

Apostila Introdutória de Algoritmos

Guilherme D. da Fonseca
Celina M. H. de Figueiredo

Versão rascunho para o curso de Análise de Algoritmos da Unirio
2009.

2 de Outubro de 2009

Problemas NP-Completo

Ao longo deste livro, estudamos técnicas para desenvolver algoritmos eficientes para diversos problemas. Porém, existem vários problemas para os quais não é conhecido nenhum algoritmo eficiente. Pergunta-se: até que ponto vale a pena tentar encontrar um algoritmo eficiente para um problema? Afinal, pode ser que tal algoritmo sequer exista. Por isso, é importante conhecer problemas para os quais não existe algoritmo eficiente, de modo a evitar esforços em vão.

Neste capítulo, estudamos uma classe de problemas para os quais acredita-se que não é possível obter algoritmos eficientes. Embora ninguém tenha conseguido provar este fato, apresentamos evidências que mostram que é pouco provável que exista algoritmo eficiente para resolver qualquer um desses problemas, chamados de problemas NP-Difíceis (um subconjunto dos problemas NP-Difíceis são os problemas NP-Completo).

10.1. Tempo Polinomial no Tamanho da Entrada

Ao longo do livro, usamos vários parâmetros da entrada, e até mesmo da saída, para expressar a complexidade de tempo dos algoritmos. Quando a entrada é um grafo, usualmente expressamos a complexidade de tempo em função de n e m , os números de vértices e de arestas do grafo. Ao analisarmos o problema de determinar se um número p é primo, seria natural expressar a complexidade em função do valor p . Assim, o algoritmo que testa dividir p por todos os números naturais de 2 até $\lfloor \sqrt{p} \rfloor$, tem complexidade de tempo $O(\sqrt{p})$. Porém, ao compararmos a complexidade de tempo de algoritmos para problemas diferentes, não podemos dizer que um algoritmo $O(\sqrt{p})$ para testar primalidade é mais eficiente ou menos eficiente que um algoritmo $O(n + m)$ para um problema em grafos. Felizmente, existe uma propriedade natural da entrada de todos os problemas que permite comparar complexidades de tempo de algoritmos para problemas diferentes.

O tamanho da entrada de um problema é o número de bits gastos para descrever esta entrada. Para representarmos um número p em uma máquina binária, precisamos de $n = O(\lg p)$ bits. A complexidade de tempo do algoritmo que testa primalidade, se descrita em função do tamanho da entrada n , é $O(2^n)$.

Um algoritmo é dito polinomial se sua complexidade de tempo é limitada por um polinômio no tamanho da entrada. Por exemplo, um algoritmo $O(n^2)$, onde n é o tamanho da entrada, é claramente polinomial. Um algoritmo $O(n^2 \lg n)$ também é polinomial, pois $O(n^2 \lg n) = O(n^3)$. O algoritmo que testa primalidade em tempo $\Theta(2^n)$ não é polinomial. Denotamos por $poli(n)$ um polinômio qualquer em n .

Ao invés de representar os números em notação binária, podemos representá-los em notação unária. Deste modo, um número p gasta $O(p)$ bits para ser representado, e não $O(\lg p)$. Um algoritmo é dito pseudo-polinomial, se a sua complexidade de tempo for $O(poli(n))$, onde n é o tamanho da entrada com todos os números escritos em notação unária. Deste modo, o algoritmo que apresentamos para testar primalidade é pseudo-polinomial, embora não seja polinomial. Neste texto, consideramos que todos os números são escritos em notação binária, a não ser quando dizemos o contrário.

Porque é útil separar os algoritmos em polinomiais e não polinomiais? Consideramos que os algoritmos polinomiais são eficientes, tendo complexidade de tempo aceitável para a maioria das aplicações práticas e consideramos que algoritmos não polinomiais não são eficientes, tendo pouca utilidade prática. A realidade é um pouco diferente. De fato um algoritmo $O(n^8)$ não é muito interessante na prática. Além disso, existem diversos algoritmos com complexidade

de tempo até mesmo linear no tamanho da entrada que, devido às grandes constantes ocultas pela notação O , não tem qualquer utilidade prática. Por outro lado, existem algoritmos não polinomiais com excelente desempenho prático, dos quais o mais famoso é o método simplex usado em programação linear. Embora o método simplex tenha complexidade exponencial no pior caso, na maioria dos casos encontrados na prática este método é bastante rápido.

Ainda assim, a grande maioria dos algoritmos polinomiais tem boa performance prática e a grande maioria dos algoritmos não polinomiais tem péssima performance prática. É raro encontrar um algoritmo com complexidade de tempo $O(n^8)$. Poucos são os algoritmos polinomiais que tem complexidade de tempo $\Omega(n^4)$.

