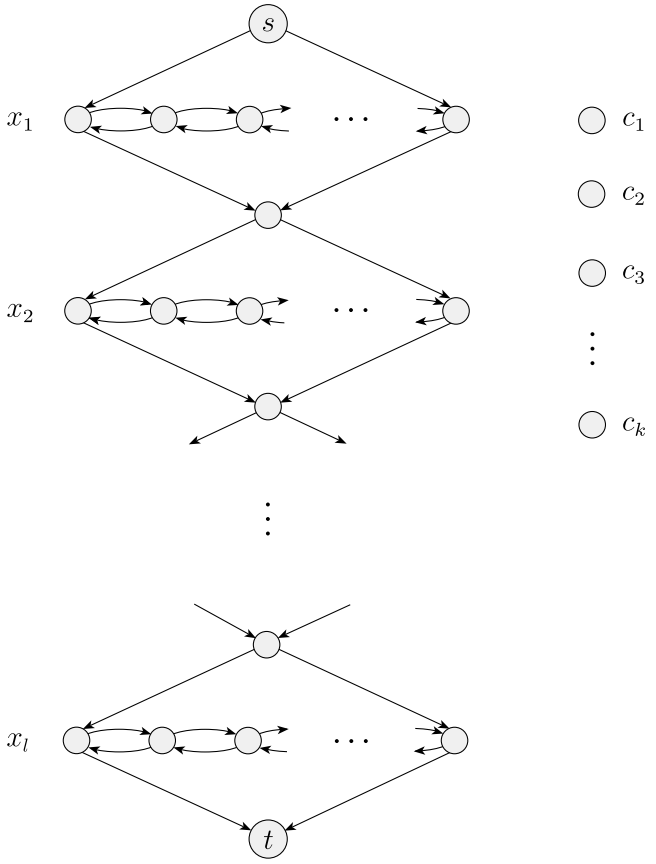


FIGURA 7.49
A estrutura de alto-nível de G

A seguir mostramos como conectar os diamantes representando as variáveis aos nós representando as cláusulas. Cada estrutura de diamante contém uma linha horizontal de nós conectados por arestas correndo em ambas as direções. A linha horizontal contém $3k + 1$ nós além dos dois nós nas extremidades pertencentes ao diamante. Esses nós são agrupados em pares adjacentes, um para cada cláusula, com nós separadores extras em seguida aos pares, como mostrado na Figura 7.50.

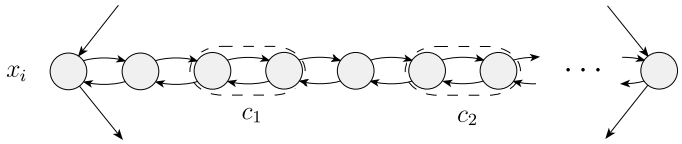


FIGURA 7.50
Os nós horizontais em uma estrutura em formato de diamante

Se a variável x_i aparece na cláusula c_j, adicionamos as duas arestas seguintes do j-ésimo par no i-ésimo diamante ao j-ésimo nó cláusula.

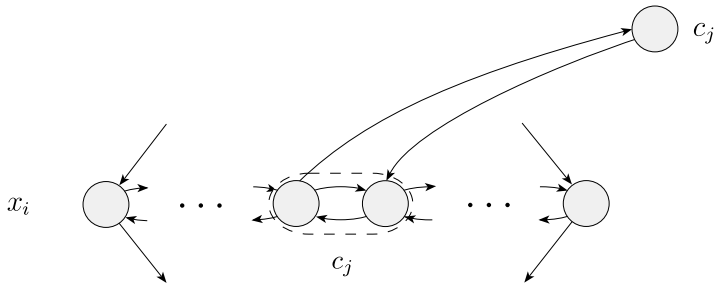


FIGURA 7.51
As arestas adicionais quando a cláusula c_j contém x_i

Se $\overline{x_i}$ aparece na cláusula c_j, adicionamos duas arestas do j-ésimo par no i-ésimo diamante ao j-ésimo nó cláusula, como mostrado na Figura 7.52.

Depois que adicionamos todas as arestas correspondentes a cada ocorrência de x_i ou $\overline{x_i}$ em cada cláusula, a construção de G está completa. Para mostrar que essa construção funciona, argumentamos que, se ϕ é satisfatível, um caminho hamiltoniano existe de s para t e, reciprocamente, se tal caminho existe, ϕ é satisfatível.

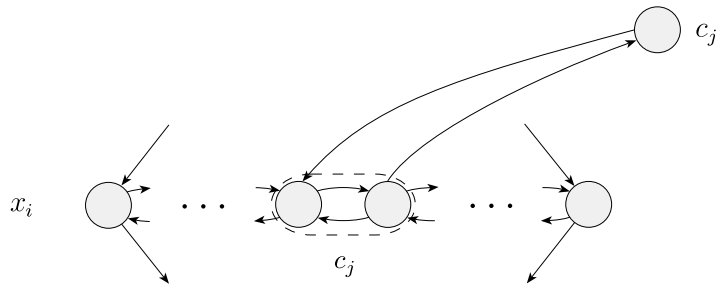


FIGURA 7.52
As arestas adicionais quando a cláusula c_j contém $\overline{x_i}$

Suponha que ϕ seja satisfatível. Para exibir um caminho hamiltoniano de s para t, primeiro ignoramos os nós cláusula. O caminho começa em s, passa por cada diamante por sua vez, e termina em t. Para atingir os nós horizontais em um diamante, o caminho ou zigue-zagueia da esquerda para a direita ou zague-zigueia da direita para a esquerda, a atribuição satisfeitora para ϕ determina qual. Se x_i for atribuída VERDADEIRO, o caminho zigue-zagueia através do diamante correspondente. Se x_i for atribuído FALSE, o caminho zague-zigueia. Mostramos ambas as possibilidades na Figura 7.53.

