Representação 2

2.1 Isomorfismo

Dois grafos são isomorfos quando têm exatamente a mesma estrutura de conexões, embora seus vértices possam estar nomeados de maneiras diferentes. Ou seja, quando há uma relação 1-1 entre seus vértices que preserva a relação de adjacência. A definição forma é a seguinte.

Definição 2.1: grafos isomorfos

Dizemos que os grafos G e H são ***isomorfos*** se existe uma função bijetora

$$f : V(G) \to V(H) \tag{2.1}$$

tal que

$$\{u, v\} \in E(G) \text{ se, e somente se, } \{f(u), f(v)\} \in E(H)$$

para todos $u, v \in V(G)$. Nesse caso, escrevemos $G \simeq H$.

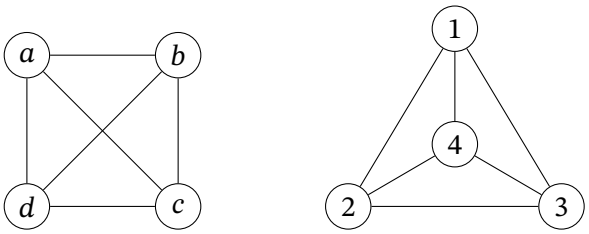
Grafos isomorfos diferem apenas na maneira como sua estrutura é representada, não na estrutura propriamente dita.

Uma função f como (2.1) acima é chamada de ***isomorfismo*** e, por abuso de notação, as vezes escrevemos

$$f : G \to H$$

para um isomorfismo como definido acima.

Por exemplo, são (trivialmente) isomorfos os grafos definidos pelas representações:



e um isomorfismo é a função f dada por

$$f(a) = 1,\ f(b) = 2,\ f(c) = 3 \text{ e } f(d) = 4.$$

Observe que há outros isomorfismos possíveis.

Notemos que quaisquer dois grafos completos G e H com n vértices são isomorfos. Mais que isso, entre grafos completos, qualquer uma das $n!$ bijeções entre $V(G)$ e $V(H)$ define um isomorfismo entre eles.

Nesse caso, dizemos que o grafo completo é ***único a menos de isomorfismos***. Ingenuamente, isso quer dizer que o grafo é único a

menos dos nomes dados aos vértices. Esse fato justifica usarmos a mesma notação para todos eles: K^n.

Exemplo 2.1

Há oito grafos distintos sobre o conjunto de vértices $\{1, 2, 3\}$, eles estão descritos nas representações da Figura 2.1 abaixo.

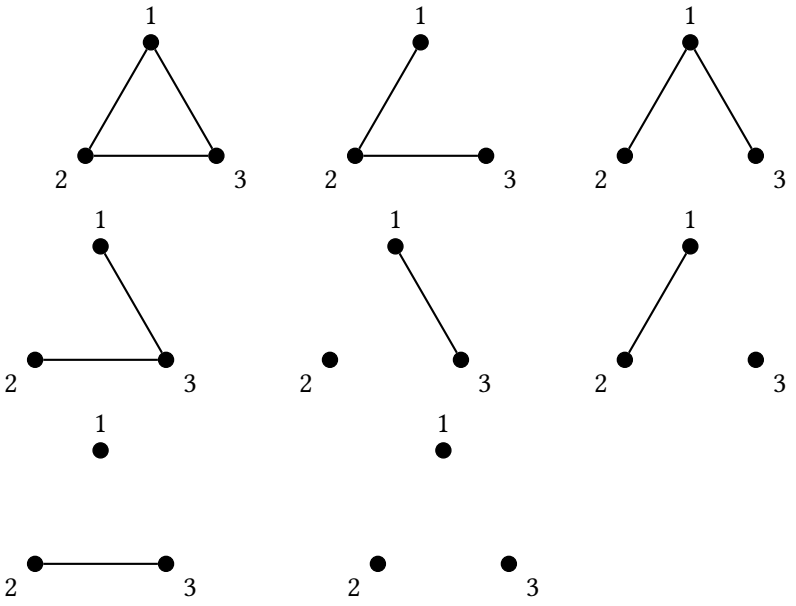


Figura 2.1: grafos distintos de ordem 3.

