Projeto e Análise de Algoritmos

Flávio L. C. de Moura*

19 de agosto de 2024

1 NP-completude

Praticamente todos os algoritmos estudados até aqui são polinomiais, i.e. o tempo de execução destes algoritmos no pior caso está em $O(p(n))$, onde $p(n)$ é um polinômio no tamanho da entrada n [1, 4, 2, 3]. Estes algoritmos formam a classe dos algoritmos que são considerados "eficientes". Os problemas que podem ser resolvidos por algoritmos polinomiais são chamados de "tratáveis". No entanto, vimos algoritmos cujos tempos de execução são exponenciais no tamanho da entrada, como por exemplo, busca em grafos utilizando força bruta. De fato, suponha que, dados um digrafo G e vértices u e v de G, queremos determinar quando existe um caminho de u para v em G. Utilizando um algoritmo força bruta, precisamos considerar todos os potenciais caminhos em G, e determinar quando algum deles é um caminho de u para v. Um potencial caminho é uma sequência de vértices de G de tamanho menor ou igual a $|V|$ (o número de vértices de G). O número de potenciais caminhos é $|V|^{|V|}$, e portanto exponencial.

Neste capítulo estudaremos uma classe de problemas para os quais as técnicas estudadas até aqui não se aplicam. Estes problemas não têm soluções polinomiais conhecidas, mas também não existe uma prova de que tais soluções não existam: esta é a famosa questão "P versus NP" que constitui um dos mais interessantes problemas em aberto na Computação. Formalmente, definimos um problema como a seguir:

Definição 1.1. *Um problema (abstrato) Q é uma relação binária que associa um conjunto I de instâncias a um conjunto S de soluções.*

Por exemplo, o problema PATH é uma relação que associa cada instância de um digrafo e dois vértices com um caminho que contém os dois vértices. O problema SHORTEST-PATH é uma relação que associa cada instância de um digrafo e dois vértices com um caminho mínimo que contém os dois vértices. Como caminhos mínimos não são necessariamente únicos, uma instância de um problema pode ter mais de uma solução. Neste capítulo trabalharemos essencialmente com *problemas de decisão*, que são problemas cujas respostas são sempre *sim* ou *não*:

Definição 1.2. *Um problema de decisão é uma relação binária sobre um conjunto I de instâncias e um conjunto binário (sim ou não) de soluções.*

Assim, podemos ver um problema de decisão como sendo uma função que associa instâncias em I ao conjunto de soluções $\{0, 1\}$. Por exemplo, o problema PATH é uma relação que associa, cada instância de um digrafo e dois vértices, com um caminho que contém os dois vértices. Este problema pode ser visto como um problema de decisão: Dados um digrafo G, e vértices u e v de G, determinar se existe um caminho de u para v em G.

Seja $\mathcal{C}$ uma classe de problemas caracterizados por uma propriedade, como por exemplo, possuir solução em tempo polinomial. Estamos interessados em identificar os problemas mais difíceis em $\mathcal{C}$, de tal forma que uma solução eficiente para algum destes problemas difíceis resulte em soluções eficientes para todos os outros problemas de $\mathcal{C}$. Na próxima seção estudaremos a classe P dos problemas que podem ser resolvidos deterministicamente em tempo polinomial.

*flaviomoura@unb.br

A classe P

A classe P consiste dos problemas que podem ser resolvidos em tempo polinomial por um algoritmo determinístico.

Teorema 1.3. *PATH está em P*

Prova. Considere o seguinte algoritmo que recebe como entrada um digrafo G e vértices u e v.

1. Marque o vértice u
2. Enquanto existir aresta $(a, b) \in G$ com a marcado, e b não-marcado, marque b.
3. Se v está marcado retorne 1, caso contrário retorne 0.

   Análise do algoritmo: O *laço* (linha 2) pode ser executado segundo o algoritmo de busca em largura, por exemplo, que é linear no tamanho da representação do grafo G. As outras linhas do algoritmo são executadas apenas uma vez, portanto o algoritmo é polinomial.

□

Teorema 1.4. *2-SAT* $\in$ *P.*

- Redução polinomial

  A *redutibilidade* entre dois problemas é uma técnica de projeto de algoritmos muito utilizada para dar informações sobre a dificuldade de problemas. Por exemplo, considere um problema A que queremos resolver, e suponha que sabemos como resolver um outro problema B. Se for possível transformar instâncias do problema A em instâncias do problema B com as seguintes características:

  – A transformação é feita em tempo polinomial;
  – As respostas são as mesmas para ambas as instâncias.

