Capítulo 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 conforme a quantidade de tempo requerida. Depois disso discutimos a possibilidade de que certos problemas decidíveis requerem quantidades enormes de tempo e como determinar quando você se defronta com tal problema.

7.1 Medindo complexidade

Vamos começar com um exemplo. Tome a linguagem $A = \{\mathtt{0}^k\mathtt{1}^k \mid k \geq 0\}$. Obviamente A é uma linguagem decidível. Quanto tempo uma máquina de uma única fita necessita para decidir A? Examinamos a seguinte MT de uma única fita M_1 para A. Damos a descrição da máquina de Turing em baixo nível, incluindo o próprio movimento 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 o seguinte passo se ambos `0`s e `1`'s permanecem na fita.
3. Faça uma varredura na fita, riscando um único `0` e um único `1`.
4. Se `0`'s ainda permanecem após todos os `1`'s tenham sido riscados, ou se `1`'s ainda permanecem após todos os `0`'s tenham sido riscados, *rejeite*. Caso contrário, se nem `0`'s nem `1`'s permanecem na fita, *aceite*."

Analisamos o algoritmo para a máquina de Turing M_1 que decide A para determinar quanto tempo ele usa.

O número de passos que um algoritmo usa sobre uma entrada específica pode depender de vários parâmetros. Por exemplo, se a entrada é um grafo, o número de passos pode depender do número de nós, do número de arestas, e do grau máximo do grafo, ou alguma combinação desses e/ou outros fatores. Por simplicidade computamos o tempo de execução de um algoritmo puramente como uma função do comprimento da cadeia que representa a entrada e não consideramos qualquer outro parâmetro. Na ***análise do pior-caso***, a forma que consideramos aqui, consideramos o maior tempo de execução de todas as entradas de um comprimento específico. Na ***análise do caso-médio*** consideramos a média de todos os 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 $f(n)$ é o tempo de execução de M, dizemos que M roda em tempo $f(n)$ e que M é uma máquina de Turing de tempo $f(n)$.

Notação O-grande e o-pequeno

Devido ao fato de que o tempo exato de execução de um algoritmo frequentemente é uma expressão complexa, usualmente apenas o estimamos. Em uma forma conveniente de estimativa, chamada ***análise assintótica***, buscamos entender o tempo de execução do algoritmo quando ele é rodado sobre entradas grandes. Fazemos isso considerando somente o termo de mais alta ordem da expressão para o tempo de execução do algoritmo, desconsiderando ambos os coeficientes daquele termo e quaisquer termos de mais baixa ordem, 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 a seguir. Seja $\mathcal{R}^+$ o conjunto dos números reais maiores que 0.

Definição 7.2

Sejam f e g duas funções $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$. Vamos dizer que $f(n) = O(g(n))$ 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 provavelmente vai encontrar têm um termo de mais alta ordem h óbvio. 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 o coeficiente 5 dá $f_1(n) = O(n^3)$.

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

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

Entretanto, $f_1(n)$ não é $O(n^2)$. Independentemente de valores que atribuamos a c e n_0, a definição permanece não-satisfeita nesse caso.

Exemplo 7.4 .....
A notação O-grande interage com logaritmos de uma maneira particular. Usualmente quando utilizamos logaritmos temos que especificar a base, como em $x = \log_2 n$. A base 2 aqui indica que essa igualdade é equivalente à igualdade $2^x = n$. Mudar o valor da base b muda o valor de $\log_b n$ de um fator constante, devido à identidade $\log_b n = \log_2 n / \log_2 b$. Por conseguinte, quando escrevemos $f(n) = O(\log n)$, especificar a base não é mais necessário porque estamos suprimindo fatores constantes de qualquer forma.

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 em 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. Devido ao fato de que o termo $O(n^2)$ domina o termo $O(n)$, essa expressão é equivalente a $f(n) = O(n^2)$. Quando o símbolo O ocorre em um 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 am algumas análises. Usando a identidade $n = 2^{\log_2 n}$ e por conseguinte que $n^c = 2^{c \log_2 n}$, vemos que $2^{O(\log n)}$ representa um limitante superior de n^c para alguma constante 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.

Frequentemente 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 dá uma maneira de se dizer que uma função é assintoticamente ***não mais que uma outra***. Para dizer que uma função é assintoticamente *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 duas funções $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$ 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 $n > n_0$.

Exemplo 7.6 .....
Os seguintes são fáceis 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 que demos para a linguagem $A = \{\mathtt{0}^k\mathtt{1}^k \mid k \geq 0\}$. Repetimo-lo aqui por conveniência.

M_1 = "Sobre a cadeia de entrada w:

1. Faą uma varredura na fita e *rejeite* se um `0` for encontrado à direita de um `1`.
2. Repita o seguinte passo se ambos `0`s e `1`'s permanecem na fita.
3. Faça uma varredura na fita, riscando um único `0` e um único `1`.
4. Se `0`'s ainda permanecem após todos os `1`'s tenham sido riscados, ou se `1`'s ainda permanecem após todos os `0`'s tenham sido riscados, *rejeite*. Caso contrário, se nem `0`'s nem `1`'s permanecem na fita, *aceite*."

Para analisar M_1 consideramos cada um dos seus três estágios separadamente. No estágio 1, a máquina faz uma varredura na fita para verificar se a entrada é da forma $\mathtt{0}^*\mathtt{1}^*$. Realizar essa varredura usa n passos. Reposicionar a cabeça na extremidade esquerda da fita usa outros n passos. Portanto o total usado nesse estágio é $2n$ passos. Em 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. Usar 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 risca um `0` e um `1` em cada varredura. Cada varredura usa $O(n)$ passos. Como cada varredura risca dois símbolos, no máximo $n/2$ varreduras pode 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 $O(n)$.

Por conseguinte o tempo total de M_1 sobre qualquer 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 estabelecer alguma notação para classificar linguagens conforme seus requisitos de tempo.

Definição 7.7 .....
Seja $t : \mathcal{N} \longrightarrow \mathcal{N}$ uma função. Defina a ***classe de complexidade de tempo*** $\mathrm{TIME}(t(n))$, como sendo

$$\mathrm{TIME}(t(n)) = \{L \mid L \text{ é uma linguagem decidida por uma MT de tempo } O(t(n))\}.$$

Lembremo-nos que a linguagem $A = \{\mathtt{0}^k\mathtt{1}^k \mid k \geq 0\}$. A análise precedente mostra que $A \in \mathrm{TIME}(n^2)$ porque M_1 decide A em tempo $O(n^2)$ e $\mathrm{TIME}(n^2)$ 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 pertence a $\mathrm{TIME}(t(n))$ para $t(n) = o(n^2)$? Podemos melhorar o tempo de execução riscando dois `0`'s e dois `1`'s em cada varredura ao invés de somente um pois fazer isso corta o número de varreduras pela metade. Mas isso melhora o tempo de execução somente por um fator de 2 e não afeta o tempo de execução assintótico. A máquina a seguir, M_2, usa um método diferente para decidir A assintoticamente mais rápido. Ela mostra que $A \in \mathrm{TIME}(n \log n)$.

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 os seguintes passos enquanto alguns `0`'s e alguns `1`'s permaneçam na fita.
3. Faça uma varredura na fita, verificando se o número total de `0`'s e `1`'s remanescentes é par ou ímpar. Se for ímpar, *rejeite*.
4. Faça novamente uma varredura na fita, riscando um `0` sim e outro não começando com o primeiro `0`, e então riscando um `1` sim e outro não começando com o primeiro `1`.
5. Se nenhum `0` e nenhum `1` permanece na fita, *aceite*. Caso contrário, *rejeite*."

Antes de analisar M_2, vamos verificar que ela realmente decide A. Em cada varredura realizada no estágio 4, o número total de `0`'s remanescentes é cortado pela metade e qualquer resto é descartado. Por conseguinte, se começássemos com 13 `0`'s, após o estágio 4 ser executado uma única vez somente 6 `0`'s permanecem. Após execuções subseqüentes desse estágio, 3, então 1, e então 0 permanecem. Esse estágio tem o mesmo efeito sobre o número de `1`'s.

Agora examinamos a paridade par/ímpar do número de `0`'s e o número de `1`'s em cada execução do estágio 3. Considere novamente começar com 13 `0`'s e 13 `1`'s. A primeira execução do estágio 3 encontra um número ímpar de `0`'s (porque 13 é um número ímpar) e um número ímpar de `1`'s. Em execuções subseqüentes um número par (6) ocorre, então um número ímpar (3), e um número ímpar (1). Não executamos esse estágio sobre 0 `0`'s ou 0 `1`'s por causa da condição no laço de repetição especificado no estágio 2. Para a seqüência de paridades encontradas (ímpar, par, ímpar, ímpar) se substituirmos as pares por `0`'s e as ímpares por `1`'s e então reverter a seqüência, obtemos `1101`, a representação binária de 13, ou o número de `0`'s e `1`'s 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 `0`'s e `1`'s remanescente é par, ele na verdade está verificando a concordância da paridade de `0`'s com a paridade de `1`'s. Se todas as paridades concordam, a representação binária dos números de `0`'s e `1`'s concordam, e portanto os dois números são iguais.