Exemplo 2.2

Dentre os 8 grafos distintos do exemplo anterior, há apenas 4 grafos não-isomorfos com três vértices, representados pelos diagramas da Figura 2.2. Nesse caso dizemos que ***a menos de isomorfismos*** há 4 grafos de ordem 3.

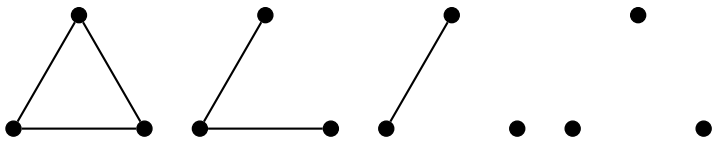


Figura 2.2: os "tipos de grafos" não-isomorfos de ordem 3.

Exemplo 2.3: Grafo de Petersen

Os grafos representados na Figura 2.3 são isomorfos pelo isomorfismo na Tabela 2.1. Qualquer grafo isomorfo a eles é chamado de ***Grafo de Petersen***, é um dos grafos mais conhecidos na teoria dos grafos. Isso se deve ao fato dele ser o menor exemplo ou o menor contraexemplo para muitos problemas em grafos. Júlio Petersen (1839–1910) foi um matemático dinamarquês que por volta de 1898 construiu o grafo que leva seu nome como o menor contraexemplo para afirmação: ***todo***

grafo conexo sem ponte admite uma 3-coloração própria das arestas. Voltaremos a essa afirmação quando tivermos conhecimento dos termos técnicos que ela contém. O grafo de Petersen é retratado nas capas de vários periódicos científicos e livros sobre teoria dos grafos e matemática discreta.

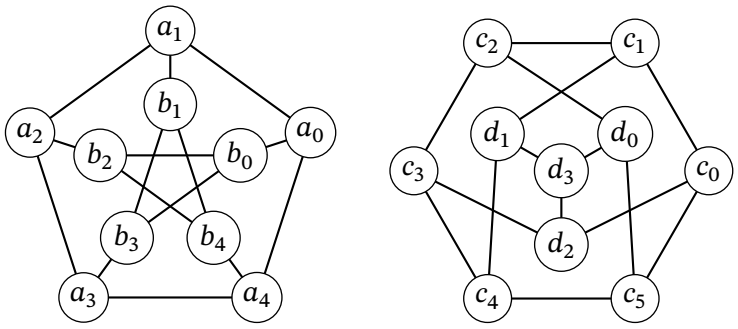


Figura 2.3: grafos isomorfos (grafo de Petersen).

v	$f(v)$	v	$f(v)$
a_0	c_3	b_0	c_4
a_1	d_2	b_1	d_3
a_2	c_0	b_2	c_5
a_3	c_1	b_3	d_1
a_4	c_2	b_4	d_0

Tabela 2.1: isomorfismo entre os grafos da Figura 2.3.

Não isomorfismo

Nenhum par dentre os grafos G, H e K representados na Figura 2.4 são isomorfos.