  Então chamamos este procedimento de *algoritmo de redução*, e o mesmo nos fornece uma forma de resolver o problema A a partir do problema B:

  – Dada uma instância α do problema A, usamos o algoritmo de redução para transformá-la em uma instância β do problema B;
  – Executamos o algoritmo polinomial para a instância β de B;
  – Usamos a resposta de β como resposta de α.

  Ou seja, podemos assim construir um algoritmo para o problema A. O que podemos concluir a partir da existência de um tal algoritmo de redução? A existência de um algoritmo eficiente para B nos permite construir um algoritmo eficiente para A, mas também que a não existência de um algoritmo eficiente para A implica na não existência de um algoritmo eficiente para B. Em outras palavras, o problema B é tão difícil quanto o problema A, ou ainda, A não é mais difícil do que B.

- Linguagem formal

  Um problema de decisão pode ser visto como um problema de reconhecimento de linguagem: seja U o conjunto de todas as entradas possíveis para o problema de decisão. Seja $L \subseteq U$, o conjunto de todas as entradas para as quais a resposta do problema é *sim*. Chamamos L a *linguagem* correspondente ao problema, e utilizamos os termos *problema* e *linguagem* de forma intercambiável. O problema de decisão deve então reconhecer se uma dada entrada pertence ou não à linguagem L.

  Definição 1.5. *Seja* Σ *um conjunto finito (de símbolos) que chamaremos de alfabeto. O conjunto de todas as palavras (*strings*) que podem ser formadas com os símbolos de* Σ *é denotado por* Σ^**. Uma linguagem* L *sobre* Σ *é qualquer subconjunto de* Σ^**. O alfabeto vazio é denotado por* ϵ*, enquanto que a linguagem vazia é denotada por* $\emptyset$*.*

Diversas operações podem ser realizadas sobre linguagens: união, interseção, complemento, concatenação e fechamento. Do ponto de vista da teoria de linguagens formais, o conjunto das instâncias de qualquer problema de decisão Q é simplesmente o conjunto Σ^*, onde $\Sigma = \{0,1\}$. Como o problema Q é caracterizado pelas instâncias que produzem resposta 1 (*sim*), podemos ver a linguagem L sobre $\Sigma = \{0,1\}$ como sendo $L = \{x \in \Sigma^* \mid Q(x) = 1\}$.

Definição 1.6. *Dizemos que um algoritmo A aceita a palavra $x \in \{0,1\}^*$ se $A(x) = 1$, i.e. a resposta do algoritmo A ao receber a entrada x é igual a 1. A linguagem aceita pelo algoritmo A é o conjunto das palavras aceitas pelo algoritmo: $L = \{x \in \{0,1\}^* \mid A(x) = 1\}$. Dizemos que o algoritmo A rejeita a palavra x, se $A(x) = 0$.*

Note que, mesmo que a linguagem L seja aceita pelo algoritmo A, isto não significa que A rejeita uma palavra $x \notin L$ porque, por exemplo, A pode entrar em *loop*.

Definição 1.7. *Uma linguagem L é decidida pelo algoritmo A, se A aceita toda palavra em L, e rejeita toda palavra que não está em L. Uma linguagem L é aceita em tempo polinomial pelo algoritmo A, se A aceita L, e se existe uma constante k tal que para qualquer palavra $x \in L$ de comprimento n, o algoritmo A aceita x em tempo $O(n^k)$, para algum $k \geq 0$. Uma linguagem L é decidida em tempo polinomial pelo algoritmo A se existir uma constante não-negativa k tal que para qualquer palavra $x \in \{0,1\}^*$, o algoritmo decide em tempo $O(n^k)$ se $x \in L$.*

Assim, para aceitar uma linguagem, o algoritmo precisa apenas produzir uma resposta para toda palavra de L, mas para decidir uma linguagem, o algoritmo precisa aceitar ou rejeitar toda palavra em $\{0,1\}^*$. Como exemplo, o problema de decisão PATH corresponde a seguinte linguagem:

PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ é um grafo (não dirigido), $u, v \in V, k \geq 0$ é um inteiro, e existe um caminho de u para v em G contendo, no máximo, k arestas$\}$. Assim, a linguagem PATH pode ser aceita em tempo polinomial pelo seguinte algoritmo: Dada uma instância (G, u, v), o algoritmo executa BFS para computar o menor caminho de u para v, e então compara o número de arestas deste menor caminho com k. Se o menor caminho possui até k arestas então o algoritmo retorna 1, e para. Caso contrário, o algoritmo entra em *loop*. Note que este algoritmo não decide PATH. Neste caso, para decidir PATH basta que o algoritmo retorne 0 e pare, ao invés de entrar em *loop*. Para outros problemas, como o *problema da parada para máquinas de Turing*, existem algoritmos de aceitação, mas não de decisão.