Para analisar o tempo de execução de M_2, primeiro observamos que todo estágio toma o tempo $O(n)$. Então determinamos o número de vezes que cada um é executado. Estágios 1 e 5 são executados uma vez, levando um total de $O(n)$. O estágio 4 risca pelo menos metade dos `0`'s e `1`'s a cada vez que ele é executado, portanto no máximo $1+\log_2 n$ iterações ocorrem antes que todos estejam riscados. Por conseguinte o tempo total dos estágios 2, 3, e 4 $(1 + \log_2 n)O(n)$, ou $O(n \log n)$. O tempo de execução de M_2 é $O(n) + O(n \log n) = O(n \log n)$.

Antes 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 verdade, qualquer linguagem que pode ser decidida em tempo $o(n \log n)$ numa máquina de Turing de uma única fita é regular, embora não provaremos esse resultado.

Podemos decidir a linguagem A em tempo $O(n)$ (também chamado ***tempo linear***) se a máquina de Turing tem uma segunda fita. A MT M_3 a seguir decide A em tempo linear.

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 `0`'s na Fita 1 até o primeiro `1`. Ao mesmo tempo copie os `0`'s para a Fita 2.
3. Faça uma varredura nos `1`'s na Fita 1 até o final da entrada. Para cada `1` lido na Fita 1, risque um `0` na Fita 2. se todos os `0`'s estão riscados antes que todos os `1`'s sejam lidos, *rejeite*.
4. Se todos os `0`'s foram agora riscados, *aceite*. Se quaisquer `0`'s permanecem, *rejeite*."

A máquina M_3 opera diferentemente das máquinas anteriores para A. Ela simplesmente copia os `0`'s para sua segunda fita e então os emparelha com os `1`'s.

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

Vamos resumir o que mostramos sobre a complexidade de tempo de 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 M_2 que decide A em tempo $O(n \log n)$. Afirmamos (sem prova) 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 sobre uma MT de uma única fita é $O(n \log n)$ e sobre uma MT de duas-fitas é $O(n)$. Note que a complexidade de A depende do modelo de computação escolhido.

Esta discussão levanta 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 equivalentes, ou seja, eles todos decidem a mesma classe de linguagens. Na teoria da complexidade, a escolha do modelo afeta a complexidade de tempo de linguagens. Linguagens que são decidíveis em, digamos, tempo linear em um modelo não são necessariamente decidíveis em tempo linear em um outro.

Na teoria da complexidade, desejamos classificar problemas computacionais conforme a quantidade de tempo requerido para a solução. Mas com qual modelo medimos

o tempo? A mesma linguagem pode ter diferentes requisitos de tempo em diferentes modelos.

Felizmente, requisitos de tempo não diferem largamente para modelos determinísticos típicos. Portanto, se nosso sistema de classificação não é muito sensível a diferenças relativamente pequenas em complexidade, o modelo determinístico escolhido não é crucial. Discutimos essa idéia em mais detalhes nas póximas 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 com uma única fita; a máquina de Turing multi-fita; 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 multi-fita 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 que no Teorema 3.8 mostramos como converter qualquer MT multi-fita 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 multi-fita 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 a MT de k-fitas que roda em tempo $t(n)$. Construimos 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 3.8. Para revisar aquela simulação, relembremo-nos que S usa sua única fita para representar o conteúdo sobre todas as k fitas de M. As fitas são armazenadas consecutivamente, com as posições das cabeças de M marcadas sobre as células apropriadas.

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

Agora analisamos essa simulação. Para cada passo de M, a máquina S faz duas varreduras sobre a parte ativa de sua fita. A primeira obtém a informação necessária para determinar o próximo movimento e a segunda o realiza. O comprimento da parte ativa da fita de S determina quanto tempo S leva para fazer a varredura, portanto temos que determinar um limitante superior sobre esse comprimento. Para fazer isso tomamos a soma dos comprimentos das partes ativas das k fitas de M. Cada uma dessas partes ativas tem comprimento no máximo $t(n)$ pois M usa $t(n)$ células de fita em $t(n)$ passos se a cabeça move para a direita a cada passo e até menos se uma cabeça em

algum momento move para a esquerda. Por conseguinte uma varredura da parte 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 um usa o 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 põe 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. Por conseguinte a simulação inteira de M usa $O(n) + O(t^2(n))$ passos.

Assumimos que $t(n) \geq n$ (uma suposição razoável porque M poderia nem sequer ler toda a entrada 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 em tal máquina é decidível em uma máquina de Turing determinística de uma única fita que requer significativamente mais tempo. Antes de fazer isso, temos que definir o tempo de execução de uma máquina de Turing não-determinística. Relembremo-nos 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 em qualquer ramo de sua computação sobre qualquer entrada de comprimento n, como mostrado na Figura 7.1.

Figura 7.1: Medindo tempo determinístico e não-determinístico

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

Teorema 7.10 . . . . . .
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 determinística de uma única fita equivalente de tempo $2^{O(t(n))}$.

Prova. Seja N uma MT não-determinística rodando em tempo $t(n)$. Construimos uma MT determinística D que simula N como na prova do Teorema 3.10 por meio de uma busca na árvore de computação não-determinística de N. Agora analisamos aquela 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 legítimas dadas pela função de transição de N. Por conseguinte o número total de folhas na árvore é no máximo $b^{t(n)}$.

A simulação procede explorando essa árvore por largura. Em outras palavras, ela visita todos os nós de profundidade d antes de ir adiante para qualquer nó de profundidade $d + 1$. O algoritmo dado na prova do Teorema 3.10 ineficientemente começa na raiz e desce para um nó sempre que ele visita aquele nó, mas eliminar essa ineficiência não altera o enunciado do teorema corrente, portanto deixamo-lo 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 a partir da raiz e descer até um nó é $O(t(n))$. Por conseguinte o tempo de execução de D é $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

Como descrito no Teorema 3.10, a MT D tem três fitas. Convertê-la 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 $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$, e o teorema está provado.

.....

7.2 A classe P .....

Os Teoremas 7.8 e 7.10 ilustram uma importante distinção. Por um lado, provamos uma diferença no máximo quadrática ou *polinomial* entre a complexidade de tempo de problemas medidos sobre máquinas de Turing determinísticas de uma única fita e multi-fitas. Por outro lado, provamos uma diferença no máximo *exponencial* entre a complexidade de tempo de problemas medidos sobre máquinas de Turing determinísticas 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 olhar por que fazer essa separação entre polinômios e exponenciais ao invés de entre algumas outras classes de funções.

Primeiro, note a diferença dramática entre a taxa de crescimento de polinômios que tipicamente ocorrem tal como n^3 e exponenciais que tipicamente ocorrem tal 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 quando resolvemos problemas por meio de busca através de um espaço de soluções, chamado **busca por força-bruta**. Por exemplo, uma maneira de fatorar um número em seus primos constituintes é buscar através de seus potenciais divisores. O tamanho do espaço de busca é exponencial, portanto essa busca usa tempo exponencial. Às vezes, busca por força-bruta pode ser evitada por meio de um entendimento mais profundo de um problema, que pode revelar um algoritmo de tempo polinomial de maior utilidade.

Todos os modelos computacionais determinísticos razoáveis são ***polinomialmente equivalentes***. Ou seja, qualquer um deles pode simular um outro com apenas um acréscimo polinomial em 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 se aproximam bastante dos tempos de execução sobre computadores reais. Por exemplo, o Teorema 7.8 mostra que os modelos de máquina de Turing determinística de uma única fita e multi-fita são polinomialmente equivalentes.

Daqui em diante nos concentramos 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, nosso objetivo é apresentar as propriedades fundamentais da *computação*, ao invés de propriedades de 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 árduo só para fazer seus programas rodar duas vezes mais rápido. Entretanto, desconsideramos fatores constantes há um tempo atrás quando introduzimos notação assintótica. Agora propomos desconsiderar as diferenças polinomiais ainda maiores, tal como aquela entre tempo n e tempo n^3.