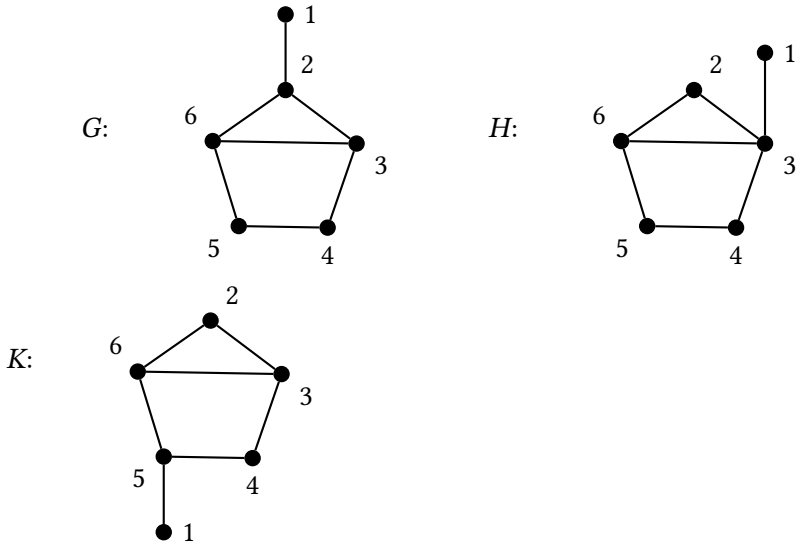


Figura 2.4: grafos não-isomorfos.

Temos que G não é isomorfo a H porque G não tem um vértice de grau quatro enquanto que o vértice 5 em H tem grau quatro, portanto não há como haver uma bijeção entre os vértices desse grafo que preserve as adjacências.

Pelo mesmo motivo H não é isomorfo a K.

Agora, G não é isomorfo a K porque caso existisse um isomorfismo $f : G \to K$ então $f(1) = 4$ e $f(2) = 5$, portanto, a imagem por f do conjunto $\{2, 3, 6\} \subset V(G)$ é, obrigatoriamente, o conjunto $\{4, 5, 6\} \subset V(K)$ o que é contradição pois $\{2, 3, 5\}$ em G induz um triângulo e $f(\{2, 3, 5\})$ em K não induz triângulo (veja o Exercício 43 abaixo).

2.2 Representações computacionais

Em problemas computacionais envolvendo grafos, é conveniente supor que o conjunto de vértices é

$$V = [n] = \{1, 2, \ldots, n\}.$$

Essa convenção não restringe a classe de grafos que podemos considerar, pois a escolha dos nomes dos vértices é irrelevante quando estamos interessados em propriedades que dependem apenas da estrutura do grafo.

Essa observação é especialmente importante do ponto de vista computacional. Uma vez que estamos interessados em propriedades invariantes por isomorfismo, podemos substituir qualquer grafo por um representante isomorfo cujos vértices sejam numerados de 1 a n. Com isso facilitamos o acesso às informações usando estruturas de dados indexadas disponíveis nas linguagens de programação estruturadas.

Dado um grafo $G = (V, E)$, podemos substituir sua representação por uma representação isomorfa na qual os vértices são inteiros. Mais precisamente, se $|V| = n$, escolhemos uma bijeção $f : V \to [n]$ e definimos o grafo $G' = ([n], E')$, onde

$$E' = \big\{\{f(u), f(v)\} : \{u, v\} \in E\big\}.$$

A função f é, por construção, um isomorfismo entre G e G'.

Para os problemas computacionais considerados nesta disciplina, essa substituição não altera o problema: as propriedades de interesse são invariantes por isomorfismo. Portanto, se G é a entrada de um problema, podemos primeiro construir sua cópia isomorfa G' e executar o algoritmo sobre G'. Uma vez obtida a solução para G', podemos transportá-la de volta para G usando o isomorfismo f^{-1}.

Por exemplo, se o algoritmo produz um caminho $(v_1, v_2, \ldots, v_k)$ em G', então $(f^{-1}(v_1), f^{-1}(v_2), \ldots, f^{-1}(v_k))$ é o caminho correspondente em G. O mesmo princípio se aplica a árvores, ciclos, conjuntos de arestas, fluxos e outras estruturas produzidas pelos algoritmos.

Na implementação, a bijeção f e sua inversa podem ser mantidas por estruturas de dados que associem os rótulos originais aos inteiros $1, \ldots, n$. Por exemplo, uma tabela de espalhamento (*hash table*) permite, em geral, realizar essas consultas em tempo esperado constante, enquanto uma árvore binária de busca permite realizá-las em tempo logarítmico quando balanceada. Não trataremos dos detalhes dessa conversão.