Até aqui, a separação dos algoritmos em polinomiais e não polinomiais ainda pode parecer arbitrária. Poderíamos dividir os algoritmos em algoritmos com complexidade até $O(n^4)$ e algoritmos que não tem complexidade $O(n^4)$, por exemplo, e dizer que os primeiros são eficientes enquanto os últimos não são. Mesmo que esta divisão fosse razoável, não conseguiríamos desenvolver a teoria com base nela. A facilidade matemática de separar os algoritmos em polinomiais e não polinomiais ficará clara na próxima sessão.

10.2. Problemas de Decisão e Reduções

Um problema de decisão é um problema que possui apenas duas respostas: *sim* e *não*. Neste capítulo, nos restringimos a problemas de decisão. Indiretamente, porém, tratamos de outros tipos de problemas. Por exemplo, se não existir algoritmo polinomial que diz se um grafo possui conjunto independente com pelo menos k vértices, então certamente não existe algoritmo polinomial que encontra o maior conjunto independente em um grafo. Afinal, a existência de um algoritmo polinomial para o problema de otimização implicaria em um algoritmo polinomial para o problema de decisão.

Outra maneira de entender problemas de decisão é como reconhecimento de linguagens. Todo problema de decisão pode ser visto como, dada uma entrada x , decidir se $x \in L$ para uma linguagem específica L . Por exemplo, se L é o conjunto dos números primos, decidir se x é primo é equivalente a decidir se $x \in L$. Por causa dessa correspondência entre problemas de decisão e linguagens, alternamos livremente entre um e outro. Denotamos por L_π a linguagem correspondente ao problema π , isto é, a linguagem que contém todas as entradas para as quais a resposta do problema π é *sim*. Denotamos por $\pi(x)$ a resposta do problema π para a entrada x . Denotamos por $A(x)$ a saída do algoritmo A para a entrada x .

Dados dois problemas π e π' , dizemos que π reduz polinomialmente a π' se existe algoritmo polinomial que transforma uma entrada x de π em uma entrada x' de π' tal que $x \in L_\pi \leftrightarrow x' \in L_{\pi'}$. Em outras palavras, π reduz polinomialmente a π' se existe algoritmo polinomial T tal que $\pi(x) = \pi'(T(x))$. O tamanho da saída $T(x)$ deve ser limitado por um polinômio no tamanho de x . Chamamos o algoritmo T de transformação. Usamos a notação $\pi' \leq_P \pi$ para dizer que π' se reduz polinomialmente a π .

O seguinte teorema mostra a utilidade das reduções polinomiais.

TEOREMA 10.1. *Dados dois problemas π e π' onde $\pi \leq_P \pi'$, se existe algoritmo polinomial para resolver π' , então existe algoritmo polinomial para resolver π . Analogamente, se não existe algoritmo polinomial para resolver π então não existe algoritmo polinomial para resolver π' .*

DEMONSTRAÇÃO. Provaremos que, se existir algoritmo polinomial para resolver π' , então existe algoritmo polinomial para resolver π . Como $\pi \leq_P \pi'$, podemos resolver π fazendo uma redução polinomial da entrada de π para a entrada de π' e, em seguida, rodando o algoritmo polinomial que resolve π' . A primeira etapa, que consiste em executar o algoritmo de transformação, leva tempo polinomial. A segunda etapa leva tempo polinomial no tamanho da entrada de π' , que, por sua vez, é um polinômio no tamanho da entrada de π (já que o algoritmo de transformação é polinomial). Como $O(\text{poli}(\text{poli}(n))) = O(\text{poli}(n))$, o teorema segue. \square

Além disso, a relação de redutibilidade polinomial é transitiva:

TEOREMA 10.2. *Se $\pi \leq_P \pi'$ e $\pi' \leq_P \pi''$, então $\pi \leq_P \pi''$.*

DEMONSTRAÇÃO. Seja T o algoritmo polinomial que transforma a entrada de π na entrada de π' e T' o algoritmo polinomial que transforma a entrada de π' na entrada de π'' . O algoritmo $T'(T(x))$ é polinomial e reduz π a π'' . \square

10.3. Certificados Polinomiais e a Classe NP

Um ciclo Hamiltoniano em um grafo é um ciclo que contém todos os vértices do grafo. Considere o problema de decisão a seguir, para o qual não é conhecido nenhum algoritmo polinomial:

PROBLEMA 23. *Dado um grafo G , dizer se G possui ciclo Hamiltoniano.*

Digamos que uma raça alienígena possua poder de computação ilimitado, podendo executar qualquer algoritmo, polinomial ou não, instantaneamente. Nós terráqueos, entretanto, estamos limitados a executar algoritmos polinomiais e possuímos dois grafos G_1 e G_2 , com milhares de vértices cada um, que desejamos saber se possuem ciclo Hamiltoniano. Então, perguntamos aos alienígenas se o grafo G_1 possui ciclo Hamiltoniano. Recebemos como resposta um sonoro *sim*.