Nossa decisão de desconsiderar diferenças polinomiais não implica que consideramos tais diferenças não-importantes. Ao contrário, certamente consideramos a diferença entre tempo n e tempo n^3 como sendo uma diferença importante. Mas algumas questões, tal como a polinomialidade ou não-polinomialidade do problema da fatoração, não dependem de diferenças polinomiais e são importantes também. Simplesmente escolhemos nos concentrar 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 7.11 .....
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 desempenha 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 corresponde aproximadamente à classe de problemas que são solúveis realisticamente em um computador.

Item 1 indica que P é uma classe matematicamente robusta. Ela não é afetada pelos particulares do modelo de computação que estamos usando.

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 é improvável que um tempo de execução de n^{100} seja de algum uso prático. Não obstante, chamar o tempo polinomial de o limiar da solubilidade prática tem provado ser de utilidade. 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 usualmente seguem, frequentemente até o ponto de utilidade prática real.

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. Fazer isso evita detalhes tediosos de fitas e movimentos de cabeça. Precisamos seguir certas convenções ao descrever um algoritmo de modo que possamos analisá-lo com respeito a polinomialidade.

Descrevemos algoritmos com estágios numerados. A noção de um estágio de um algoritmo é análoga a um passo de uma máquina de Turing, embora obviamente, 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) sobre 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 pode ser implementado em tempo polinomial sobre um 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 problemas. Continuamos a usar a notação $\langle \cdot \rangle$ para indicar uma condificação razoável de um ou mais objetos em uma cadeia, sem especificar qualquer método específico de codificação. Agora, um método razoável é aquele que permite codificação e decodificação polinomial de objetos em representações internas naturais ou em outras codificações razoáveis. Métodos de codificação familiares para grafos, autômatos, e semelhantes são todos razoáveis. Mas note que notação unária para codificar números (como no número 17 codificado pela cadeia unária `11111111111111111`) não é razoável porque ela é exponencialmente maior que codificações verdadeiramente razoáveis, tais como a 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 se não existe. Quando analisamos algoritmos sobre grafos, o tempo de execução pode ser computado 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 analisarmos um algoritmo e mostrarmos 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 nós s e t, como mostrado na Figura 7.2. O problema $CAMINHO$ é determinar se um caminho direcionado existe de s para t. Seja

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

Figura 7.2: O problema $CAMINHO$: Existe um caminho de s para t?

Teorema 7.12 .....
$CAMINHO \in$ P.
.....

Idéia da prova. Provamos esse teorema apresentando um algoritmo de tempo polinomial que decide $CAMINHO$. 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 $CAMINHO$ procede examinando todos os caminhos em potencial em G e determinando se qualquer deles é um caminho direcionado de s para t. Um caminho em potencial é uma seqüência 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 em potencial é m^m, 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 $CAMINHO$ você tem que fazer algo que evite a força bruta. Uma maneira é usar um método de busca em grafo tal como busca em largura. Aqui, marcamos sucessivamente todos os nós em G que são atingíveis a partir de s por caminhos direcionados de comprimento 1, então 2, então 3, até m. Limitar o tempo de execução dessa estratégia por um polinômio é fácil.

Prova. Um algoritmo de tempo polinomial M para $CAMINHO$ opera da seguinte maneira:

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

1. Coloque uma marca sobre o nó s.
2. Repita os seguintes passos até que nenhum nó adicional esteja marcado.
3. Faça uma varredura nas 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 polinômio no tamanho de G.

Os estágios 1 e 4 de M são facilmente implementados em tempo polinomial sobre qualquer modelo determinístico razoável. O estágio 3 envolve uma varredura da

entrada e um teste se certos nós estão marcados, o que também é facilmente implementado em tempo polinomial. Portanto M é um algoritmo de tempo polinomial para $CAMINHO$.

.....

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 dos dois seja um número primo em si próprio, enquanto que 10 e 22 não são primos entre si porque ambos são divisíveis por 2. Seja $PRIMENTSI$ o problema de se testar se dois números são primos entre si. Por conseguinte

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

Teorema 7.13 .....
$PRIMENTSI \in \mathrm{P}$.

.....

Idéia da prova. Um algoritmo que resolve esse problema faz uma busca por 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 na notação de qualquer outra base k para $k \geq 2$, é exponencial no comprimento de sua representação. Por conseguinte esse algoritmo de força-bruta busca através de um número exponencial de potenciais divisores e tem um tempo exponencial.

Ao invés disso, resolvemos esse problema com um procedimento numérico antigo, chamado de ***algoritmo euclideano***, para calcular o máximo divisor comum. O ***máximo divisor comum*** de dois números naturais x e y, escrito $\mathrm{mdc}(x, y)$, é o maior inteiro que divide ambos x e y. Por exemplo, $\mathrm{mdc}(18, 24) = 6$. Obviamente, x e y são primos entre si sse $\mathrm{mdc}(x, y) = 1$. Descrevemos o algoritmo euclideano como algoritmo E na prova. Ele usa a função mod, onde $x \bmod y$ é o resto da divisão 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. Faça a atribuição $x \leftarrow x \bmod y$.
3. Intercambie os conteúdos de x e y.
4. Dê como saída x."

O algoritmo R resolve $PRIMENTSI$, 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 o faz também R e portanto precisamos apenas de analisar E por tempo e corretude. A corretude desse algoritmo é bem conhecida portanto não a discutiremos mais aqui.

Para analisar a complexidade de tempo de E, primeiro mostramos que toda execução do estágio 2 (exceto possivelmente o primeiro), corta o valor de x em pelo menos a metade. Após o estágio 2 ser executado, $x < y$ devido à natureza da função $\mod$. Após o estágio 3, $x > y$ porque os dois tiveram seus conteúdos intercambiados. Por conseguinte, quando o estágio 2 é subsequentemente 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 a pelo menos metade em passagens alternadas pelo laço. Por conseguinte o número máximo de vezes que os estágios 2 e 3 são executados é o mínimo entre $\log_2 x$ e $\log_2 y$. 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 total de execução é polinomial.

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

Teorema 7.14
Toda linguagem livre-do-contexto é um membro de P.

Idéia da prova. No Teorema 4.8 provamos que toda LLC é decidível. Para fazer isso demos um algoritmo para cada LLC que a decide. Se aquele algoritmo roda em tempo polinomial, o teorema corrente segue como um corolário. Vamos relembrar aquele algoritmo e descobrir se ele roda rapidamente o suficiente.

Seja L uma LLC gerada por uma GLC G que está na forma normal de Chomsky. Do Problema 2.19, qualquer derivação de uma cadeia w tem $2n - 1$ passos, onde n é o comprimento de w, devido ao fato de G está na forma normal de Chomsky. O decisor para L funciona tentando todas as possíveis derivações com $2n - 1$ passos quando sua entrada é uma cadeia de comprimento n. Se qualquer desses for uma derivação de w, o decisor aceita; se não, 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 poderosa chamada ***programação dinâmica***. Essa técnica usa a acumulação de informação sobre subproblemas menores para resolver problemas maiores. Memorizamos a solucção para qualquer subproblema de tal modo que precisamos resolvê-lo apenas uma vez. Fazemos isso montando uma tabela com todos os subproblemas e entrando com suas soluções sistematicamente à medida que as encontramos.

Neste caso, consideramos os subproblemas de se determinar se cada variável em G gera cada subcadeia de w. O algoritmo entra com a solução para esse 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 na tabela as entradas para cada subcadeia de w. Primeiro ele preenche nas entradas para as subcadeias de comprimento 1, aí então aquelas de comprimento 2, e assim por diante. Ele usa as entradas para comprimentos menores para ajudar a determinar as entradas para comprimentos maiores.

Por exemplo, suponha que o algoritmo já determinou quais variáveis gera todas as subcadeias até o comprimento k. Para determinar se uma variável A gera uma subcadeia específica de comprimento $k+1$ o algoritmo parte aquela subcadeia em dois pedaços não-vazios nas k maneiras possíveis. Para cada divisão, o algoritmo examina cada regra $A \rightarrow BC$ para determinar se B gera o primeiro pedaço e C gera o segundo pedaço, usando as entradas na tabela que já foram computadas. Se ambas B e C geram os respectivos pedaços, A gera a subcadeia e portanto é adicionada à entrada na tabela associada. O algoritmo começa o processo com as cadeias de comprimento 1 examinando a tabela para as regras $A \rightarrow \mathsf{b}$.