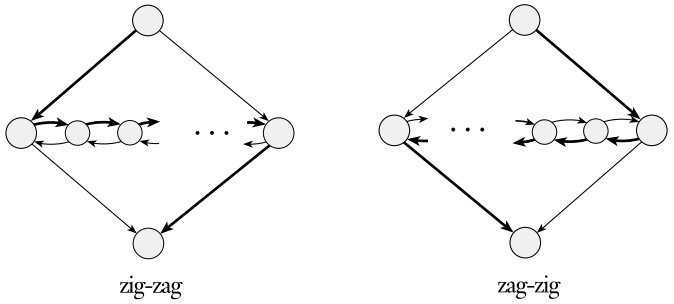


FIGURA 7.53
Zigue-zagueando e zague-zigueando através de um diamante, como determinado pela atribuição satisfeitora

Até agora esse caminho cobre todos os nós em G exceto os nós cláusula. Podemos facilmente incluí-los adicionando desvios nos nós horizontais. Em cada cláusula, selecionamos um dos literais atribuídos VERDADEIRO pela atribuição satisfeitora.

Se selecionássemos x_i na cláusula c_j, podemos desviar no j-ésimo par no i-ésimo diamante. Fazer isso é possível porque x_i tem que ser VERDADEIRO, portanto o caminho zigue-zagueia da esquerda para a direita pelo diamante correspondente. Logo, as arestas para o nó c_j estão na ordem correta para permitir um desvio e retorno.

Similarmente, se selecionássemos $\overline{x_i}$ na cláusula c_j, podemos desviar no j-ésimo par no i-ésimo diamante porque x_i tem que ser FALSO, portanto o caminho zague-zigueia da direita para a esquerda pelo diamante correspondente. Logo, as arestas para o nó c_j novamente estão na ordem correta para permitir um desvio e retorno. (Note que cada literal verdadeiro numa cláusula provê uma *opção* de um desvio para atingir o nó cláusula. Como um resultado, se vários literais numa cláusula são verdadeiros, somente um desvio é tomado.) Por conseguinte, construimos o caminho hamiltoniano desejado.

Para a direção reversa, se G tem um caminho hamiltoniano de s para t, exibimos uma atribuição satisfeitora para ϕ. Se o caminho hamiltoniano é *normal*—ele passa pelos diamantes na ordem do superior para o inferior, exceto pelos desvios para os nós cláusula—podemos facilmente obter a atribuição satisfeitora. Se o caminho zigue-zagueia pelo diamante, atribuimos à variável correspondente VERDADEIRO, e se ele zague-zigueia, atribuimos FALSE. Em razão do fato de que cada nó cláusula aparece no caminho, observando como o desvio para ele é tomado, podemos determinar qual dos literais na cláusula correspondente é VERDADEIRO.

Tudo o que resta para ser mostrado é que um caminho hamiltoniano tem que ser normal. Normalidade pode falhar somente se o caminho entra numa cláusula de um diamante mas retorna a um outro, como na Figura 7.54.

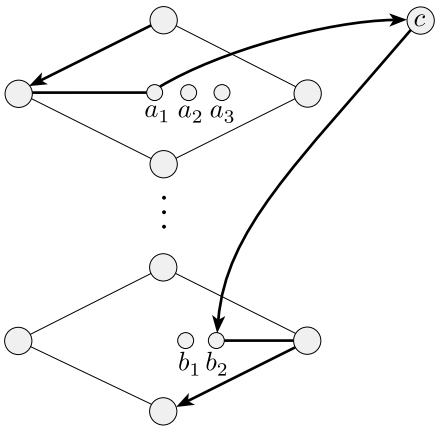


FIGURA 7.54
Essa situação não pode ocorrer

O caminho vai do nó a_1 para c, mas ao invés de retornar para a_2 no mesmo diamante, ele retorna para b_2 num diamante diferente. Se isso ocorre, ou a_2 ou a_3 tem que ser um nó separador. Se a_2 fosse um nó separador node, as únicas arestas entrando em a_2 seriam de a_1 e a_3. Se a_3 fosse um nó separador, a_1 e a_2 estaríamos no mesmo par de cláusulas, e portanto as únicas arestas entrando em

a_2 seriam de a_1, a_3 e c. Em qualquer dos casos, o caminho não poderia conter o nó a_2. O caminho não pode entrar em a_2 de c ou a_1 porque o caminho vai para outros lugares a partir desses nós. O caminho não pode entrar em a_2 a partir de a_3, porque a_3 é o único nó disponível para o qual a_2 aponta, portanto o caminho tem que deixar a_2 via a_3. Logo, um caminho hamiltoniano tem que ser normal. Essa redução obviamente opera em tempo polinomial e a prova está completa.