Essa convenção é apenas uma escolha de representação. Ela não restringe os grafos considerados, pois todo grafo com n vértices é isomorfo a um grafo com esse conjunto de vértices.

No que segue, sempre que tratamos problemas computacionais so-

bre grafos estamos assumindo os grafos são definidos sobre o conjunto de vértices $\{1, 2, \ldots, n\}$, para algum $n \in \mathbb{N}$.

2.2.1 Matriz de adjacências de G

A matriz de adjacências de $G = (V, E)$ é a matriz $|V| \times |V|$ denotada por A_G, ou A simplesmente, definida por

$$A(i, j) = \begin{cases} 1, & \text{se } \{i, j\} \in E(G) \\ 0, & \text{caso contrário.} \end{cases}$$

Para o grafo G representado na Figura 2.4, a matriz de adjacências é

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}.$$

2.2.2 Listas de adjacências de G

Outro modo menos óbvio que usa o conceito de lista ligada, ou lista encadeada. Uma lista ligada é uma coleção de elementos com uma ordem dada pelas "ligações" chamadas ponteiros. Cada elemento dessa coleção, chamado nó, tem uma informação e uma referência para o próximo elemento. Na Figura 2.6 abaixo representamos num diagrama uma lista ligada para o conjunto $\{12, 37, 99\}$. A ideia agora

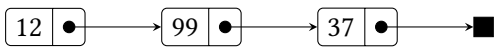


Figura 2.5: lista ligada.

é usar uma lista para a vizinhança de cada vértice de um grafo.

Lista de adjacência é um vetor $N(\,)$ de listas ligadas. A lista $N(i)$ contém os vizinhos do vértice i e cada nó dessa lista é composto por uma variável que armazena um vértice e um ponteiro que aponta para o próximo nó da lista.

A representação de uma grafo por listas de adjacências não é única, pois qualquer permutação dos nós em $N(i)$ define uma lista válida. Uma lista de adjacências para o G grafo representado na Figura 2.4 é representada na Figura 2.6.

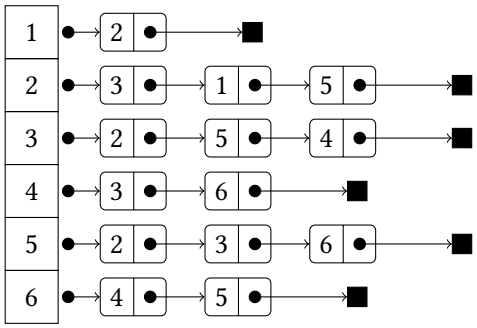


Figura 2.6: Lista de adjacências do grafo G na Figura 2.4.

2.3 Complexidade de algoritmos em grafos

Usualmente, a expressão que estabelece a complexidade de um algoritmo é escrita em função do ***tamanho da entrada***, isto é, o tamanho da representação de uma entrada para o algoritmo, em notação assintótica.

Se um algoritmo recebe um grafo (V, E) como entrada, e nada mais, então o parâmetro "tamanho da entrada" é o ***tamanho da representação*** do grafo. No caso de matriz de adjacências é razoável admitirmos que o tamanho da representação é $|V|^2$ e no caso de listas de adjacências $|V| + |E|$.

Neste texto entendemos por tempo de execução do algoritmo a ***complexidade de tempo no pior caso***, isto é, uma estimativa para o número máximo, dentre todas as entradas do mesmo tamanho, de instruções executadas e que é expressa em função do tamanho n da representação

$$T(n) = \max\{T(I) : I \text{ tem representação de tamanho } n\}$$

onde $T(I)$ denota o número de instruções executadas pelo algoritmo com entrada I.

Exemplo 2.4

Sejam A e B dois algoritmos distintos que resolvem o mesmo problema sobre um grafo $G = (V, E)$. O algoritmo A usa, no pior caso, exatos $|V|^2$ passos para executar a tarefa e o algoritmo B usa exatos $(|V| + |E|)\lfloor \log |E| \rfloor$ passos no pior caso. Para grafos com $|V|^2/4$ arestas o primeiro algoritmo é sempre melhor enquanto que para grafos com $1.000|V|$ arestas o segundo algoritmo é ***assintoticamente*** melhor (melhor para todo grafo com $|V| \geq 16.645$ vértices; o primeiro é melhor para $|V| \leq 16.644$).