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

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

1. Se $w = \varepsilon$ e $S \rightarrow \varepsilon$ é uma regra, *aceite*. ⟦trata o caso $w = \varepsilon$⟧
2. Para $i = 1$ até n, ⟦examina cada subcadeia de comprimento 1⟧
3. Para cada variável A,
4. Teste se $A \rightarrow \mathsf{b}$, é uma regra, onde $\mathsf{b} = w_i$.
5. Se for, coloque A na $tabela(i, i)$.
6. Para $l = 2$ até n, ⟦l é o comprimento da subcadeia⟧
7. Para $i = 1$ até $n - l + 1$, ⟦i é a posição inicial da subcadeia⟧
8. Faça $j = i + l - 1$, ⟦j é a posição final da subcadeia⟧
9. Para $k = i$ até $j - 1$, ⟦k é a posição onde a divisão ocorre⟧
10. Para cada regra $A \rightarrow BC$,
11. Se $tabela(i, k)$ contém B e $tabela(k + 1, j)$ contém C, ponha A na $tabela(i, j)$.
12. Se S está em $tabela(1, n)$, *aceite*. Caso contrário, *rejeite*."

Agora analisamos D. Cada estágio é facilmente implementado para rodar em tempo polinomial. Os estágios 4 e 5 rodam em no máximo nv vezes, onde v é o número de variáveis em G e é uma constante fixa independente de n; daí esses estágios rodam $O(n)$ vezes. O estágio 6 rodam 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. Por conseguinte 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 por força-bruta em muitos problemas e obter soluções de tempo polinomial. Entretanto, tentativas de evitar a força bruta em alguns outros problemas, incluindo muitos problemas interessantes e úteis, não têm sido bem sucedidas, e algoritmos de tempo polinomial que os resolvem não se sabe se existem.

Por que temos tido insucesso em encontrar algoritmos polinomiais para esses problemas? Não sabemos a resposta para essa importante questão. Talvez esses problemas têm algoritmos polinomiais, até agora não descobertos, que repousam sobre princíios desconhecidos. Ou possivelmente alguns desses problemas simplesmente *não* podem ser resolvidos em tempo polinomial. Eles podem ser intrinsecamente difíceis.

Uma descoberta impressionante concernente a essa questão mostra que as complexidades de muitos problemas estão ligadas. A descoberta de um algoritmo de tempo polinomial para um tal problema pode ser usada 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 direcionado 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 específicos, como mostrado na Figura 7.3. Seja

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

Figura 7.3: Um caminho hamiltoniano passa por todo nó exatamente uma vez

Podemos facilmente obter um algoritmo de tempo exponencial para o problema $CAMHAMIL$ modificando o algoritmo por força-bruta para $CAMINHO$ dado no Teorema 7.12. Precisamos apenas adicionar um teste para verificar se o caminho em potencial é hamiltoniano. Ninguém sabe se $CAMHAMIL$ é solúvel em tempo polinomial.

O problema $CAMHAMIL$ realmente tem uma característica chamada ***verificabilidade polinomial*** que é importante para entender sua complexidade. Muito embora não conheçamos uma maneira 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), poderíamos facilmente convencer alguém 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 a propriedade de ser composto. Lembre-se que um número natural é ***composto*** se ele é o produto de dois inteiros maiores que 1 (i.e., um número composto é um número que não é primo). Seja

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

Embora não conheçamos nenhum algoritmo de tempo polinomial para decidir esse problema, podemos facilmente verificar que um número é composto—tudo que é necessário é um divisor daquele número.

Alguns problemas podem não ser verificáveis polinomialmente. Por exemplo, tome $\overline{CAMHAMIL}$, o complemento do problema $CAMHAMIL$. Mesmo se pudéssemos determinar (de alguma forma) que um grafo realmente *não* tem um caminho hamiltoniano, não conheceríamos uma maneira pela qual uma outra pessoa pudesse verificar sua não-existência usando o mesmo algoritmo de tempo exponencial inicialmente usado para fazer a própria determinação da não-existência. Uma definição formal segue.

Definição 7.15 . . . . . .
Um ***verificador*** para uma linguagem A é um algoritmo V, onde

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

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 informação adicional, representada pelo símbolo c na Definição 7.15, para verificar que uma cadeia w é um membro de A. Essa informação é chamada um ***certificado***, ou ***prova***, de 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. Vamos aplicar essa definição às linguagens $CAMHAMIL$ e $COMPOSTOS$.

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

Definição 7.16 . . . . . .
NP é a classe de linguagens que têm verificadores polinomiais.

A classe NP é importante porque ela contém muitos problemas de interesse prático. Da discussão precedente, ambos $CAMHAMIL$ e $COMPOSTOS$ são membros de NP. O termo NP vem do ***tempo polinomial não-determinístico*** e é derivado de uma caracterização alternativa por meio do uso de máquinas de Turing não-determinísticas de tempo polinomial.

O que segue é uma máquina de Turing não-determinística (MTN) que decide o problema $CAMHAMIL$ em tempo polinomial não-determinístico. Lembre-se que na Definição 7.9 definimos o tempo de uma máquina não-determinística como sendo o tempo usado pelo ramo mais longo de computação.

N = "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 não-deterministicamente selecionado como estando 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 um dos dois falha, *rejeite*.

4. Para cada i entre 1 e $m - 1$, verifique se (p_i, p_{i+1}) é uma aresta de G. Se quaisquer não forem, *rejeite*. Caso contrário, todos os testes foram bem sucedidos, portanto *aceite*."

Para analisar esse algoritmo e verificar que ele roda em tempo polinomial não-determinístico, examinamos cada um dos 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 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 7.17

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 e vice-versa. A MT simula o verificador adivinhando o certificado. O verificador simula a MTN usando o ramo de aceitação como o certificado.

Prova. Para a direção de trás para frente deste 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 forma:

$N =$ "Sobre a entrada w de comprimento n,

1. Não-deterministicamente selecione a cadeia c de comprimento 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 é decidida por uma MTN de tempo polinomial N e construa um verificador de tempo polinomial V da seguinte forma:

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

1. Simule N sobre a entrada w, tratando cada símbolo de c como uma descrição da escolha não-determinística a fazer a cada passo (como na prova do Teorema 3.10).
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 7.18

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

Corolário 7.19 . . . . . .
$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$.

A classe NP é insensível à escolha de modelo computacional não-determinístico razoável porque todos tais modelos são polinomialmente equivalentes. Quando estamos descrevendo e analisando algoritmos de tempo polinomial não-determinístico seguimos as convenções precedentes para algoritmos de tempo polinomial determinístico. Cada estágio de um algoritmo de tempo polinomial não-determinístico tem que 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 cada dois nós são conectados por uma aresta. Um ***k-clique*** é um clique que contém k nós. A Figura 7.4 ilustra um grafo tendo um 5-clique.

Figura 7.4: 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 7.20 . . . . . .
$CLIQUE$ está em NP.

. . . . . .
Idéia da prova. O clique é o certificado.

Prova. O seguinte é 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 passam, *aceite*; caso contrário, *rejeite*."

Prova alternativa. Se você prefere pensar em NP em termos de máquinas de Turing de tempo polinomial não-determinístico, você pode provar esse teorema fornecendo um 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 os nós em c.

3. Se sim, *aceite*; caso contrário, *rejeite*."

. . . . . .

A seguir consideramos o problema $SOMA\text{-}SUBCONJ$ concernente à aritmética inteira. 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 cuja soma é t. Por conseguinte

$$SOMA\text{-}SUBCONJ = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ e para algum } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ temos } \textstyle\sum y_i = t\}.$$

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

Teorema 7.21 . . . . . .
$SOMA\text{-}SUBCONJ$ está em NP.

. . . . . .

Idéia da prova. O subconjunto é o certificado.

Prova. O seguinte é um verificador V para $SOMA\text{-}SUBCONJ$.

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

1. Teste se c é uma coleção de números cuja soma é t.

2. Teste se S contém todos os números em c.

3. Se ambos passam, *aceite*; caso contrário, *rejeite*."

Prova alternativa. Podemos também provar esse teorema dando uma máquina de Turing de tempo polinomial não-determinístico para $SOMA\text{-}SUBCONJ$ 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 cuja soma é t.

3. Se o teste passa, *aceite*; caso contrário, *rejeite*."

. . . . . .

Observe que os complementos desses conjuntos, $\overline{CLIQUE}$ e $\overline{SOMA\text{-}SUBCONJ}$, não são obviamente membros de NP. Verificar que algo *não* está presente parece ser mais difícil que verificar que *está* presente. Montamos 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 questão P versus NP

Como temos estado dizendo, NP é a classe de linguagens que são solúveis em tempo polinomial numa máquina de Turing não-determinística, ou, equivalentemente, é a classe de linguagens na qual pertinência na linguagem pode ser verificada em tempo polinomial. P é a classe de linguagens nas quais pertinência pode ser testada em tempo polinomial. Resumimos essa informação da seguinte forma, onde nos referimos frouxamente a solúvel em tempo polinomial como solúvel "rapidamente."

P = a classe de linguagens nas quais pertinência pode ser *decidida* rapidamente.
NP = a classe de linguagens nas quais pertinência pode ser *verificada* rapidamente.