A seguir consideramos uma versão não-direcionada do problema do caminho hamiltoniano, chamado *UHAMPATH*. Para mostrar que *UHAMPATH* é NP-completo damos uma redução de tempo polinomial da versão direcionada do problema.

TEOREMA 7.55

UHAMPATH é NP-completo.

PROVA A redução toma um grafo direcionado G com nós s e t e constrói um grafo não-direcionado G' com nós s' e t'. O grafo G tem um caminho hamiltoniano de s para t sse G' tem um caminho hamiltoniano de s' para t'. Descrevemos G' da seguinte forma.

Cada nó u de G, exceto por s e t, é substituído por uma tripla de nós u^{in}, u^{mid} e u^{out} em G'. Os nós s e t em G são substituídos por nós s^{sai} e t^{entra} em G'. As arestas de dois tipos aparecem em G'. Primeiro, as arestas conectam u^{meio} com u^{entra} e u^{sai}. Segundo, uma aresta conecta u^{sai} com v^{entra} se uma aresta vai de u para v em G. Isso completa a construção de G'.

Podemos demonstrar que essa construção funciona mostrando que G tem um caminho hamiltoniano de s para t sse G' tem um caminho hamiltoniano de s^{sai} para t^{entra}. Para mostrar uma direção, observamos que um caminho hamiltoniano P em G,

$$s, u_1, u_2, \ldots, u_k, t,$$

tem um caminho hamiltoniano P' em G',

$$s^{\text{out}}, u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, u_2^{\text{mid}}, u_2^{\text{out}}, \ldots, t^{\text{in}}.$$

Para mostrar a outra direção, afirmamos que qualquer caminho hamiltoniano em G' de s^{sai} para t^{entra} em G' deve ir de uma tripla de nós para uma tripla de nós, exceto pelo início e o fim, como faz o caminho P' que acabamos de descrever. Isso completaria a prova porque qualquer desses caminhos tem um caminho hamiltoniano correspondente em G. Provamos a afirmação seguindo o caminho começando no nó s^{sai}. Observe que o nó seguinte no caminho deve ser u_i^{entra} para algum i porque somente aqueles nós são conectados a s^{sai}. O nó seguinte deve ser u_i^{meio}, porque nenhuma outra maneira está disponível para incluir u_i^{meio} no caminho hamiltoniano. Após u_i^{meio} vem u_i^{sai} porque esse é o único outro ao qual u_i^{meio} está conectado. O nó seguinte deve ser u_j^{entra} para algum j

porque nenhum outro nó disponível está conectado a u_i^{sai}. O argumento então se repete até que t^{entra} seja atingido.

O PROBLEMA DA SOMA DE SUBCONJUNTOS

Retomemos o problema *SUBSET-SUM* definido na página 287. Naquele problema, nos era dada uma coleção de números $x_1, \ldots, x_k$ juntamente com um número alvo t, e tínhamos que determinar se a coleção contém uma subcoleção cuja soma é t. Agora mostramos que esse problema é NP-completo.

TEOREMA 7.56

SUBSET-SUM é NP-completo.

IDÉIA DA PROVA Já mostramos que *SUBSET-SUM* está em NP no Teorema 7.25. Provamos que todas as linguagens em NP são redutíveis em tempo polinomial a *SUBSET-SUM* reduzindo a linguagem NP-completa *3SAT* a ela. Dada uma 3fnc-fórmula ϕ construimos uma instância do problema *SUBSET-SUM* que contém uma subcoleção cuja soma é o alvo t se e somente se ϕ é satisfatível. Chame essa subcoleção T.

Para conseguir essa redução encontramos estruturas do problema *SUBSET-SUM* que representem variáveis e cláusulas. A instância do problema *SUBSET-SUM* que construimos contém números de grande magnitude apresentados em notação decimal. Representamos variáveis por pares de números e cláusulas por certas posições nas representações decimais dos números.

Representamos a variável x_i por dois números, y_i e z_i. Provamos que ou y_i ou z_i deve estar em T para cada i, o que estabelece a codificação para o valor-verdade de x_i na atribuição satisfeitora.

Cada posição de cláusula contém um certo valor no alvo t, o que impõe um requisito no subconjunto T. Provamos que esse requisito é o mesmo que aquele na cláusula correspondente—a saber, que um dos literais nessa cláusula é atribuída VERDADEIRO.

PROVA Já sabemos que *SUBSET-SUM* $\in$ NP, portanto agora mostramos que *3SAT* $\leq_{\text{P}}$ *SUBSET-SUM*.

Seja ϕ uma fórmula booleana com as variáveis $x_1, \ldots, x_l$ e as cláusulas $c_1, \ldots, c_k$. A redução converte ϕ para uma instância do problema *SUBSET-SUM* $\langle S, t \rangle$, na qual os elementos de S e o número t são as linhas na tabela na Figura 7.57, expressos na notação decimal ordinária. As linhas acima da linha dupla são rotuladas

$$y_1, z_1,\ y_2, z_2,\ \ldots, y_l, z_l \quad \text{e} \quad g_1, h_1,\ g_2, h_2,\ \ldots, g_k, h_k$$

e compreende os elementos de S. A linha abaixo da linha dupla é t.