Neste momento, surge uma dúvida: “Será que os alienígenas falam a verdade?” Para esclarecer esta dúvida, um terráqueo tem a seguinte idéia: “Peça para eles nos mostrarem o ciclo.” Então, os alienígenas fornecem uma seqüência de milhares de vértices que corresponde ao ciclo Hamiltoniano. Com algum trabalho, verificamos que esta seqüência tem todos os vértices do grafo exatamente uma vez e todas as arestas do ciclo de fato existem. Afinal, esta verificação pode ser feita em tempo polinomial. Assim, temos certeza que os alienígenas forneceram a resposta certa com relação ao grafo G_1 .

Apresentamos, então, o grafo G_2 aos alienígenas, que desta vez respondem *não*. Neste caso, não podemos pedir para os alienígenas exibirem o ciclo Hamiltoniano que não existe. Ficamos para sempre na dúvida se eles disseram ou não a verdade sobre o grafo G_2 .

Esta história ilustra a classe de problemas chamada de NP, à qual o problema ciclo Hamiltoniano pertence. Para os problemas na classe NP, existe um certificado polinomial para a resposta *sim*. Mais formalmente, se $\pi \in NP$ então, para toda entrada $x \in L_\pi$ existe uma seqüência de bits c com $|c| = O(\text{poli}(|x|))$, chamada de certificado polinomial, tal que existe algoritmo polinomial que, recebendo como entrada x e c , verifica que $x \in L_\pi$. No caso do ciclo Hamiltoniano o certificado polinomial é o próprio ciclo (figura 10.1). No problema de dizer se um número é composto, o certificado polinomial pode ser um fator do número.

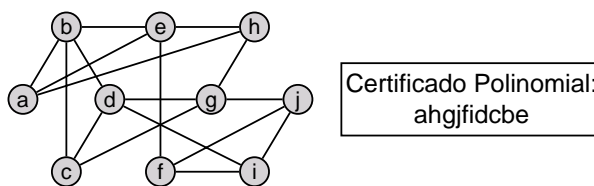


FIGURA 10.1. Grafo que possui ciclo hamiltoniano com o certificado polinomial.

Entretanto, para a resposta *não*, isto é, quando $x \notin L_\pi$, não é necessário que exista este certificado. No caso, não temos necessariamente como provar que um grafo não possui ciclo Hamiltoniano. Quando um número é primo, também não parece óbvio que exista certificado polinomial para dizer que o número é primo (embora exista quando o número é composto). De fato, existe um certificado polinomial que diz que um número é primo, mas este certificado não é simples e não entraremos em detalhes aqui.

Outra classe de problemas é chamada de CO-NP. Um problema pertence a CO-NP quando existe certificado para a resposta *não*. Todo problema pertencente a NP possui um problema simétrico em CO-NP. Dizer se um grafo *não* possui ciclo Hamiltoniano é um problema em CO-NP. Como mencionamos, o problema de dizer se um número é primo, ou, simetricamente, dizer se um número é composto, pertence simultaneamente a NP e a CO-NP, pois possui certificado polinomial tanto para o *sim* quanto para o *não*.

Todos os problemas de decisão que examinamos nos capítulos anteriores deste livro estão tanto em NP quanto em CO-NP. Isto acontece porque estes problemas estão em P, que é a classe dos problemas de decisão que podem ser resolvidos por um algoritmo polinomial no tamanho da entrada. Quando um problema pertence a P, o certificado vazio é um certificado polinomial válido tanto para o *sim* quanto para o *não*. Afinal, fornecendo apenas a entrada do problema, é possível decidir em tempo polinomial se a resposta é *sim* ou *não*. O diagrama com as classes P, NP e CO-NP está representado na figura 10.2. Claramente P está na interseção de NP e CO-NP, porém, não se sabe se P é a interseção de NP e CO-NP, ou seja, se existe algum problema π tal que $\pi \in NP \cap CO-NP$ e $\pi \notin P$. Além disso, não se sabe se de fato $P = NP = CO-NP$, ou seja, todos os problemas em NP e CO-NP de fato podem ser resolvidos em tempo polinomial. Embora a grande maioria dos estudiosos do assunto acredite que esta igualdade não é verdade, até hoje ninguém conseguiu provar este fato.