Apresentamos exemplos de linguagens, tais como $CAMHAMIL$ 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 de decidibilidade polinomial. Mas, por mais difícil que seja de imaginar, P e NP poderiam ser iguais. Estamos incapacitados 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 ainda não resolvidos em ciência da computação teórica e matemática contemporânea. Se essas classes fossem iguais, qualquer problema polinomialmente verificável seria polinomialmente decidível. A maioria dos pesquisadores acreditam que as duas classes não são iguais porque as pessoas têm investido enormes esforços para encontrar algoritmos de tempo polinomial para certos problemas em NP, sem sucesso. Pesquisadores também têm tentado provar que as classes são desiguais, mas isso acarretaria mostrar que nenhum algoritmo rápido existe para substituir a busca por força-bruta. Fazer isso está atualmente além do alcance científico. A Figura 7.5 mostra as duas possibilidades.

Figura 7.5: 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-completude . . . . . .

Um importante avanço na questão P versus NP veio no início dos anos 1970's 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 algum desses problemas, todos os problemas em NP seriam solúveis em tempo polinomial. Esses problemas são chamados ***NP-completos***. O fenômeno de NP-completude é importante tanto por razões práticas quanto teóricas.

No lado teórico, um pesquisador tentando mostrar que P é diferente de NP pode se concentrar num problema NP-completo. Se qualquer problema em NP requer mais

que tempo polinomial, um NP-completo também requer. Ademais, um pesquisador tentando provar que P é igual a NP precisa somente 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 o desperdício de tempo na busca por um algoritmo de tempo polinomial não existente para resolver um problema específico. Muito embora possivelmente não tenhamos 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 o ***problema da satisfatibilidade***. Lembre-se que variáveis que podem tomar valores VERDADEIRO e FALSO são chamadas ***variáveis booleanas*** (veja a Seção 0.2). Usualmente, representamos VERDADEIRO por 1 e FALSO por 0. As ***operações booleanas*** E, OU, e NÃO, representadas pelo símbolos $\wedge$, $\vee$, e $\neg$, respectivamente, são descritas na lista abaixo. Usamos a barra superior como uma abreviação para o símbolo $\neg$, portanto $\bar{x}$ significa $\neg x$.

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

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

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

é uma fórmula booleana. Uma fórmula booleana é ***satisfatível*** se alguma atribuição de 0's e 1's às variáveis faz com que o valor da fórmula seja 1. A fórmula precedente é satisfatível porque a atribuição $x = 0$, $y = 1$, e $z = 0$ faz com que ϕ tenha valor 1. Dizemos que a atribuição *satisfaz* ϕ. O ***problema da satisfatibilidade*** é testar se uma fórmula booleana é satisfatível. Seja

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

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

Teorema 7.22 .....
Teorema de Cook–Levin $SAT \in$ P sse P=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 outro. Quando o problema A se reduz ao problema B, uma solução para B pode ser usada para resolver A. Agora definimos uma versão de redubilidade que leva em conta 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 7.23 .....
Uma função $f : \Sigma^* \longrightarrow \Sigma^*$ é uma ***função computável em tempo polinomial*** se alguma máquina de Turing de tempo polinomial M existe que pára com exatamente $f(w)$ sobre sua fita, quando iniciada com qualquer entrada w.

Definição 7.24 .....
A linguagem A é ***redutível por mapeamento em tempo polinomial***,[1] em alguns outros livros-texto. ou simplesmente ***redutível em tempo polinomial***, à linguagem B, escrito $A \leq_{\mathrm{P}} B$, se uma função computável em tempo polinomial $f : \Sigma^* \longrightarrow \Sigma^*$ existe, onde para toda w, $w \in A \iff f(w) \in B$. A função f é chamada a ***redução de tempo polinomial*** de A para B.

Redutibilidade em tempo polinomial é o análogo eficiente à redutibilidade por mapeamento, como definido na Seção 5.3. Outras formas de redutibilidade eficiente estão disponíveis, mas redutibilidade polinomial é uma forma simples que é adequada para nossos propósitos portanto não discutiremos as outras aqui. A Figura 7.6 ilustra a redutibilidade em tempo polinomial.

Figura 7.6: Função de tempo polinomial f reduzindo A para B

Como na redução por mapeamento comum, uma redução em tempo polinomial de A para B provê uma maneira de converter teste de pertinência em A para 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 testamos se $f(w) \in B$.

Se uma linguagem é redutível em tempo polinomial para uma linguagem para a qual já se sabe que tem uma solução polinomial, obtemos uma solução polinomial para a linguagem original, como no seguinte teorema.

Teorema 7.25 .....
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 em 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 $f(w)$.

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

Se $w \in A$, então $f(w) \in B$ 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 dos seus dois estágios roda em tempo polinomial. Note que o estágio 2 roda em tempo polinomial porque a composição de dois polinômios é um polinômio.
.....

[1] É chamada redutibilidade muitos-para-um em tempo polinomial

Antes de demonstrar uma redução em 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 $\bar{x}$. Uma ***cláusula*** é vários literais conectados com $\vee$'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 com $\wedge$'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 têm 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 $3SAT = \{\langle\phi\rangle \mid \phi$ é uma 3fnc-fórmula satisfatível$\}$. Em uma fnc-fórmula, cada cláusula tem que conter pelo menos um literal que recebe o valor 1.

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

Teorema 7.26 .....
$3SAT$ é redutível em tempo polinomial para $CLIQUE$.

.....

Idéia da prova. A redução em tempo polinomial f que demonstramos de $3SAT$ para $CLIQUE$ converte fórmulas em grafos. Nos grafos construídos, cliques de um tamanho especificado correspondem a atribuições que satisfazem à fórmula. Estruturas dentro do grafo são projetadas para reproduzir o comportamento das variáveis e das 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 como segue.

Os nós em 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 7.7 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 7.7: O grafo que a redução produz a partir de $\phi = (x_1 \vee x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2 x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

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

Suponha que ϕ tenha uma atribuição que a satisfaz. Nessa atribuição, pelo menos um literal é verdadeiro em toda cáusula. Em cada tripla de G, selecionamos um nó correspondendo a um literal verdadeiro na atribuição que satisfaz a fórmula. Se mais

de um literal é verdadeiro em uma cláusula específica, escolhemos um dos literais verdadeiros arbitrariamente. Os nós selecionados formam um k-clique. O número de nós selecionados 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 numa das exceções descritas anteriormente. Eles não poderiam ser da mesma tripla porque selecionamos apenas um nó por tripla. Eles não poderiam term rótulos contraditórios porque os literais associados eram ambos verdadeiros na atribuição que satisfaz a fórmula. Por conseguinte G é um k-clique.

Suponha que G tenha um k-clique. Não é o caso que dois dos nós do clique ocorrem na mesma tripla porque nós na mesma tripla não estão conectados por arestas. Por conseguinte cada uma das k triplas contém exatamente um dos nós do k clique. Atribuimos valores-verdade às variáveis de ϕ de modo que cada literal rotulando um nó do clique é tornado 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 que é atribuído VERDADEIRO. Por conseguinte ϕ é satisfatível.

.....

Os teoremas 7.25 e 5.26 nos dizem que, se $CLIQUE$ for solúvel em tempo polinomial, então $3SAT$ também o é. À primeira vista, essa conexão entre esses dois problemas parece um tanto notável porque, superficialmente, eles são bastante diferentes. Mas redutibilidade em tempo polinomial nos permite ligar suas complexidades. Agora nos voltamos para uma definição que nos permitirá ligar as complexidades de uma classe inteira de problemas.

Definição de NP-completude

Definição 7.27 .....
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 7.28 .....
Se B é NP-completa e $B \in \mathrm{NP}$, então P=NP.

Prova. Esse teorema segue diretamente da definição de redutibilidade em tempo polinomial.

.....

Teorema 7.29 .....
Se B é NP-completa e $B \leq_{\mathrm{P}} C$ para C em NP, então C é NP-completa.

Prova. Já sabemos que C está em NP, portanto temos que mostrar que toda A que está em NP é redutível em tempo polinomial a C. Devido ao fato de que B é 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 compõem; ou seja, se A é redutível em tempo polinomial a C e C é redutível em tempo polinomial

a B, então A é redutível em tempo polinomial a B. Daí 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 em tempo polinomial a partir do primeiro. Entretanto, estabelecer o primeiro problema NP-completo é mais difícil. Agora fazemos isso provando que SAT é NP-completo.

Teorema 7.30
SAT é NP-completo.[2]

Esse teorema reenuncia o Teorema 7.22, o teorema de Cook–Levin, de uma outra forma.