Por fim, definimos a classe P em termos de linguagens:

$P = \{L \subseteq \{0,1\}^* :$ existe um algoritmo A que decide L em tempo polinomial$\}$

Definição 1.8. *Dizemos que uma linguagem L_1 é redutível polinomialmente à linguagem L_2, notação $L_1 \leq_p L_2$, se existe uma função computável em tempo polinomial $f : \{0,1\}^* \to \{0,1\}^*$ tal que para todo $x \in \{0,1\}^*$, $x \in L_1$ se, e somente se, $f(x) \in L_2$. A função f é chamada de* função de redução*, e o algoritmo polinomial F que computa a função f é chamado de* algoritmo de redução*.*

Observe que ao reduzirmos uma linguagem (ou problema) L_1 para outra linguagem (ou problema) L_2, queremos que cada instância de L_1 seja transformada em uma instância de L_2, isto é, se $x \in L_1$ então $f(x) \in L_2$. Adicionalmente, precisamos que elementos que não sejam instância de L_1 não sejam levados em L_2: $x \notin L_1$ então $f(x) \notin L_2$, o que é equivalente por contraposição a se $f(x) \in L_2$ então $x \in L_1$. Juntando estas duas informações temos a equivalência "$x \in L_1$ se, e somente se, $f(x) \in L_2$" da definição acima.

Reduções polinomiais são uma ferramenta poderosa que nos permitem provar que outras linguagens estão em P:

Lema 1.9. *Sejam $L_1, L_2 \in \{0,1\}^*$ linguagens tais que $L_1 \leq_p L_2$, então $L_2 \in P$ implica que $L_1 \in P$.*

Prova. Seja A_2 um algoritmo polinomial que decide a linguagem L_2, e F um algoritmo de redução polinomial que computa a função de redução f. Construiremos um algoritmo A_1 que decide L_1 da seguinte forma: dado $x \in \{0,1\}^*$, o algoritmo A_1 inicialmente usa F para transformar x em $f(x)$, e então usa o algoritmo A_2 para responder. Note que A_1 é polinomial já que tanto A_2 quanto F são polinomiais. □

Considere o problema 2-SAT, cujas instâncias são expressões lógicas formadas por conjunções de disjunções de dois literais, onde um literal é uma variável booleana ou a negação de uma variável booleana. Por exemplo, a expressão a seguir é uma instância de 2-SAT:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

Uma solução da instância acima é uma designação de valores booleanos (0 ou 1) para as variáveis que satisfaçam a expressão, ou seja, que retornem 1. Por exemplo, a designação $x_1 = 1$, $x_2 = 1$ e $x_3 = 0$ satisfaz a expressão acima.

A classe NP

A classe NP consiste dos problemas que podem ser resolvidos em tempo polinomial por um algoritmo não-determinístico. A ideia é que inicialmente, o algoritmo advinhe uma solução (fase não-determinística), e em seguida esta solução deve ser verificada em tempo polinomial deterministicamente. Assim, a forma mais usual de apresentar a classe NP, consiste em considerar os problemas que podem ser verificados em tempo polinomial por um algoritmo determinístico [1, 4].

Como vimos, em alguns casos é possível evitar uma abordagem força bruta e encontrar soluções polinomiais para o problema em questão. Mas é fácil imaginar que isto nem sempre será possível. De fato, veremos que existem diversos problemas interessantes/importantes para os quais soluções polinomiais não foram encontradas até hoje, mas que ainda é possível verificar em tempo polinomial, dado um certificado.

Exemplo 1.10. *O problema de encontrar ciclos Hamiltonianos em (di)grafos tem sido estudado por muito tempo (mais de 100 anos!). Formalmente, um ciclo Hamiltoniano de um grafo $G = (V, E)$ é um ciclo simples que contém cada vértice de V, i.e. cada vértice de G é visitado uma única vez. Um (di)grafo que contém um ciclo Hamiltoniano é dito Hamiltoniano.*

Denotaremos por HAM-CYCLE o problema de encontrar ciclos Hamiltonianos em (di)grafos.