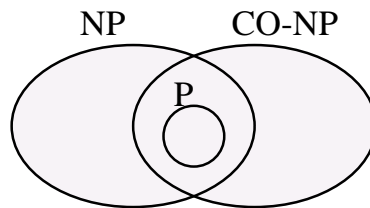


FIGURA 10.2. Diagrama das classes P, NP e CO-NP.

Muitos leigos acreditam que, assim como um problema em P é um problema polinomial, um problema em NP é um problema não polinomial. Isto está completamente errado, afinal os problemas polinomiais estão certamente na classe NP. A sigla NP surgiu de *non-deterministic polynomial time* (tempo polinomial não determinístico), pois uma outra definição para a classe NP é a classe dos problemas que podem ser resolvidos em tempo polinomial por uma máquina de Turing não determinística. Sem entrarmos em muitos detalhes, uma máquina de Turing não determinística é um modelo para um computador que pode, a cada instrução executada, se bifurcar em dois caminhos de processamento distintos, respondendo *sim* caso algum caminho responda *sim* e *não*, caso todos os caminhos respondam *não*. A máquina de Turing não determinística, diferente da máquina de Turing determinística, modela um computador que, pelo menos por enquanto, não sabemos como construir fisicamente. Um exemplo de algoritmo não determinístico que resolve ciclo Hamiltoniano em tempo polinomial está na figura 10.3.

10.4. Os Problemas NP-Completo

Cada classe de problemas contém infinitos problemas bastante diferentes entre si. É natural que alguns problemas sejam mais difíceis de resolver que outros da mesma classe, segundo algum critério. Na classe NP, existe um conjunto de problemas, chamados de NP-Completo (ou apenas NPC) que são os problemas mais difíceis de resolver da classe NP. Um problema $\Pi \in NP$ é NP-Completo se, para todo o problema $\pi \in NP$, $\pi \leq_P \Pi$. Deste modo, se for descoberto um algoritmo polinomial para algum problema NP-Completo, então $P = NP = CO-NP$.

Não parece simples provar que um problema é NP-Completo. De fato, é relativamente complicado provar que um *primeiro* problema é NP-Completo. Porém, a partir do momento que foi provado que um problema Π é NP-completo, é bem simples provar que Π' também é NP-Completo: basta provar que $\Pi \leq_P \Pi'$.

TEOREMA 10.3. *Se $\Pi \in NPC$, $\Pi' \in NP$ e $\Pi \leq_P \Pi'$, então $\Pi' \in NPC$.*

DEMONSTRAÇÃO. Se $\Pi \in NPC$, então para todo $\pi \in NP$, $\pi \leq_P \Pi$. Como $\Pi \leq \Pi'$, pelo teorema 10.2, para todo $\pi \in NP$, $\pi \leq_P \Pi'$. Como $\Pi' \in NP$, o teorema segue. \square

Uma maneira mais sucinta de definir a classe NPC é definir primeiro a classe dos problemas NP-Difíceis. Um problema Π é NP-Difícil se, para todo o problema $\pi \in NP$, $\pi \leq_P \Pi$.

Entrada: G : Grafo conexo.Saída:Resposta *sim* ou *não* para a questão se G possui ciclo Hamiltoniano.Observações:Este pseudo-código é para um modelo não determinístico de computação, respondendo *sim* caso alguma ramificação responda *sim* e *não* caso todas as ramificações respondam *não*.CicloHamiltoniano(G) $v_0 \leftarrow v \leftarrow$ vértice qualquer de $V(G)$ $c \leftarrow 1$ Enquanto $c \leq |V(G)|$ Marcar v $c \leftarrow c + 1$ Se existir vértice v' não marcado tal que $(v, v') \in E(G)$ Para todo vértice v' não marcado tal que $(v, v') \in E(G)$ Crie uma ramificação do processamento onde $v \leftarrow v'$

Senão

 Retorne *não*Se $(v, v_0) \in E(G)$ Retorne *sim*

Senão

 Retorne *não*

FIGURA 10.3. Algoritmo que resolve ciclo Hamiltoniano em tempo polinomial em uma máquina não determinística.

Um problema NP-Difícil não precisa pertencer a classe NP. De fato, um problema NP-Difícil não precisa nem mesmo ser um problema de decisão, mas não entraremos nesse assunto aqui. Com esta definição, pode-se definir a classe NPC como $\text{NPC} = \text{NP-Difícil} \cap \text{NP}$. Analogamente, pode-se definir a classe de problemas CO-NP-Completo (ou CO-NPC) como $\text{CO-NPC} = \text{NP-Difícil} \cap \text{CO-NP}$. Um diagrama dessas classes, como a maioria dos estudiosos do assunto acredita que se relacionem, está na figura 10.4. Lembramos que, até hoje, ninguém conseguiu provar que $P \neq \text{NP}$.