Idéia da prova. Mostrar que SAT está em NP é fácil, e fazemos isso logo mais. A parte difícil da prova é mostrar que qualquer linguagem em NP é redutível em tempo polinomial a SAT.

Para fazer isso construimos uma redução em tempo polinomial para cada linguagem A em NP para SAT. A redução para A toma uma cadeia w e produz uma fórmula booleana ϕ que simula a máquina NP para A sobre a entrada w. Se a máquina aceita, ϕ tem uma atribuição que satisfaz correspondendo à computação de aceitação. Se a máquina não aceita, nenhuma atribuição satisfaz ϕ. Por conseguinte, 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 tenhamos que 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. Portanto, o fato de que podemos desenhar 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 somente aqueles leitores interessados em detalhes devem se preocupar com esse pequeno detalhe.) 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 mostrado na Figura 7.8. Por conveniência mais adiante assumimos que cada configuração começa e termina com um símbolo #, portanto a primeira e a última coluna 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 tabelau é de ***aceitação*** se qualquer linha do tableau é uma configuração de aceitação.

Figura 7.8: Um tableau é uma tabela $n^k \times n^k$ de configurações

Todo tableau de aceitação para N sobre w corresponde a um ramo da computação de N sobre w. Por conseguinte 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 chamado de uma ***célula***. A célula na linha i e coluna j é chamada $celula[i,j]$ 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 $celula[i,j]$ contém um s.

Agora desenhamos ϕ de modo que uma atribuição às variáveis que satisfaça a fórmula corresponde a um tableau de aceitação para N sobre w. A fórmula ϕ é o E das quatro partes $\phi_{\text{celula}} \wedge \phi_{\text{inicio}} \wedge \phi_{\text{movimento}} \wedge \phi_{\text{aceita}}$ e descrevemos cada um por vez.

Como mencionamos anteriormente, ligar uma variável $x_{i,j,s}$ corresponde a colocar um símbolo s em $celula[i,j]$. A primeira coisa que temos que 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} assegura esse requisito expressando-o em termos de operações booleanas:

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

Os símbolos $\bigwedge$ e $\bigvee$ significam iteração de E's e OU's respectivamente. Por exemplo, o fragmento da fórmula anterior

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

é uma abreviação de

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

onde $C = \{s_1, s_2, \ldots, s_l\}$. Portanto, ϕ_{celula} é na verdade 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 ligada) na célula correspondente. Esses fragmentos são conectados por operações $\wedge$.

A primeira parte de ϕ_{celula} dentro dos colchetes 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

[2] Uma prova alternativa desse teorema aparece na Seção 9.3 na página 321.

variáveis que satisfaz ϕ e por conseguinte ϕ_{celula} tem que ter exatamente uma variável ligada para toda célula. Por conseguinte qualquer atribuição que satisfaz a fórmula especifica um símbolo em cada célula da tabela. As partes ϕ_{inicio}, $\phi_{\text{movimento}}$, e ϕ_{aceita} asseguram que a tabela é na verdade um tableau de aceitação da seguinte forma.

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

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

A fórmula ϕ_{aceita} garante que uma configuração de aceitação ocorre no tableau. Ela assegura 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:

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

Finalmente, a fórmula $q_{\text{movimento}}$ garante que cada linha da tabela corresponde a uma configuração que legalmente segue da configuração da linha precedente conforme as regras de N. Ela faz isso assegurando que cada janela 2×3 de células é legal. Dizemos que uma janela 2×3 é *legal* se aquela 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 são membros do alfabeto de fita e q_1 e q_2 são estados de N. Assuma que quando no estado q_1 com a cabeça lendo um a, N escreve um b, permanece no estado q_1 e move para a direita, e que quando no estado q_1 com a cabeça lendo um b, N não-deterministicamente

1. escreve um c, vai para q_2 e move para a esquerda, ou

2. escreve um a, vai para q_2 e move para a direita.

Expressando formalmente, $\delta(q_1, \mathsf{a}) = \{(q_1, \mathsf{b}, \mathrm{D})\}$ e $\delta(q_1, \mathsf{b}) = \{(q_2, \mathsf{c}, \mathrm{E}), (q_2, \mathsf{a}, \mathrm{D})\}$. Exemplos de janelas legais para essa máquina são mostrados na Figura 7.9.

(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 7.9: Exemplos de janelas legais

[3] Poderíamos dar uma definição precisa de ***janela legal*** aqui, em termos da função de transição. Mas fazer isso é bastante entendiante e seria uma distração da principal linha da argumentação. Qualquer um que deseje mais precisão deve se remeter à análise relacionada na prova do Teorema 5.11, a indecidibilidade do Problema da Correspondência de Post.

Na Figura 7.9, as janelas (a) e (b) são legais porque a função de transição permite que N se move da maneira indicada. A janela (c) é legal 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 poderia 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 linhas superior e inferior são idênticas, o que ocorreria se a cabeça não estivesse adjacente à localização da janela. Note que um # pode aparecer na esquerda ou na direita de ambas as 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 ela teria então movido para a esquerda no estado q_2 para aparecer na extremidade direita da linha inferior. Finalmente, a janela (f) é legal porque o estado q_1 poderia ter estado imediatamente à esquerda da linha superior e ela poderia ter modificado o b para um c e movido para a esquerda.

As janelas mostradas na Figura 7.10 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 7.10: Exemplos de janelas ilegais

Na janela (a) o símbolo central na linha superior não pode modificar 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 7.31 . . . . . .
Se a linha superior da tabela é a configuração inicial e toda janela na tabela é legal, cada linha da tabela é uma configuração que segue legalmente a anterior.

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 estados. Por conseguinte aquele símbolo tem que aparecer o mesmo na célua central inferior da janela. Por conseguinte ele aparece na mesma posição na configuração inferior.

A janela contendo o símbolo de estado na célula central superior garante que as três posições correspondentes são atualizadas consistentemente com a função de transição. Por conseguinte, se a configuração superior é uma configuração legal, assim o é a configuração inferior, e a inferior segue a superior conforme as regras de N. Note que esta prova, embora simples, depende crucialmente da nossa escolha de um tamanho de janela de 2×3, como mostra o Exercício 7.32.

Agora voltamos à construção de $\phi_{\text{movimento}}$. Ela estipula que todas as janelas no tableau são legais. Cada janela contém seis células, que podem ser arranjadas de um número fixo de maneiras para produzir uma janela legal. A fórmula $\phi_{\text{movimento}}$ diz que os arranjos daquelas células tem que ser uma dessas maneiras, ou

$$\phi_{\text{movimento}} = \bigwedge_{1<i\leq n^k, 1<j<n^k} \text{(a janela } (i,j) \text{ é legal)}$$

Substituimos o texto "a janela (i,j) é legal" nessa fórmula pela fórmula a seguir. Escrevemos o conteúdo das 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. Lembre-se que o tableau é uma tabela de $n^k \times n^k$, portanto ele contém n^{2k} células. Cada célula tem l variáveis associadas a ela, o número total de variáveis é $O(n^{2k})$.

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 $\phi_{\text{movimento}}$ e ϕ_{aceita} cada uma contém um fragmento de tamanho-fixo da fórmula para cada célula do tableau, portanto seu tamanho é $O(n^{2k})$. Por conseguinte o tamanho total de ϕ é $O(n^{2k})$. Esse resultado é bom porque o tamanho de ϕ é polinomial em n. Se ele fosse mais que polinomial, a redução não teria qualquer chance de gerá-la em tempo polinomial. (Na realidade nossas estimativas são baixas por um fator de $O(\log n)$ porque cada variável tem índices que podem variar até n^k e portanto podem requerer $O(\log n)$ símbolos para escrever nas fórmulas, 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. Por conseguinte podemos facilmente construir uma redução que produz ϕ em tempo polinomial a partir da entrada w.

. . . . . .

Portanto concluimos a prova do teorema de Cook–Levin, mostrando que a linguagem SAT é NP-completa. Mostrar a NP-completude de outras linguagens geralmente não requer uma prova tão longa. Ao invés disso, NP-completude pode ser provada com uma redução em tempo polinomial a partir de uma linguagem que já é conhecida ser NP-completa. Podemos usar SAT para esse propósito, mas usar $3SAT$, o caso especial de SAT que definimos na página 251, é usualmente mais fácil. Lembre-se 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 isso como um corolário do Teorema 7.30.

Corolário 7.32 . . . . . .
$3SAT$ é NP-completa.

Prova. Obviamente $3SAT$ está em NP, portanto só precisamos provar que todas as linguagens em NP se reduzem a $3SAT$ em tempo polinomial. Uma maneira de fazer isso é mostrar que SAT se reduz em tempo polinomial a $3SAT$. Ao invés disso, modificamos a prova do Teorema 7.30 de modo que ela produza diretamente uma fórmula na forma normal conjuntiva com três literais por cláusula.