Definição 2.2: notação O

Sejam f e g duas funções definidas nos naturais e que assumem valores naturais (ou, reais positivos), então

$$f(n) = O(g(n)), \quad \text{para } n \text{ suficientemente grande}$$

significa que *existem* constantes positivas c e n_0 tais que *para todo* $n \geq n_0$ vale

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

Vejamos a complexidade de alguns algoritmos que executam tarefas simples comparando cada uma das representações dadas acima para um grafo fixo $G = (V, E)$

Tarefa	Matriz de adjacências	Lista de adjacências
$\{i, j\} \in E$?	$O(1)$	$O(\|V\|)$
Computar $d(i)$	$O(\|V\|)$	$O(\|V\|)$
Escrever E	$O(\|V\|^2)$	$O(\|E\|)$

Um algoritmo em grafo é de ***complexidade de tempo polinomial*** se o tempo de execução no pior caso é expressa por uma função polinomial no tamanho da representação. É importante notar que "ter complexidade polinomial" é independente da representação do grafo ser por matriz ou lista de adjacências pois essas representações são ***polinomialmente relacionadas*** (isto é, uma pode ser obtida da outra por algoritmo de tempo polinomial). Ademais, entendemos por ***algoritmo eficiente*** um algoritmo com complexidade polinomial.

Notação assintótica

As estimativas para o custo de um algoritmo são expressas usando notação assintótica. A análise assintótica foca no comportamento do algoritmo para entradas "suficientemente grandes" e simplifica a expressão da complexidade de tempo ao ignorar constantes multiplicativas e termos aditivos de ordem inferior, uma vez que essas diferenças não alteram a ordem de grandeza do crescimento da função. A análise assintótica permite uma avaliação de desempenho representativa, independente de tecnologia ou detalhes de implementação, tornando mais simples a comparação direta entre algoritmos.

Sejam $f, g : \mathbb{N} \to \mathbb{N}$ com $g(n) > 0$ para todo n suficientemente grande. Dizemos que f é ***assintoticamente muito menor*** que g e escrevemos

$$f(n) = o(g(n)) \text{ quando } n \to \infty \tag{2.2}$$

se, e só se,

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

Por exemplo,

1. $1 = o(\log(\log(n)))$.
2. $\log(\log(n)) = o(\log(n))$.
3. $\log(n) = o(n^\varepsilon)$ para todo $\varepsilon > 0$.
4. $n^\varepsilon = o(n^c)$ para quaisquer $0 < \varepsilon < 1 \leq c$.
5. $n^c = o(n^{\log n})$ para todo $1 \leq c$.
6. $n^{\log n} = o(\exp(n))$.
7. $\exp(n) = o(n^n)$.
8. $n^n = o(\exp(\exp(n)))$.

Também usamos a notação $f \ll g$ o que nos permite escrever, a partir do exemplo acima, a sequência monótona

$$1 \ll \log(\log(n)) \ll \log(n) \ll n^\varepsilon \ll n^c \ll n^{\log n} \ll e^n \ll n^n \ll e^{e^n}.$$

Dizemos que f ***assintoticamente menor*** que g se

$$f(n) = O(g(n)). \tag{2.3}$$

Na definições dadas nas equações (2.2) e (2.3) o símbolo "=" não é a igualdade no sentido usual, é um abuso da notação que permitimos em troca de algumas conveniências. Temos que $n = O(n^2)$ e $n^2+2n+1 = O(n^2)$ mas $n \neq n^2 + 2n + 1$. Quando usamos essas definições.

Proposição 2.1

Se $f(n) = o(g(n))$ então $f(n) = O(g(n))$.

A prova é imediata da definição de limite. A recíproca da Proposição 2.1 não vale, como pode ser visto tomando-se $f(n) = g(n) = n^2$.