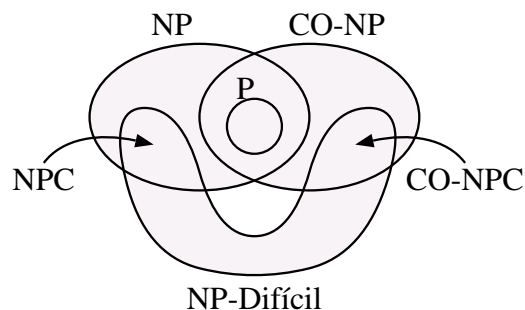


FIGURA 10.4. Diagrama das classes P, NP e CO-NP, NPC, CO-NPC e NP-Difícil.

Nas próximas sessões, provamos que alguns problemas são NP-Completo. Estas demonstrações são baseadas no teorema 10.3, fazendo redução de um problema para outro. A figura 10.5 mostra, em ordem, as reduções que são apresentadas.

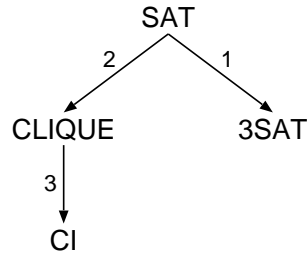


FIGURA 10.5. Reduções entre problemas NP-Completo, numeradas segundo a ordem com que são apresentadas neste capítulo.

10.5. Satisfabilidade

O primeiro problema que foi provado NP-Completo é chamado de satisfabilidade, ou simplesmente **SAT**. Neste problema, é fornecida uma expressão lógica na forma normal conjuntiva e deseja-se saber se a expressão é satisfatível. A forma normal conjuntiva é formada por um conjunto de cláusulas *ou* (representado pelo operador \vee) unidas pelo operador *e* (representado por \wedge). Um exemplo de expressão na forma normal conjuntiva é:

$$(a \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{c} \vee d) \wedge (b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee \bar{d}).$$

Nestas expressões, o literal \bar{a} representa a negação do literal a , ou seja, a é verdadeiro se e só se \bar{a} é falso. A expressão é satisfatível se existir atribuição de valores verdadeiro e falso aos literais de modo que a expressão seja verdadeira. A expressão acima é satisfatível, podendo ser satisfeita pela atribuição:

$$a = \text{verdadeiro}, b = \text{verdadeiro}, c = \text{verdadeiro}, d = \text{falso}.$$

Um exemplo mínimo de uma expressão na forma normal conjuntiva não satisfatível é $(a) \wedge (\bar{a})$. Eis o problema **SAT**:

PROBLEMA 24. (SAT) *Dada uma expressão lógica na forma normal conjuntiva, dizer se a expressão é satisfatível.*

O teorema a seguir foi provado por Cook, mas prová-lo foge do escopo deste livro. Nos contentamos em justificar que $\text{SAT} \in NP$, pois a atribuição de variáveis é um certificado polinomial para a resposta *sim*.

TEOREMA 10.4. $\text{SAT} \in NPC$

Uma variação do problema **SAT** é chamada de **3SAT**.

PROBLEMA 25. (3SAT) *Dada uma expressão lógica na forma normal conjuntiva, com no máximo 3 literais por cláusula, dizer se a expressão é satisfatível.*

Um exemplo de expressão de **3SAT** é:

$$(a \vee b \vee \bar{d}) \wedge (\bar{a} \vee c \vee \bar{d}) \wedge (b \vee d) \wedge (\bar{b} \vee \bar{c} \vee \bar{d}).$$

Certamente o problema **3SAT** não é mais difícil de resolver que o problema **SAT**. Afinal, o problema **3SAT** é um caso específico do problema **SAT**. Seria extremamente simples provar que $\text{3SAT} \leq_P \text{SAT}$, porém queremos provar a direção contrária.

TEOREMA 10.5. $\text{3SAT} \in NPC$

DEMONSTRAÇÃO. Claramente, $\text{3SAT} \in NP$, pois uma atribuição de valores aos literais é um certificado polinomial para o *sim*. Pelo teorema 10.3, basta provarmos que $\text{SAT} \leq_P \text{3SAT}$.

Podemos transformar uma cláusula C com $n > 3$ literais em duas cláusulas C_1 e C_2 com $n - 1$ e 3 literais, respectivamente, pelo processo que definimos a seguir. A aplicação sucessiva

deste método permite que uma cláusula com um número arbitrariamente grande de literais seja reduzida a várias cláusulas com 3 literais por cláusula.