HAM-CYCLE = $\{\langle G\rangle : G$ é um (di)grafo Hamiltoniano $\}$

Podemos verificar uma possível solução em tempo polinomial: Suponha que um colega te diz que um (di)grafo G é Hamiltoniano, e como justificativa, fornece uma sequência de vértices na ordem que ele diz formar um caminho Hamiltoniano.

1. *Verifique que os vértices dados constituem o conjunto V dos vértices de G;*
2. *Verifique que cada par de vértices consecutivos da sequência dada corresponde a uma aresta de G.*

Como a verificação acima pode ser feita em tempo polinomial, temos que HAM-CYCLE $\in$ NP.

- Algoritmos de Verificação

Definição 1.11. *Um algoritmo de verificação é um algoritmo A que recebe dois argumentos x e y, e verifica se $A(x, y) = 1$. Uma linguagem verificada por um algoritmo de verificação A é*

$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^*$ *tal que* $A(x, y) = 1\}$

Intuitivamente, o algoritmo A verifica a linguagem L se, para cada $x \in L$, existe um certificado y tal que A é usado para provar que $x \in L$. Formalmente, a classe NP é a classe das linguagens que podem ser verificadas em tempo polinomial.

$$NP = \{L \subseteq \{0,1\}^* : \text{ existe um algoritmo determinístico } A \text{ que verifica } L \text{ em tempo polinomial}\}$$

Mais precisamente, $L \in$ NP se, e somente se, existe um algoritmo polinomial A e uma constante c tais que $L = \{x \in \{0,1\}^* : \exists y, |y| = O(|x|^c)$ tal que $A(x, y) = 1\}$.

Agora é fácil ver que HAM-CYCLE $\in$ NP. Basta checar que o caminho dado (sequência de vértices) é uma permutação de V, isto é, cada vértice ocorre apenas uma vez, e que existe uma aresta em G para cada par de vértices consecutivos do caminho dado, e entre o primeiro e último vértices.

Estamos interessados em algoritmos que verificam se uma instância está ou não em uma linguagem. Por exemplo, dada uma instância (G, u, v) do problema de decisão PATH, e um caminho p de u para v, podemos facilmente verificar se p é um caminho em G.

A classe NP consiste dos problemas que podem ser verificados em tempo polinomial, i.e. dado um certificado, podemos verificar em tempo polinomial que este certificado é correto. Por exemplo, considerando o problema dos ciclos hamiltonianos em um (di)grafo $G = (V, E)$ com $n = |V|$, um certificado pode ser uma sequência $\langle v_1, v_2, \ldots, v_n \rangle$ de vértices que pode ser verificada em tempo polinomial. Para o problema 3-SAT, um certificado pode ser uma designação de valores para as variáveis da fórmula que pode ser verificado (se satisfaz ou não a fórmula dada) em tempo polinomial.

Exemplo 1.12. *Um clique em um grafo (não dirigido) é um subgrafo onde dois vértices quaisquer estão ligados por uma aresta. Um k-clique é um clique que contém k vértices. O problema CLIQUE consiste em determinar se um grafo contém um clique de um tamanho especificado:*

$CLIQUE = \{(G, k) : G$ *é um grafo com um k-clique*$\}$

***Afirmação**: $CLIQUE \in NP$*

O clique é o certificado. Para a entrada $((G, k), c)$

1. *Verifique se c é um subconjunto de G.V de tamanho k;*
2. *Verifique se G contém todas as arestas que conectam vértices em c;*
3. *Se ambas as verificações podem ser feitas então retorne 1, caso contrário, retorne 0.*

Teorema 1.13. *3-SAT* $\leq_P$ *CLIQUE*

Prova. Nos grafos a serem construídos, cliques de um tamanho específico correspondem a designações satisfatíveis da fórmula. Seja φ uma fórmula com k cláusulas

$$\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots (a_k \vee b_k \vee c_k)$$

A redução f constrói a codificação $\langle G, k \rangle$ onde G é um grafo não dirigido dado por:

- Os vértices de G são organizados em k grupos de 3 vértices cada $t_1, t_2, \ldots, t_k$. Cada tripla t_i corresponde a uma das cláusulas de φ, e cada vértice na tripla corresponde a um literal da cláusula associada. Marque cada vértice de G com o literal correspondente em φ. As arestas de G conectam todos os vértices exceto:
  - vértices contraditórios, como x e $\neg x$;
  - vértices da mesma tripla.