O Teorema 7.30 produz uma fórmula que já está quase na forma normal conjuntiva. A fórmula ϕ_{celula} é um grande E de subfórmulas, cada uma das quais contém um grande OU e um grande E de OU's. Por conseguinte ϕ_{celula} é um E de cláusulas e portanto 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 portanto é uma única cláusula. A fórmula $\phi_{\text{movimento}}$ é a única que não já está na fnc, mas podemos facilmente convertê-la em uma fórmula que está na fnc da seguinte maneira.

Lembre-se que $\phi_{\text{movimento}}$ é um grande E de subfórmulas, cada uma das quais é um OU de E's que descreve todas as possíveis janelas legais. As leis de distributividade, conforme descritas no Capítulo 0, enunciam que podemos substituir um OU de E's por um E de OU's equivalente. Fazendo isso aumenta significativamente o tamanho de cada subfórmula, mas só pode aumentar o tamanho total de $\phi_{\text{movimento}}$ por um fator constante porque o tamanho de cada subfórmula depende somente de N. O resultado é uma fórmula que está na forma normal conjuntiva.

Agora que escrevemos a fórmula na fnc, convertemo-la para uma com três literais por cláusula. Em cada cláusula que no momento 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 adicionais para preservar a satisfatibilidade ou não-satisfatibilidade 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 composta 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 dos a_i's satisfaz a cláusula original, podemos encontrar alguma valoração de z de modo que as duas novas cláusulas são 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 se e somente se a fórmula original o era, portanto a prova está completa.

. . . . . .

7.5 Problemas NP-completos adicionais . . . . . .

Nesta seção apresentamos teoremas adicionais mostrando que várias linguagens são NP-completas. Nossa estratégia geral é exibir uma redução em 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 construímos uma redução em tempo polinomial 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 *dispositivos*. Por exemplo, na redução de $3SAT$ para $CLIQUE$ apresentada no Teorema 7.26, os nós individualmente simulam variáveis e triplas de nós simulam cláusulas. Um nó individualmente pode ou não ser um membro do clique, o que corresponde a uma variável que pode ou ser verdadeira na atribuição que satisfaz a fórmula. Cada cláusula tem que conter um literal que tem o valor-verdade 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 corolário do Teorema 7.26 a seguir enuncia que $CLIQUE$ é NP-completa.

Corolário 7.33 .....
$CLIQUE$ é 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 pelo tamanho da menor cobertura de vértices. Seja

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

Teorema 7.34 .....
$COBERT\text{-}VERT$ é NP-completa.

Prova. Damos uma redução de $3SAT$ para $COBERT\text{-}VERT$ que opera em tempo polinomial. A redução mapeia uma fórmula booleana ϕ num grafo G e um valor k. Cada aresta em G tem que tocar em pelo menos um nó na cobertura de vértices, de modo que um dispositivo natural para uma variável é uma única aresta. Fazendo aquela variável ser VERDADEIRO corresponde a selecionar o nó esquerdo para a cobertura de vértices, enquanto que fazendo-a ser FALSO corresponde ao nó direito. Rotulamos os dois nós no dispositivo para a variável x como x e $\overline{x}$.

Os dispositivos para as cláusulas são um pouco mais complexos. Cada dispositivo de cláusula é uma tripla de três nós que são rotulados com os três literais da cláusula. Esses três nós são conectados uns aos outros e aos nós nas variáveis-dispositivos que têm 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 7.11.

Figura 7.11: 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

O problema do caminho hamiltoniano

Lembre-se 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 7.35 . . . . . .
$CAM\text{-}HAMIL$ é NP-completo.

. . . . . .

Idéia da prova. Para mostrar que $CAM\text{-}HAMIL$ é NP-completo temos que demonstrar duas coisas: (1) que $CAM\text{-}HAMIL$ está em NP; e (2) que toda linguagem A em NP é redutível em tempo polinomial a $CAM\text{-}HAMIL$. O primeiro fizemos na Seção 7.3. Para fazer o segundo mostramos que um problema NP-completo conhecido, $3SAT$, é redutível em tempo polinomial a $CAM\text{-}HAMIL$. Damos uma maneira de converter 3fnc-fórmulas em grafos nos quais caminhos hamiltonianos correspondem a atribuições que satisfazem a fórmula. Os grafos contêm dispositivos que imitam variáveis e cláusulas. O dispositivo de variáveis é uma estrutura em forma de losango que pode ser percorrida de duas maneiras, correspondendo às duas atribuições de valor-verdade. O dispositivo de cláusulas é um nó. Assegurar que o caminho passa por cada dispositivo de cláusula corresponde a assegurar que cada cláusula é satisfeita na atribuição que satisfaz a fórmula.

Prova. Anteriormente mostramos que $CAM\text{-}HAMIL$ está em NP, portanto tudo o que resta ser feito é mostrar que $3SAT \leq_{\mathrm{P}} CAM\text{-}HAMIL$. 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 3cnf-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 em um grafo G. O grafo G que construímos tem várias partes para representar as estruturas (variáveis e cláusulas) que aparecem em ϕ.

Representamos cada variável x_i com um estrutura em formato de losango que contém uma linha horizontal de nós, como mostrado na Figura 7.12. Mais adiante especificamos o número de nós que aparecem na linha horizontal.

Figura 7.12: Representando a variável x_i como uma estrutura em forma de losango

Representamos cada cláusula de ϕ como um único nó, da seguinte maneira:

Figura 7.13: Representando a cláusula c_j como um nó

A Figura 7.14 mostra a estrutura global de G. Ela mostra todos os elementos de G e seus relacionamentos, exceto as arestas que representam o relacionamento das variáveis às cláusulas que as contêm.

Figura 7.14: A estrutura de alto-nível de G

A seguir mostramos como conectar os losangos representando as variáveis aos nós representando as cláusulas. Cada estrutura em forma de losango contém uma linha horizontal de nós conectados por arestas indo em ambas as direções. A linha horizontal contém $3k + 1$ nós além dos dois nós nas extremidades pertencentes ao losango. Esses nós são agrupados em pares adjacentes, um para cada cláusula, com nós separadores extra em seguida aos pares, como mostrado na Figura 7.15.

Figura 7.15: Os nós horizontais em uma estrutura em forma de losango

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

Figura 7.16: As arestas adicionais quando a cláusula c_j contém x_i

Se $\overline{x_i}$ aparece na cláusula c_j, adicionamos as duas arestas seguintes do j-ésimo par no i-ésimo losango para o j-ésimo nó cláusula.

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.

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 losango a cada vez, e termina em t. Para atingir os nós horizontais em um losango, o caminho ou faz um zigue-zague da esquerda para a direita ou um zigue-zague através do losango correspondente. Se x_i recebe o valor VERDADEIRO, faça um zigue-zague através do losango correspondente. Se x_i recebe o valor FALSO, faça um zigue-zague. Mostramos ambas as possibilidades na Figura 7.18.

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, selecione um dos literais que recebem o valor VERDADEIRO pela atribuição que satisfaz a fórmula.

Se selecionamos x_i na cláusula c_j, podemos desviar no j-ésimo par no i-ésimo losango. Fazer isso é possível porque x_i tem que ser VERDADEIRO, portanto o caminho faz um zigue-zague da esquerda para a direita através do losango correspondente. Daí as arestas para o nó c_j estão na ordem correta para permitir um desvio e um retorno.

Figura 7.17: As arestas adicionais quando a cláusula c_j contém $\overline{x_i}$

Figura 7.18: Fazendo zigue-zague e zigue-zagueando através de um losango, conforme determinado pela atribuição que satisfaz

Igualmente, se selecionamos $\overline{x_i}$ na cláusula c_j, podemos desviar no j-ésimo par no i-ésimo losango. Fazer isso é possível porque x_i tem que ser FALSO, portanto o caminho zigue-zagueia da direita para a esquerda através do losango correspondente. Daí as arestas para o nó c_j estão novamente na ordem correta para permitir um desvio e um retorno. (Note que cada literal verdadeiro em uma cláusula fornece uma *opção* de um desvio para atingir o nó cláusula. Como um resultado, se diversos literais em uma cláusula são verdadeiros, somente um desvio é tomado.) Por conseguinte construimos o caminho hamiltoniano desejado.