Por conseguinte, S contém um par de números, y_i, z_i, para cada variável x_i em ϕ. A representação decimal desses números está em duas partes, como indicados na tabela. A parte da esquerda compreende um 1 seguido de $l - i$ 0s. A parte direita contém um dígito para cada cláusula, onde o j-ésimo dígito de y_i é 1 se a cláusula c_j contém o literal x_i e o j-ésimo dígito de z_i é 1 se a cláusula c_j contém o literal $\overline{x_i}$. Os dígitos não especificados como sendo 1 são 0.

A tabela é parcialmente preenchida para ilustrar cláusulas de amostra, c_1, c_2 e c_k:

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \cdots) \wedge \cdots \wedge (\overline{x_3} \vee \cdots \vee \cdots).$$

Adicionalmente, S contém um par de números, g_j, h_j, para cada cláusula c_j. Esses dois números são iguais e consistem de um 1 seguido por $k - j$ 0s.

Finalmente, o número alvo t, a linha inferior da tabela, consiste de l 1s seguidos por k 3s.

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

FIGURA 7.57
Reduzindo *3SAT* a *SUBSET-SUM*

Agora mostramos por que essa construção funciona. Demonstramos que ϕ é satisfatível sse algum subconjunto de S soma t.

Suponha que ϕ seja satisfatível. Construimos um subconjunto de S da se-

guinte forma. Selecionamos y_i se x_i é atribuída VERDADEIRO na atribuição satisfeitora e z_i se x_i é atribuída FALSO. Se somarmos o que selecionamos até então, obtemos um 1 em cada um dos primeiros l dígitos porque selecionamos ou y_i ou z_i para cada i. Além do mais, cada um dos últimos k dígitos é um número entre 1 e 3 porque cada cláusula é satisfeita e portanto contém entre 1 e 3 literais verdadeiros. Agora selecionamos ainda uma quantidade suficiente dos números g e h para trazer cada um dos últimos k dígitos para 3, portanto atingindo o alvo.

Suppose that a subset of S sums to t. We construct a satisfying assignment to ϕ after making several observations. First, all the digits in members of S are either 0 or 1. Furthermore, each column in the table describing S contains at most five 1s. Hence a "carry" into the next column never occurs when a subset of S is added. To get a 1 in each of the first l columns the subset must have either y_i or z_i for each i, but not both.

Now we make the satisfying assignment. If the subset contains y_i, we assign x_i VERDADEIRO; otherwise, we assign it FALSE. This assignment must satisfy ϕ because in each of the final k columns the sum is always 3. In column c_j, at most 2 can come from g_j and h_j, so at least 1 in this column must come from some y_i or z_i in the subset. If it is y_i, then x_i appears in c_j and is assigned VERDADEIRO, so c_j is satisfied. If it is z_i, then $\overline{x_i}$ appears in c_j and x_i is assigned FALSE, so c_j is satisfied. Therefore ϕ is satisfied.

Finally, we must be sure that the reduction can be carried out in polynomial time. The table has a size of roughly $(k + l)^2$, and each entry can be easily calculated for any ϕ. So the total time is $O(n^2)$ easy stages.

EXERCÍCIOS

7.1 Answer each part TRUE or FALSE.

a. $2n = O(n)$.
b. $n^2 = O(n)$.
R c. $n^2 = O(n \log^2 n)$.
R d. $n \log n = O(n^2)$.
e. $3^n = 2^{O(n)}$.
f. $2^{2^n} = O(2^{2^n})$.

7.2 Answer each part TRUE or FALSE.

a. $n = o(2n)$.
b. $2n = o(n^2)$.
R c. $2^n = o(3^n)$.
R d. $1 = o(n)$.
e. $n = o(\log n)$.
f. $1 = o(1/n)$.

7.3 Which of the following pairs of numbers are relatively prime? Show the calculations that led to your conclusions.

a. 1274 and 10505

b. 7289 and 8029

7.4 Fill out the table described in the polynomial time algorithm for context-free language recognition from Theorem 7.16 for string $w =$ baba and GLC G:

$$\begin{aligned} S &\to RT \\ R &\to TR \mid \mathtt{a} \\ T &\to TR \mid \mathtt{b} \end{aligned}$$

7.5 Is the following formula satisfiable?

$$(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$$

7.6 Show that P is closed under union, concatenation, and complement.

7.7 Show that NP is closed under union and concatenation.

7.8 Let $CONNECTED = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$. Analyze the algorithm given on page 167 to show that this language is in P.

7.9 A ***triangle*** in an undirected graph is a 3-clique. Show that $TRIANGLE \in$ P, where $TRIANGLE = \{\langle G \rangle \mid G \text{ contains a triangle}\}$.

7.10 Show that TOD_{AFD} is in P.

7.11 Call graphs G and H ***isomorphic*** if the nodes of G may be reordered so that it is identical to H. Let $ISO = \{\langle G, H \rangle \mid G \text{ and } H \text{ are isomorphic graphs}\}$. Show that $ISO \in$ NP.