Proposição 2.2

Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$ então

1. $f_1(n) + f_2(n) = O(\max\{ g_1(n), g_2(n) \})$.
2. $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.
3. $a \cdot f_1(n) = O(g_1(n))$ para toda constante $a \in \mathbb{R}$.

Demonstração. Vamos provar o item 1. Digamos que $f_1(n) \leq c_1 g_1(n)$ para todo $n \geq n_1$ e que $f_2(n) \leq c_2 g_2(n)$ para todo $n \geq n_2$, onde n_1, n_2, c_1, c_2 são as constantes dadas pela definição de notação O. Então

$$\begin{aligned} f_1(n) + f_2(n) &\leq \max\{ c_1, c_2 \}(g_1(n) + g_2(n)) \text{ com } c = \max\{ c_1, c_2 \} \\ &\leq c(2 \cdot \max\{ g_1(n), g_2(n) \}) \end{aligned}$$

para todo $n \geq \max\{ n_1, n_2 \}$, o que prova a afirmação.

As outras propriedades são deduzidas de modo análogo e são deixadas como exercício. □

É preciso observamos alguns cuidados com o teorema acima. Para

inteiros positivos n e k não vale que $1^k + 2^k + \cdots + (n-1)^k + n^k = O(\max\{ 1^k, 2^k, \ldots, (n-1)^k, n^k \}) = O(n^k)$. O problema aqui é que o máximo só pode ser tomado sobre um número de termos que não dependa de n. De fato, temos que $1^k + \cdots + n^k = O(n^{k+1})$ e que $1^k + \cdots + n^k \neq O(n^k)$.

Fica como exercício a verificação do seguinte resultado.

Proposição 2.3: transitividade

Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ então $f(n) = O(h(n))$.

Mais alguns exemplos são dados a seguir.

1. $an^2 + bn + c = O(n^2)$ para toda constante $a > 0$.
   Primeiro, observamos que, para n suficientemente grande, $|an^2 + bn + c| \leq |a|n^2 + |b|n + |c|$ e agora usamos a Proposição 2.2 em cada operando das somas, de $an^2 = O(n^2)$, $|b|n = O(n)$ e $|c| = O(1)$ temos $an^2 + |b|n + |c| = O(\max\{ n^2, n, 1 \}) = O(n^2)$. Analogamente, para todo $k \in \mathbb{N}$

$$\sum_{i=0}^{k} a_i n^i = O(n^k).$$

2. $n \log(n!) = O(n^2 \log n)$.
   Primeiro, temos $n = O(n)$. Depois, $n! = \prod_{i=1}^{n} i < \prod_{i=1}^{n} n = n^n$. Como log é crescente $\log(n!) < \log(n^n) = n \log(n)$, portanto $\log(n!) = O(n \log(n))$. Pela Proposição 2.2 $n \log(n!) = O(n^2 \log n)$.
3. Para toda constante $a > 1$, $\log_a(n) = O(\log(n))$. De fato,

$$\log_a(n) = \frac{1}{\log a} \log n$$

   porém $\frac{1}{\log a} = O(1)$ e $\log n = O(\log n)$ e pela Proposição 2.2 $\log_a(n) = O(\log(n))$.

Um algoritmo eficiente com respeito ao tempo de execução é um algoritmo que nas instâncias de tamanho n tem tempo de execução $O(n^k)$ para algum inteiro positivo k fixo.

Convenções de uso da notação assintótica

Ao usar notação assintótica nós desconsideramos os coeficientes, por exemplo, usamos $O(n^2)$ ao invés de $O(3n^2)$ e $O(1)$ ao invés de $O(1024)$ ainda que, como classes de funções, $O(n^2) = O(3n^2)$ e $O(1) = O(1024)$.

Escrevemos no argumento de $O(\cdot)$ somente o termo mais significativo, por exemplo, usamos $O(n^2)$ ao invés de $O(2n^2 + 5n \log n + 4)$. Nesse caso, da Proposição 2.2 vale que $2n^2 + 5n \log n + 4 = O(\max\{ n^2, n \log n, 1 \}) = O(n^2)$.