Para a direção contrária, se G tem um caminho hamiltoniano de s para t, exibimos uma atribuição que satisfaz ϕ. Se o caminho hamiltoniano é *normal*, ou seja, passa pelos losangos na ordem e primeiro o de cima depois o de baixo, exceto para os desvios para os nós cláusula, podemos facilmente obter a atribuição que satisfaz a fórmula. Se o caminho zigue-zagueia através do losango, atribuímos à variável correspondente o valor VERDADEIRO, e, se o caminho faz um zigue-zague, atribuímos FALSO. Devido ao fato de que cada nó cláusula aparece no caminho, por meio da observação do losango no qual o desvio para ele é tomado, podemos determinar qual dos literais na cláusula é VERDADEIRO.

Tudo o que resta ser feito é mostrar que um caminho hamiltoniano tem que ser normal. A única maneira da normalidade falhar seria para o caminho entrar numa cláusula a partir de um losango mas retornar para um outro, como na Figura 7.19. O caminho vai do nó a_1 para c, mas ao invés de retornar a a_2 ou a_3 tem que ser um nó separador. Se a_2 fosse um nó separador, 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 estariam no mesmo par de cláusulas, e portanto as únicas arestas entrando em a_2 seriam de a_1, a_3, e c. Em qualquer caso, o caminho não poderia conter o nó a_2. 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 sair de a_2 via a_3. Portanto um caminho hamiltoniano tem que ser normal. Essa redução obviamente opera em tempo polinomial e a prova está completa.

Figura 7.19: Essa situação não pode ocorrer

. . . . . .

A seguir consideramos uma versão não-direcionada do problema do caminho hamiltoniano, chamado $CAM\text{-}HAMIL\text{-}ND$. Para mostrar que $CAM\text{-}HAMIL\text{-}ND$ é NP-completo damos uma redução em tempo polinomial a partir da versão direcionada do problema.

Teorema 7.36 . . . . . .
$CAM\text{-}HAMIL\text{-}ND$ 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 se e somente se G' tem um caminho hamiltoniano de s' para t'. Descrevemos G' da seguinte maneira.

Cada nó u de G, exceto s e t, é substituído por uma tripla de nós u^{entra}, u^{meio} e u^{sai} em G'. Os nós s e t em G são substituídos pelos nós s^{sai} e t^{entra} em G'. Arestas de dois tipos aparecem em G'. Primeiro, 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 se e somente se 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 correspondente P' em G',

$$s^{\text{sai}}, u_1^{\text{entra}}, u_1^{\text{meio}}, u_1^{\text{sai}}, u_2^{\text{entra}}, u_2^{\text{meio}}, u_2^{\text{sai}}, \ldots, t^{\text{entra}}.$$

Para mostrar a outra direção, afirmamos que qualquer caminho hamiltoniano em G' de s^{sai} para t^{entra} em G' tem que ir de uma tripla de nós para uma tripla de nós, exceto o início e o fim, como faz o caminho P' que acabamos de descrever. Isso completaria a prova porque qualquer caminho desse 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 tem que ser u_i^{entra} para algum i porque somente aqueles nós estão conectados a s^{sai}. O nó seguinte tem que 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 nó ao qual u_i^{meio} está conectado. O nó seguinte tem que ser u_j^{entra} para algum j porque nenhum outro nó disponível está conectado a u_i^{sai}. O argumento então repete até que t^{sai} seja atingido.

.....

O problema da soma de subconjuntos

Retomemos o problema $SOMA\text{-}SUBCONJ$ definido na página 246. Naquele problema, nos é dada uma coleção de números, $x_1, \ldots, x_k$ juntamente com um número alvo t, e deseja-se determinar se a coleção contém uma subcoleção cuja soma é t. Agora mostramos que esse problema é NP-completo.

Teorema 7.37 .....
$SOMA\text{-}SUBCONJ$ é NP-completo.

.....

Idéia da prova. Já mostramos que $SOMA\text{-}SUBCONJ$ está em NP no Teorema 7.21. Provamos que todas as linguagens em NP são redutíveis em tempo polinomial a $SOMA\text{-}SUBCONJ$ reduzindo a linguagem NP-completa $3SAT$ a ele. Dada uma 3fnc-fórmula ϕ construimos uma instância do problema $SOMA\text{-}SUBCONJ$ que contém uma subcoleção cuja soma é t se e somente se ϕ é satisfatível. Chame essa subcoleção T.

Para obter essa redução encontramos estruturas do problema $SOMA\text{-}SUBCONJ$ que representam variáveis e cláusulas. A instância do problema $SOMA\text{-}SUBCONJ$

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 tem que estar em T para cada i, o que estabelece a codificação para o valor-verdade de x_i na atribuição que satisfaz a fórmula.

Cada posição na cláusula contém um certo valor no alvo t, que impõe um requisito sobre o subconjunto T. Provamos que esse requisito é o mesmo que aquele sobre a cláusula correspondente, a saber, que um dos literais naquela cláusula seja atribuído o valor VERDADEIRO.

Prova. Já sabemos que $SOMA\text{-}SUBCONJ \in$ NP, portanto agora mostramos que $3SAT \leq_{\mathrm{P}} SOMA\text{-}SUBCONJ$.

Seja ϕ uma fórmula booleana com variáveis $x_1, \ldots, x_l$ e cláusulas $c_1, \ldots, c_k$. A redução converte ϕ para uma instância do problema $SOMA\text{-}SUBCONJ$ $\langle S, t\rangle$, na qual os elementos de S e o número t são as linhas na seguinte tabela expressa em notação decimal comum. 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 compreendem os elementos de S. A linha abaixo da linha dupla é a linha 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 indicado na tabela. A parte da esquerda compreende um 1 seguido de $l - i$ 0's. A parte da 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 $\overline{x_i}$. Dígitos não especificados como sendo 1 são 0.

A tabela é parcialmente preenchida para ilustrar cláusulas-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 cáusula c_j. Esses dois números são iguais e consistem de um 1 seguido de $k - j$ 0's.

Finalmente, o número alvo t, a última linha da tabela, consiste de l 1's seguidos de k 3's.

Agora mostramos por que essa construção funciona. Demonstramos que ϕ é satisfatível se e somente se algum subconjunto S tem soma igual a t.

Suponha que ϕ seja satisfatível. Construimos um subconjunto de S da seguinte maneira. Selecionamos y_i se x_i recebe o valor VERDADEIRO na atribuição que satisfaz ϕ e z_i se x_i recebe o valor FALSO. Se somarmos o que selecionamos até então, obtemos um 1 em cada um dos l primeiros 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 mais números g e h em quantidade suficiente para trazer cada um dos últimos k dígitos para 3, e portanto atingindo o alvo.

Suponha que um subconjunto de S tem soma igual a t. Construimos uma atribuição que satisfaz ϕ após fazer várias observações. Primeiro, todos os dígitos nos membros de S são 0 ou 1. Além do mais, cada coluna na tabela que descreve S contém no

máximo cinco 1's. Logo um "transporte" para a próxima coluna nunca ocorre quando um subconjunto de S é somado. Para obter um 1 em cada uma das l primeiras colunas o subconjunto tem que ter ou y_i ou z_i para cada i, mas não ambos.

Agora montamos uma atribuição que satisfaz ϕ. Se o subconjunto contém y_i, atribuimos VERDADEIRO a x_i, caso contrário, atribuimos FALSO. Essa atribuição tem que satisfazer ϕ porque em cada uma das k colunas finais a soma é sempre 3. Na coluna c_j, no máximo 2 podem vir de g_j e h_j, portanto pelo menos 1 nessa coluna tem que vir de algum y_i ou z_i no subconjunto. Se for y_i, então x_i aparece em c_j e recebe VERDADEIRO, portanto c_j é satisfeita. Se for z_i, então $\overline{x_i}$ aparece em c_j e x_i recebe FALSO, portanto c_j é satisfeita. Por conseguinte ϕ é satisfeita.

Finalmente, temos que ter certeza de que a redução pode ser realizada em tempo polinomial. A tabela tem um tamanho de aproximadamente $(k + l)^2$, e cada entrada é facilmente calculada para qualquer ϕ. Portanto o tempo total é $O(n^2)$ estágios fáceis.

. . . . . .

. . . . . .

Exercícios

7.1 Para cada item abaixo diga se é VERDADEIRO ou FALSO.

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

7.2 Para cada item abaixo diga se é VERDADEIRO ou FALSO.

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

7.3 Quais dos seguintes pares de números são primos entre si? Mostre os cálculos que levaram a suas conclusões.

a. 1274 e 10505
b. 7289 e 8029

7.4 Preencha a tabela descrita no algoritmo de tempo polinomial para reconhecimento de linguagem livre-do-contexto do Teorema 7.14 para a cadeia $w =$ `baba` e a GLC G:

$$\begin{array}{lcl} S & \to & RT \\ R & \to & TR \mid \mathsf{a} \\ T & \to & TR \mid \mathsf{b} \end{array}$$