PROBLEMAS

7.12 Let

$$MODEXP = \{\langle a, b, c, p \rangle \mid a, b, c, \text{ and } p \text{ are binary integers such that } a^b \equiv c \pmod{p}\}.$$

Show that $MODEXP \in$ P. (Note that the most obvious algorithm doesn't run in polynomial time. Hint: Try it first where b is a power of 2.)

7.13 A ***permutation*** on the set $\{1, \ldots, k\}$ is a one-to-one, onto function on this set. When p is a permutation, p^t means the composition of p with itself t times. Let

$$PERM\text{-}POWER = \{\langle p, q, t \rangle \mid p = q^t \text{ where } p \text{ and } q \text{ are permutations on } \{1, \ldots, k\} \text{ and } t \text{ is a binary integer}\}.$$

Show that $PERM\text{-}POWER \in$ P. (Note that the most obvious algorithm doesn't run within polynomial time. Hint: First try it where t is a power of 2).

7.14 Show that P is closed under the star operation. (Hint: Use dynamic programming. On input $y = y_1 \cdots y_n$ for $y_i \in \Sigma$, build a table indicating for each $i \leq j$ whether the substring $y_i \cdots y_j \in A^*$ for any $A \in$ P.)

R7.15 Show that NP is closed under the star operation.

7.16 Let *UNARY-SSUM* be the subset sum problem in which all numbers are represented in unary. Why does the NP-completeness proof for *SUBSET-SUM* fail to show *UNARY-SSUM* is NP-complete? Show that *UNARY-SSUM* $\in$ P.

7.17 Show that, if P = NP, then every language $A \in$ P, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete.

*7.18 Show that *PRIMES* $= \{m | m$ is a prime number in binary$\} \in$ NP. (Hint: For $p > 1$ the multiplicative group $Z_p^* = \{x |\, x$ is relatively prime to p and $1 \leq x < p\}$ is both cyclic and of order $p - 1$ iff p is prime. You may use this fact without justifying it. The stronger statement *PRIMES* $\in$ P is now known to be true, but it is more difficult to prove.)

7.19 We generally believe that *PATH* is not NP-complete. Explain the reason behind this belief. Show that proving *PATH* is not NP-complete would prove P $\neq$ NP.

7.20 Let G represent an undirected graph. Also let

$$SPATH = \{\langle G, a, b, k\rangle |\, G \text{ contains a simple path of length at most } k \text{ from } a \text{ to } b\},$$

and

$$LPATH = \{\langle G, a, b, k\rangle |\, G \text{ contains a simple path of length at least } k \text{ from } a \text{ to } b\}.$$

a. Show that *SPATH* $\in$ P.

b. Show that *LPATH* is NP-complete. You may assume the NP-completeness of *UHAMPATH*, the Hamiltonian path problem for undirected graphs.

7.21 Let *DOUBLE-SAT* $= \{\langle\phi\rangle |\, \phi$ has at least two satisfying assignments$\}$. Show that *DOUBLE-SAT* is NP-complete.

R**7.22** Let *HALF-CLIQUE* $= \{\langle G\rangle |\, G$ is an undirected graph having a complete subgraph with at least $m/2$ nodes, where m is the number of nodes in $G\}$. Show that *HALF-CLIQUE* is NP-complete.

7.23 Let *CNF*$_k$ $= \{\langle\phi\rangle |\, \phi$ is a satisfiable cnf-formula where each variable appears in at most k places$\}$.

a. Show that *CNF*$_2 \in$ P.

b. Show that *CNF*$_3$ is NP-complete.

7.24 Let ϕ be a 3cnf-formula. An ***≠-assignment*** to the variables of ϕ is one where each clause contains two literals with unequal truth values. In other words, an $\neq$-assignment satisfies ϕ without assigning three true literals in any clause.

a. Show that the negation of any $\neq$-assignment to ϕ is also an $\neq$-assignment.

b. Let $\neq$*SAT* be the collection of 3cnf-formulas that have an $\neq$-assignment. Show that we obtain a polynomial time reduction from *3SAT* to $\neq$*SAT* by replacing each clause c_i

$$(y_1 \vee y_2 \vee y_3)$$

with the two clauses

$$(y_1 \vee y_2 \vee z_i) \quad \text{and} \quad (\overline{z_i} \vee y_3 \vee b),$$

where z_i is a new variable for each clause c_i and b is a single additional new variable.

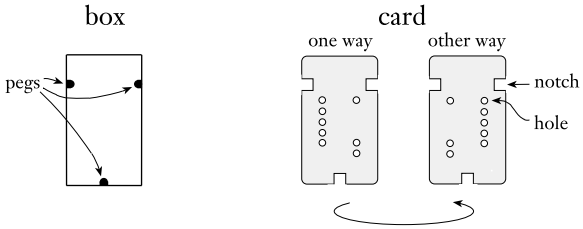
c. Conclude that $\neq SAT$ is NP-complete.