Sejam x_1, \dots, x_n os literais de uma cláusula $C = (x_1 \vee \dots \vee x_n)$ com $n > 3$ literais. Criamos uma variável adicional y e definimos as duas cláusulas como:

$$C_1 = (x_1 \vee \dots \vee x_{n-2} \vee y) \text{ e } C_2 = (x_{n-1}, x_n, \bar{y}).$$

Precisamos provar que a aplicação dessa transformação não altera a satisfabilidade da expressão.

Dada uma atribuição de valores às variáveis, caso a cláusula C seja verdadeira, algum literal x_i é verdadeiro. Então, ou x_i está em C_1 ou x_i está em C_2 . Caso x_i esteja em C_1 , podemos satisfazer as duas cláusulas criadas fazendo $y = \text{falso}$. Caso x_i esteja em C_2 , podemos satisfazer as duas cláusulas criadas fazendo $y = \text{verdadeiro}$.

Caso a cláusula C seja falsa, não existe literal x_i verdadeiro. Neste caso, não importa se $y = \text{verdadeiro}$ ou $y = \text{falso}$, uma das duas cláusulas C_1 ou C_2 não será satisfeita. Deste modo, a expressão inteira não será satisfeita.

Claramente esta transformação leva tempo polinomial no tamanho da entrada. \square

Deste modo, podemos transformar a expressão de SAT:

$$(a \vee \bar{b} \vee \bar{c} \vee d \vee \bar{e}) \wedge (b \vee \bar{c} \vee d \vee e) \wedge (a \vee c) \wedge (\bar{a} \vee \bar{d} \vee e)$$

na expressão de 3SAT:

$$(a \vee \bar{b} \vee y_3) \wedge (\bar{c} \vee y_1 \vee \bar{y}_3) \wedge (d \vee \bar{e} \vee \bar{y}_1) \wedge (b \vee \bar{c} \vee y_2) \wedge (d \vee e \vee \bar{y}_2) \wedge (a \vee c) \wedge (\bar{a} \vee \bar{d} \vee e).$$

10.6. Clique e Conjunto Independente

Uma clique em um grafo é um subconjunto de seus vértices cujo subgrafo induzido é completo. Em outras palavras, uma clique em um grafo G é um subconjunto $Q \subseteq V(G)$ tal que, para todo par de vértices distintos $v_1, v_2 \in Q$, a aresta $(v_1, v_2) \in E(G)$. Um exemplo de clique está na figura 10.6.

PROBLEMA 26. (CLIQUE) Dados um grafo G e um inteiro k , dizer se G possui clique com pelo menos k vértices.

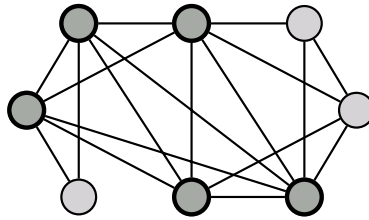


FIGURA 10.6. Grafo com uma clique de 5 vértices em destaque.

Provaremos que CLIQUE é NP-Completo fazendo uma redução polinomial de SAT a CLIQUE. Note que estamos reduzindo problemas que não parecem ter qualquer relação. O problema CLIQUE é um problema de grafos, enquanto o problema SAT é um problema de lógica.

TEOREMA 10.6. $CLIQUE \in NPC$

DEMONSTRAÇÃO. Claramente, $CLIQUE \in NP$, pois a própria clique é um certificado polinomial para o *sim*. Pelo teorema 10.3, basta provarmos que $SAT \leq_P CLIQUE$.

A nossa transformação é definida da seguinte maneira. Para cada literal x_i em cada cláusula c criamos um vértice correspondente x_i^c no grafo. As arestas são colocadas sempre entre vértices de cláusulas distintas, desde que estes vértices não correspondam a um literal e sua negação. Um exemplo desta transformação está na figura 10.7. O valor de k é definido como o número de cláusulas.