Afirmação: φ é satisfatível se, e somente se, G possui um k-clique.

Suponha que φ é satisfatível, e portanto cada cláusula possui pelo menos um literal verdadeiro. Em cada tripla em G, selecionamos um vértice correspondente ao literal verdadeiro. Se mais de um literal for verdadeiro na mesma cláusula, escolhemos um deles aleatoriamente. Os vértices selecionados formam um k-clique: o número de vértices selecionados é k, cada par de vértices selecionado está ligado por uma aresta.

Suponha que G possui um k-clique. Nenhum par de vértices do clique ocorre na mesma tripla porque vértices da mesma tripla não são ligados por arestas. Portanto, cada tripla contém exatamente um dos vértices do k-clique. Designamos valores para as variáveis de φ de forma que cada literal que marca um vértice assume valor 1 (verdadeiro). Isto é possível porque vértices contraditórios não são ligados. Esta designação de variáveis satisfaz a fórmula φ porque cada tripla corresponde a um vértice do clique, e portanto cada cláusula de φ tem valor 1. □

A classe NPC

Dizemos que um problema é *NP-completo*, i.e. que está na classe NPC, se está na classe NP e é tão difícil quanto qualquer problema em NP. Ao mostrarmos que um problema é NP-completo, não estamos tentando provar a existência de um algoritmo eficiente, mas concluir que a existência de um tal algoritmo é improvável. De fato, estamos concluindo sobre o quão difícil ele é, e não sobre quão fácil como fizemos até então. Portanto, um algoritmo eficiente provavelmente não existe para este problema.

Uma importante questão em aberto é quando P é ou não um subconjunto próprio de NP, o que corresponde ao problema "P vs NP" citado anteriormente. Problemas NP-completos normalmente são considerados intratáveis dada a grande quantidade de problemas NP-completos já estudados, e sem solução polinomial encontrada.

Propriedade interessante: Se algum problema NP-completo puder ser resolvido em tempo polinomial então qualquer problema em NP terá solução polinomial.

Seria uma surpresa encontrar uma solução polinomial para algum (e portanto para todos) destes problemas. Neste sentido, se um problema é NP-completo então isto pode ser visto como uma evidência da sua intratabilidade. Neste caso, algoritmos aproximados devem ser considerados, ao invés de soluções rápidas e exatas.

Por que até hoje ninguém conseguiu encontrar soluções polinomiais para estes problemas? Não sabemos, talvez porque elas simplesmente não existam, ou porque elas estejam baseadas em princípios ainda desconhecidos. Qualquer problema em P está também em NP porque pode ser verificada em tempo polinomial pelo mesmo algoritmo que a decide em P sem a necessidade de utilizar certificados.

Definição 1.14. *Uma linguagem $L \subseteq \{0,1\}^*$ é dita* ***NP-completa*** *se:*

1. $L \in NP$;
2. $L' \leq_P L, \forall L' \in NP$.

Denotamos por NPC a classe das linguagens NP-completas.

Se uma linguagem L satisfaz a propriedade 2 acima, mas não necessariamente a propriedade 1, dizemos que L é **NP-difícil**.

Como NP-completude consiste em mostrar quão difícil é um problema, utilizamos a redução polinomial na outra direção para mostrar que um problema é NP-completo:

Para mostrarmos que um problema B não possui solução polinomial, consideremos um problema A, para o qual sabemos não existir solução polinomial. Suponha também que temos um algoritmo de redução que transforma instâncias de A em instâncias de B em tempo polinomial. Agora, se existisse uma solução polinomial para B então poderíamos construir uma solução polinomial para A como acima, contradizendo a suposição de que A não possui solução polinomial.

Lema 1.15. *Se L é uma linguagem tal que $L' \leq_P L$ para algum $L' \in NPC$, então L é NP-difícil. Se adicionalmente, $L \in NP$ então $L \in NPC$.*

Prova. Como L' é NP-completo, então $\forall L'' \in$ NP, temos que $L'' \leq_P L'$, e por hipótese temos que $L' \leq_P L$. Logo. $L'' \leq_P L$, o que mostra que L é uma linguagem NP-difícil. Agora, se $L \in$ NP então, por definição temos que $L \in$ NPC. □

O Lema 1.15 nos dá um método para mostrar que uma linguagem L é NP-completa:

1. Prove que $L \in$ NP;
2. Escolha uma linguagem L' que seja NP-completa;
3. Descreva um algoritmo que computa uma função f que mapeia cada instância x de L' em uma instância $f(x) \in L$;
4. Prove que $x \in L'$, se e somente se, $f(x) \in L$;
5. Prove que o algoritmo que computa f é polinomial

Os passos de 2-5 mostram que L é NP-difícil.