7.25 A ***cut*** in an undirected graph is a separation of the vertices V into two disjoint subsets S and T. The size of a cut is the number of edges that have one endpoint in S and the other in T. Let

$$\mathit{MAX\text{-}CUT} = \{\langle G, k\rangle \mid G \text{ has a cut of size } k \text{ or more}\}.$$

Show that *MAX-CUT* is NP-complete. You may assume the result of Problem 7.24. (Hint: Show that $\neq SAT \leq_{\mathrm{P}} \mathit{MAX\text{-}CUT}$. The variable gadget for variable x is a collection of $3c$ nodes labeled with x and another $3c$ nodes labeled with $\overline{x}$, where c is the number of clauses. All nodes labeled x are connected with all nodes labeled $\overline{x}$. The clause gadget is a triangle of three edges connecting three nodes labeled with the literals appearing in the clause. Do not use the same node in more than one clause gadget. Prove that this reduction works.)

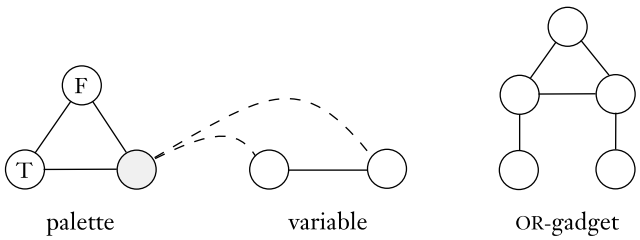
7.26 You are given a box and a collection of cards as indicated in the following figure. Because of the pegs in the box and the notches in the cards, each card will fit in the box in either of two ways. Each card contains two columns of holes, some of which may not be punched out. The puzzle is solved by placing all the cards in the box so as to completely cover the bottom of the box, (i.e., every hole position is blocked by at least one card that has no hole there.) Let $\mathit{CHARADA} = \{\langle c_1, \ldots, c_k\rangle \mid$ each c_i represents a card and this collection of cards has a solution$\}$. Show that *CHARADA* is NP-complete.



7.27 A ***coloring*** of a graph is an assignment of colors to its nodes so that no two adjacent nodes are assigned the same color. Let

$$\mathit{3COLOR} = \{\langle G\rangle \mid \text{the nodes of } G \text{ can be colored with three colors such that no two nodes joined by an edge have the same color}\}.$$

Show that *3COLOR* is NP-complete. (Hint: Use the following three subgraphs.)



7.28 Let *SET-SPLITTING* $= \{\langle S, C\rangle |$ S is a finite set and $C = \{C_1, \ldots, C_k\}$ is a collection of subsets of S, for some $k > 0$, such that elements of S can be colored *red* or *blue* so that no C_i has all its elements colored with the same color.$\}$ Show that *SET-SPLITTING* is NP-complete.

7.29 Consider the following scheduling problem. You are given a list of final exams $F_1, \ldots, F_k$ to be scheduled, and a list of students $S_1, \ldots, S_l$. Each student is taking some specified subset of these exams. You must schedule these exams into slots so that no student is required to take two exams in the same slot. The problem is to determine if such a schedule exists that uses only h slots. Formulate this problem as a language and show that this language is NP-complete.

7.30 This problem is inspired by the single-player game *Minesweeper*, generalized to an arbitrary graph. Let G be an undirected graph, where each node either contains a single, hidden *mine* or is empty. The player chooses nodes, one by one. If the player chooses a node containing a mine, the player loses. If the player chooses an empty node, the player learns the number of neighboring nodes containing mines. (A neighboring node is one connected to the chosen node by an edge.). The player wins if and when all empty nodes have been so chosen.

In the *mine consistency problem* you are given a graph G, along with numbers labeling some of G's nodes. You must determine whether a placement of mines on the remaining nodes is possible, so that any node v that is labeled m has exactly m neighboring nodes containing mines. Formulate this problem as a language and show that it is NP-complete.

R**7.31** In the following solitaire game, you are given an $m \times m$ board. On each of its n^2 positions lies either a blue stone, a red stone, or nothing at all. You play by removing stones from the board so that each column contains only stones of a single color and each row contains at least one stone. You win if you achieve this objective. Winning may or may not be possible, depending upon the initial configuration. Let *SOLITAIRE* $= \{\langle G\rangle |$ G is a winnable game configuration$\}$. Prove that *SOLITAIRE* is NP-complete.

7.32 Let $U = \{\langle M, x, \#^t\rangle |$ MT M accepts input x within t steps on at least one branch$\}$. Show that U is NP-complete.

7.33 Recall, in our discussion of the Church–Turing thesis, that we introduced the language $D = \{\langle p\rangle |$ p is a polynomial in several variables having an integral root$\}$. We stated, but didn't prove, that D is undecidable. In this problem you are to prove a different property of D—namely, that D is NP-hard. A problem is ***NP-hard*** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself. So, you must show that all problems in NP are polynomial time reducible to D.

7.34 A subset of the nodes of a graph G is a ***dominating set*** if every other node of G is adjacent to some node in the subset. Let

$$\textit{DOMINATING-SET} = \{\langle G, k\rangle \mid G \text{ has a dominating set with } k \text{ nodes}\}.$$

Show that it is NP-complete by giving a reduction from *VERTEX-COVER*.

*7.35 Show that the following problem is NP-complete. You are given a set of states $Q = \{q_0, q_1, \ldots, q_l\}$ and a collection of pairs $\{(s_1, r_1), \ldots, (s_k, r_k)\}$ where the s_i are distinct strings over $\Sigma = \{0, 1\}$, and the r_i are (not necessarily distinct) members of Q. Determine whether a AFD $M = (Q, \Sigma, \delta, q_0, F)$ exists where $\delta(q_0, s_i) = r_i$ for each i. Here, $\delta(q, s)$ is the state that M enters after reading s, starting at state q. (Note that F is irrelevant here).