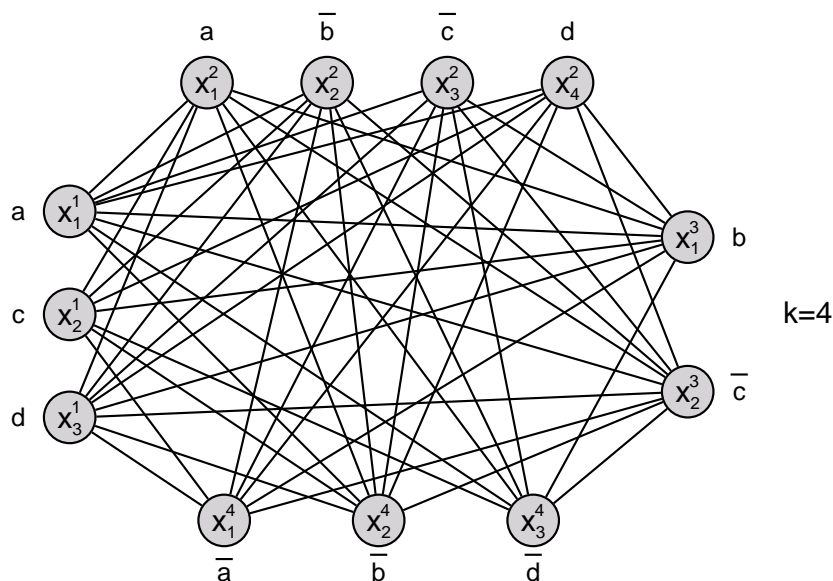


FIGURA 10.7. Grafo obtido pela redução da expressão $(a \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{c} \vee d) \wedge (b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee \bar{d})$.

Claramente esta transformação pode ser feita em tempo polinomial no tamanho da entrada, embora este tempo não seja linear, mas sim quadrático. Precisamos provar que o grafo obtido pela transformação possui clique de tamanho pelo menos k se e só se a expressão lógica é satisfável.

Suponha que o grafo possui uma clique com pelo menos k vértices. Como os vértices provenientes da mesma cláusula não possuem arestas entre si, certamente a clique possui um vértice vindo de cada cláusula. Certamente, não há na clique vértices correspondentes a um literal e sua negação, pois estes vértices não possuiriam aresta entre eles. Então, podemos atribuir valor verdadeiro a todos os literais correspondentes aos vértices da clique. Esta atribuição satisfaz a todas as cláusulas, pois tem pelo menos um literal verdadeiro em cada cláusula.

Para provar a outra direção, suponha que a expressão lógica é satisfável e fixe uma atribuição de valores que a satisfaça. Então, cada cláusula possui pelo menos um literal verdadeiro. Defina Q como um conjunto de vértices correspondente a um literal verdadeiro de cada cláusula. Por definição, Q possui k vértices, um de cada cláusula. Além disso, como não há em Q vértices correspondentes a um literal e sua negação, então Q é uma clique. \square

Um conjunto independente em um grafo é um subconjunto de seus vértices tal que não exista aresta entre qualquer par de vértices do subconjunto. O problema abaixo é extremamente semelhante ao problema clique.

PROBLEMA 27. (CI) Dados um grafo G e um inteiro k , dizer se G possui conjunto independente com pelo menos k vértices.

Podemos provar que CI é NP-Completo fazendo uma redução simples de CLIQUE para CI.

TEOREMA 10.7. $CI \in NPC$

DEMONSTRAÇÃO. Claramente, $CI \in NP$, pois o próprio conjunto independente é um certificado polinomial para o *sim*. Pelo teorema 10.3, basta provarmos que $CLIQUE \leq_P CI$.

A transformação polinomial de CLIQUE para CI é bastante simples. Basta mantermos o valor de k inalterado e gerarmos o grafo \bar{G} como o complemento do grafo G . O conjunto Q é uma clique em G se e só se o conjunto Q é um conjunto independente em \bar{G} (figura 10.8). Esta redução é claramente polinomial. \square

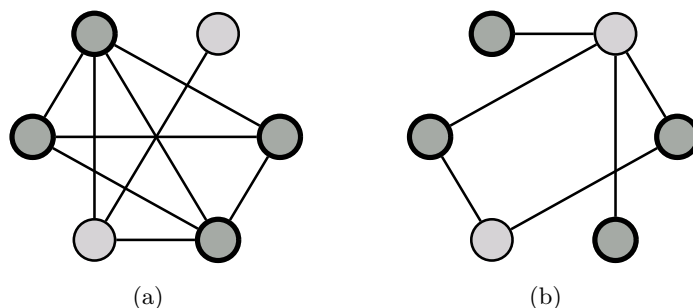


FIGURA 10.8. (a) Grafo G com uma clique de 4 vértices destacada. (b) Grafo \overline{G} com o conjunto independente correspondente a clique da figura (a) destacado.

10.7. Resumo e Observações Finais

Neste capítulo estudamos uma classe de problemas de decisão que não parecem poder ser resolvidos por nenhum algoritmo polinomial, embora ninguém tenha conseguido provar esta afirmação.