Exemplo 1.16. *SAT* $= \{\langle\varphi\rangle : \varphi$ *é uma fórmula booleana satisfatível*$\}$.

Podemos determinar em tempo exponencial se uma dada fórmula booleana φ *contendo* n *variáveis é satisfatível: basta checar cada uma das* 2^n *possíveis valorações para as variáveis de* φ*. Nenhum algoritmo polinomial é conhecido para SAT. O teorema a seguir, mostra que é muito improvável que tal algoritmo exista.*

O teorema a seguir é conhecido como o *teorema de Cook-Levin*:

Teorema 1.17. *SAT* $\in$ *NPC.*

Teorema 1.18. *3-SAT* $\in$ *NPC.*

Prova.
1. 3-SAT $\in$ NP: Dada uma designação de valores para as variáveis de uma 3-FNC fórmula (certificado), o algoritmo de verificação substitui cada variável pelo valor dado e avalia a expressão. Se a expressão resulta em 1 então o certificado é válido e a fórmula é satisfatível.
2. SAT $\leq_P$ 3-SAT. Feito em aula.

□

Exercício 1.19. *Mostre que CLIQUE* $\in$ *NPC.*

Exercício 1.20. *Uma cobertura de vértices de um grafo* $G = (V, E)$ *(não-dirigido) é um subconjunto* $V' \subseteq V$ *tal que* $(u, v) \in E$ *então* $u \in V'$ *ou* $v \in V'$ *(ou ambos!). O tamanho de uma cobertura de vértices é o seu número de vértices, ou seja,* $|V'|$*. O problema da cobertura de vértices consiste em encontrar uma cobertura de vértices de tamanho mínimo em um grafo. Reescrevendo este problema como um problema de decisão, queremos determinar se um grafo possui uma cobertura de vértices de tamanho dado* k*:*

VERTEX-COVER $= \{\langle G, k\rangle : G$ *possui uma cobertura de vértices de tamanho* $k\}$

Mostre que VERTEX-COVER $\in$ *NPC.*

Exercício 1.21. *Um* caminho Hamiltoniano *em um digrafo* G *é um caminho de um vértice* u *para um vértice* v *que visita todos os vértices de* G *exatamente uma vez. O problema de decisão HAMPATH consiste em, dado um digrafo* G*, responder se* G *possui um caminho Hamiltoniano ou não.*

HAMPATH $= \{\langle G, u, v\rangle : G$ *é um digrafo com um caminho Hamiltoniano de* u *para* $v\}$

Mostre que HAMPATH $\in$ *NPC.*

Exercício 1.22. *Considere o seguinte jogo em um grafo (não-dirigido)* $G = (V, E)$*, que inicialmente contém 0 ou mais bolas de gude em seus vértices: um movimento deste jogo consiste em remover duas bolas de gude de um vértice* $v \in V$*, e adicionar uma bola a algum vértice adjacente de* v*. Agora, considere o seguinte problema (de decisão): Dado um grafo* G*, e uma função* $p(v)$ *que retorna o número de bolas de gude no vértice* v*, existe uma sequência de movimentos que remove todas as bolas de* G*, exceto uma? Mostre que este problema está em NPC.*

Exercício 1.23. *Um* ciclo Hamiltoniano *em um digrafo* G *é um ciclo simples contendo cada vértice de* G*. O problema de decisão HAM-CYCLE consiste em, dado um digrafo* G*, responder se* G *possui um ciclo Hamiltoniano ou não.*

HAM-CYCLE $= \{\langle G\rangle : G$ *é um digrafo com um ciclo Hamiltoniano.*$\}$

Mostre que HAM-CYCLE $\in$ *NPC.*

Exercício 1.24. *Considere um jogo de tabuleiro que contém* n *linhas e* n *colunas. Cada uma das* n^2 *posições, pode ter uma bola azul, uma bola vermelha ou pode estar vazia (configuração inicial). O jogo consiste em remover as bolas do tabuleiro de forma que cada coluna tenha bolas de uma única cor, e cada linha contenha pelo menos uma bola (configuração vencedora). Mostre que o problema de determinar se uma configuração inicial resulta em uma configuração vencedora está em NPC.*

Referências

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

[3] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.

[4] Michael Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1st edition, 1996.