*7.36 Show that if P = NP, a polynomial time algorithm exists that produces a satisfying assignment when given a satisfiable Boolean formula. (Note: The algorithm you are asked to provide computes a function, but NP contains languages, not functions. The P = NP assumption implies that *SAT* is in P, so testing satisfiability is solvable in polynomial time. But the assumption doesn't say how this test is done, and the test may not reveal satisfying assignments. You must show that you can find them anyway. Hint: Use the satisfiability tester repeatedly to find the assignment bit-by-bit.)

*7.37 Show that if P = NP, you can factor integers in polynomial time. (See the note in Problem 7.36.)

R*7.38 Show that if P = NP, a polynomial time algorithm exists that takes an undirected graph as input and finds a largest clique contained in that graph. (See the note in Problem 7.36.)

7.39 In the proof of the Cook–Levin theorem, a window is a 2×3 rectangle of cells. Show why the proof would have failed if we had used 2×2 windows instead.

*7.40 Consider the algorithm *MINIMIZE*, which takes a AFD M as input and outputs AFD M'.

MINIMIZE = "On input $\langle M\rangle$, where $M = (Q, \Sigma, \delta, q_0, A)$ is a AFD:

1. Remove all states of M that are unreachable from the start state.
2. Construct the following undirected graph G whose nodes are the states of M.
3. Place an edge in G connecting every accept state with every nonaccept state. Add additional edges as follows.
4. Repeat until no new edges are added to G:
5. For every pair of distinct states q and r of M and every $a \in \Sigma$:
6. Add the edge (q, r) to G if $(\delta(q, a), \delta(r, a))$ is an edge of G.
7. For each state q, let $[q]$ be the collection of states $[q] = \{r \in Q \mid \text{ no edge joins } q \text{ and } r \text{ in } G\}$.
8. Form a new AFD $M' = (Q', \Sigma, \delta', q_0', A')$ where $Q' = \{[q] \mid q \in Q\}$, (if $[q] = [r]$, only one of them is in Q'), $\delta'([q], a) = [\delta(q, a)]$, for every $q \in Q$ and $a \in \Sigma$, $q_0' = [q_0]$, and $A' = \{[q] \mid q \in A\}$.
9. Output $\langle M'\rangle$."

a. Show that M and M' are equivalent.

b. Show that M' is minimal—that is, no AFD with fewer states recognizes the same language. You may use the result of Problem 1.52 without proof.

c. Show that *MINIMIZE* operates in polynomial time.

7.41 For a cnf-formula ϕ with m variables and c clauses, show that you can construct in polynomial time an AFN with $O(cm)$ states that accepts all nonsatisfying assignments, represented as Boolean strings of length m. Conclude that AFNs cannot be minimized in polynomial time unless P = NP.

*__7.42__ A ***2cnf-formula*** is an E of clauses, where each clause is an OU of at most two literals. Let $2SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 2cnf-formula}\}$. Show that $2SAT \in$ P.

7.43 Modify the algorithm for context-free language recognition in the proof of Theorem 7.16 to give a polynomial time algorithm that produces a parse tree for a string, given the string and a GLC, if that grammar generates the string.

7.44 Say that two Boolean formulas are ***equivalent*** if they have the same set of variables and are true on the same set of assignments to those variables (i.e., they describe the same Boolean function). A Boolean formula is ***minimal*** if no shorter Boolean formula is equivalent to it. Let *MIN-FORMULA* be the collection of minimal Boolean formulas. Show that, if P = NP, then *MIN-FORMULA* $\in$ P.

7.45 The ***difference hierarchy*** $\mathrm{D}_i\mathrm{P}$ is defined recursively as

a. $\mathrm{D}_1\mathrm{P} = \mathrm{NP}$ and

b. $\mathrm{D}_i\mathrm{P} = \{A \mid A = B \setminus C \text{ for } B \text{ in NP and } C \text{ in } \mathrm{D}_{i-1}\mathrm{P}\}$.
(Here $B \setminus C = B \cap \overline{C}$.)

For example, a language in $\mathrm{D}_2\mathrm{P}$ is the difference of two NP languages. Sometimes $\mathrm{D}_2\mathrm{P}$ is called DP (and may be written D^{P}). Let

$$Z = \{\langle G_1, k_1, G_2, k_2\rangle \mid G_1 \text{ has a } k_1\text{-clique and } G_2 \text{ doesn't have a } k_2\text{-clique}\}.$$

Show that Z is complete for DP. In other words, show that every language in DP is polynomial time reducible to Z.

*__7.46__ Let $MAX\text{-}CLIQUE = \{\langle G, k\rangle \mid$ the largest clique in G is of size exactly $k\}$. Use the result of Problem 7.45 to show that *MAX-CLIQUE* is DP-complete.

*__7.47__ Let $f : \mathcal{N} \longrightarrow \mathcal{N}$ be any function where $f(n) = o(n \log n)$. Show that $\mathrm{TIME}(f(n))$ contains only the regular languages.