Uma redução polinomial de um problema π a um problema π' consiste de um algoritmo polinomial que transforma a entrada de π em uma entrada de π' tal que a resposta dos respectivos problemas para estas entradas seja a mesma. Se existe redução polinomial de π para π' , dizemos que π reduz polinomialmente a π' e denotamos por $\pi \leq_P \pi'$.

Os problemas de decisão da classe NP são aqueles cuja resposta *sim* pode ser verificada em tempo polinomial. Analogamente, os problemas de decisão da classe CO-NP são aqueles cuja resposta *não* pode ser verificada em tempo polinomial. Um problema Π é NP-Difícil se, para todo problema $\pi \in \text{NP}$, $\pi \leq_P \Pi$. A classe NP-Completo, ou NPC, é definida como $\text{NPC} = \text{NP} \cap \text{NP-Difícil}$. A classe CO-NP-Completo, ou CO-NPC, é definida como $\text{CO-NPC} = \text{CO-NP} \cap \text{NP-Difícil}$.

O primeiro problema provado NP-Completo foi o problema SAT. Provamos que outros problemas são NP-Completos reduzindo-os polinomialmente a SAT. Provamos que 3SAT, CLIQUE e CI são NP-Completos.

Exercícios

- 10.1) Prove que todo problema na classe NP pode ser resolvido por um algoritmo com complexidade de tempo exponencial no tamanho da entrada.
- 10.2) Prove que todo problema de decisão que pode ser resolvido em tempo polinomial por uma máquina não determinística com as características descritas a seguir pertence a classe NP. A máquina não determinística retorna *sim* caso alguma ramificação do processamento retorne *sim* e retorna *não* caso todas as ramificações do processamento retornem *não*. Como deveria ser uma máquina não determinística de modo a provar que todo problema de decisão resolvido por ela em tempo polinomial pertença a classe CO-NP?
- 10.3) O problema EXATAMENTE3SAT consiste em, dada um expressão lógica na forma normal conjuntiva com *exatamente* 3 literais por cláusula, decidir se a expressão é satisfatível. Prove que $\text{EXATAMENTE3SAT} \in \text{NPC}$. Sugestão: prove que $3\text{SAT} \leq_P \text{EXATAMENTE3SAT}$.
- 10.4) O problema 2SAT consiste em, dada um expressão lógica na forma normal conjuntiva com até 2 literais por cláusula, decidir se a expressão é satisfatível. Este problema é polinomial. Tente provar que 2SAT é NP-Completo. Explique porque não é possível fazer uma pequena adaptação na prova que 3SAT é NP-Completo de modo a provar que 2SAT é NP-Completo.

- 10.5) O problema MAXV2SAT consiste em, dados uma expressão lógica na forma normal conjuntiva com até 2 literais por cláusula e um inteiro k , decidir se a expressão pode ser satisfeita por uma atribuição de valores lógicos as variáveis onde pelo menos k variáveis possuem valor verdadeiro. O problema MAX2SAT consiste em, dados uma expressão lógica na forma normal conjuntiva com até 2 literais por cláusula e um inteiro k , decidir se é possível satisfazer pelo menos k cláusulas da expressão. Prove que MAXV2SAT e MAX2SAT são NP-Completos. Sugestão: prove que $CI \leq_P \text{MAXV2SAT} \leq_P \text{MAX2SAT}$.
- 10.6) Uma cobertura de vértice em um grafo G é um subconjunto C de $V(G)$ tal que, para toda aresta $(v, v') \in E(G)$, $v \in C$ ou $v' \in C$ (possivelmente v e v' pertencem a C). Prove que decidir se um grafo possui cobertura de vértice com no máximo k vértices é NP-Completo.
- 10.7) Um caminho Hamiltoniano em um grafo G é um caminho que contém todos os vértices de G . Prove que decidir se um grafo G possui caminho Hamiltoniano é NP-Completo. Considere provado que *ciclo* Hamiltoniano é NP-Completo.
- 10.8) Na sessão 8.3, definimos o problema de programação linear. Prove que a versão de decisão do problema de programação linear, com um número livre de variáveis, com a restrição adicional de que as variáveis só podem possuir valor 0 ou 1, é NP-Completa. Na versão de decisão, o problema de programação linear consiste em obter uma atribuição de variáveis tal que a função objetivo tenha valor pelo menos k , onde k é parte da entrada do problema.
- *10.9) Prove que decidir se um grafo possui Ciclo Hamiltoniano é NP-Completo.
- *10.10) Um grafo G é 3-colorível se existe uma atribuição de cores aos seus vértices, dentre um conjunto de 3 cores, tal que quaisquer dois vértices adjacentes possuam cores distintas. Prove que decidir se um grafo é 3-colorível é um problema NP-Completo.