*__7.48__ Call a regular expression ***star-free*** if it does not contain any star operations. Let $EQ_{\mathsf{SF-REX}} = \{\langle R, S\rangle \mid R \text{ and } S \text{ are equivalent star-free regular expressions}\}$. Show that $EQ_{\mathsf{SF-REX}}$ is in coNP. Why does your argument fail for general regular expressions?

*__7.49__ This problem investigates ***resolution***, a method for proving the unsatisfiability of cnf-formulas. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be a formula in cnf, where the C_i are its clauses. Let $\mathcal{C} = \{C_i \mid C_i \text{ is a clause of } \phi\}$. In a *resolution step* we take two clauses C_a and C_b in $\mathcal{C}$ which both have some variable x, occurring positively in one of the clauses and negatively in the other. Thus $C_a = (x \vee y_1 \vee y_2 \vee \cdots \vee y_k)$ and $C_b = (\overline{x} \vee z_1 \vee z_2 \vee \cdots \vee z_l)$, where the y_i and z_i are literals. We form the new clause $(y_1 \vee y_2 \vee \cdots \vee y_k \ \vee \ z_1 \vee z_2 \vee \cdots \vee z_l)$ and remove repeated literals. Add this new clause to $\mathcal{C}$. Repeat the resolution steps until no additional clauses can be obtained. If the empty clause ( ) is in $\mathcal{C}$ then declare ϕ unsatisfiable.

Say that resolution is ***sound*** if it never declares satisfiable formulas to be unsatisfiable. Say that resolution is ***complete*** if all unsatisfiable formulas are declared to be unsatisfiable.

a. Show that resolution is sound and complete.

b. Use part (a) to show that $2SAT \in$ P.

SOLUÇÕES SELECIONADAS

7.1 **(c)** FALSE; **(d)** TRUE.

7.2 **(c)** TRUE; **(d)** TRUE.

7.15 Let $A \in$ NP. Construct MTN M to decide A in nondeterministic polynomial time.

$M =$ "On input w:

1. Nondeterministically divide w into pieces $w = x_1 x_2 \cdots x_k$.
2. For each x_i, nondeterministically guess the certificates that show $x_i \in A$.
3. Verify all certificates if possible, then *aceite*.
   Otherwise if verification fails, *rejeite*."

7.22 We give a polynomial time mapping reduction from *CLIQUE* to *HALF-CLIQUE*. The input to the reduction is a pair $\langle G, k\rangle$ and the reduction produces the graph $\langle H\rangle$ as output where H is as follows. If G has m nodes and $k = m/2$ then $H = G$. If $k < m/2$, then H is the graph obtained from G by adding j nodes, each connected to every one of the original nodes and to each other, where $j = m - 2k$. Thus H has $m + j = 2m - 2k$ nodes. Observe that G has a k-clique iff H has a clique of size $k + j = m - k$ and so $\langle G, k\rangle \in$ *CLIQUE* iff $\langle H\rangle \in$ *HALF-CLIQUE*. If $k > 2m$, then H is the graph obtained by adding j nodes to G without any additional edges, where $j = 2k - m$. Thus H has $m + j = 2k$ nodes, and so G has a k-clique iff H has a clique of size k. Therefore $\langle G, k_k\rangle \in$ *CLIQUE* iff $\langle H\rangle \in$ *HALF-CLIQUE*. We also need to show *HALF-CLIQUE* $\in$ NP. The certificate is simply the clique.

7.31 First, *SOLITAIRE* $\in$ NP because we can verify that a solution works, in polynomial time. Second, we show that *3SAT* $\leq_{\mathrm{P}}$ *SOLITAIRE*. Given ϕ with m variables $x_1, \ldots, x_m$ and k clauses $c_1, \ldots, c_k$, construct the following $k \times m$ game G. We assume that ϕ has no clauses that contain both x_i and $\overline{x_i}$ because such clauses may be removed without affecting satisfiability.

If x_i is in clause c_j put a blue stone in row c_j, column x_i. If $\overline{x_i}$ is in clause c_j put a red stone in row c_j, column x_i. We can make the board square by repeating a row or adding a blank column as necessary without affecting solvability. We show that ϕ is satisfiable iff G has a solution.

($\rightarrow$) Take a satisfying assignment. If x_i is true (false), remove the red (blue) stones from the corresponding column. So, stones corresponding to true literals remain. Because every clause has a true literal, every row has a stone.

($\leftarrow$) Take a game solution. If the red (blue) stones were removed from a column, set the corresponding variable true (false). Every row has a stone remaining, so every clause has a true literal. Therefore ϕ is satisfied.

7.38 If you assume that P = NP, then *CLIQUE* $\in$ P, and you can test whether G contains a clique of size k in polynomial time, for any value of k. By testing whether G contains a clique of each size, from 1 to the number of nodes in G, you can determine the size t of a maximum clique in G in polynomial time. Once you know t, you can find a clique with t nodes as follows. For each node x of G, remove x and calculate the resulting maximum clique size. If the resulting size decreases, replace x and continue with the next node. If the resulting size is still t, keep x permanently removed and continue with the next node. When you have considered all nodes in this way, the remaining nodes are a t-clique.