Quando notação assintótica aparece em equações, na forma "expressão 1 = expressão 2" onde "expressão" são expressões algébricas que envolvem notação assintótica, os termos assintóticos em "expressão 1" são quantificados universalmente, enquanto que os termos assintóticos em "expressão 2" são quantificados existencialmente. Por exemplo, em $n^3 + O(n^2) = O(n^3) + n^2 + n$ entendemos como: existe um $n_0 > 0$ tal que para todo $n \geq n_0$

para todo $f(n)$ em $O(n^2)$, existe $g(n)$ em $O(n^3)$ tal que $n^3+f(n) = g(n)+n^2+n$.

Notação Ω e Θ

A notação Ω adaptada por Donald Knuth da notação introduzida por outros matemáticos é definida por

$$f(n) = \Omega(g(n)) \text{ se e somente se } g(n) = O(f(n)).$$

Escrevemos $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $g(n) = O(f(n))$.

P×NP

Atualmente não sabemos se existe algoritmo eficiente para vários problemas em grafos.

Por exemplo, não sabemos se existe algoritmo eficiente que receba dois grafos e decida se eles são isomorfos.

- ***O problema do isomorfismo de grafos***: Dados os grafos $G = (V, E)$ e $H = (V, E')$ decidir se eles são isomorfos.

Entretanto, não é difícil projetar um algoritmo de tempo polinomial que recebe a terna (G, H, f), formada por dois grafos e uma função $f : V(G) \to V(H)$ e devolve ***sim*** caso G e H são isomorfos e f é o isomorfismo, caso contrário devolve ***não***.

Em linguagem técnica, da Teoria da Computação, dizemos que ***o problema do isomorfismo de grafos está na classe* NP *de complexidade de problemas computacionais.*** Não é sabido se esse problema é NP-difícil, caso seja, um algoritmo eficiente para esse problema implica na existência de algoritmo eficiente para todo problema computacional em NP.

O problema complementar está em coNP:

- ***O problema do não-isomorfismo de grafos***: Dados os grafos $G = (V, E)$ e $H = (V, E')$ decidir se eles são não-isomorfos.

Também não se conhece algoritmo de tempo polinomial no tamanho dos grafos para decidir se dois grafos não são isomorfos. Mais do que isso, não se conhece um algoritmo de tempo polinomial no tamanho dos grafos que receba como entrada uma terna (G, H, P) onde P é uma "prov curta" (i.e., de tamanho polinomial em $|V| + |E| + |E'|$) e que decida se P é uma prova do não-isomorfismo entre G e H. Em lingua-

gem técnica dizemos que *não se sabe se o problema do não-isomorfismo de grafos está na classe* **NP** *de complexidade computacional.*

Em 2015, Babai apresentou um algoritmo para o problema de isomorfismo de grafos com tempo de execução ***quase-polinomial***. Assim, embora não se saiba se existe um algoritmo de tempo polinomial, sabemos que o problema de isomorfismo pode ser resolvido em tempo $\exp((\log n)^c)$, onde n representa o tamanho da entrada e $c > 0$ constante. O o mesmo resultado fornece um algoritmo quase-polinomial para GNI.

Assim, a dificuldade não está simplesmente em encontrar *algum* argumento para provar que dois grafos não são isomorfos. A questão mais profunda é determinar um procedimento geral e eficiente que, dados dois grafos, decida se eles pertencem à mesma classe de isomorfismo. É justamente essa questão que dá origem ao problema de isomorfismo de grafos e à sua interessante relação com a teoria da complexidade computacional.

Exercícios

Exercício 39. *Determine quais pares dentre os grafos abaixo são isomorfos.*

(i) G_1 *tal que* $V(G_1) = \{v_1, u_1, w_1, x_1, y_1, z_1\}$ *e*
$E(G_1) = \{\{u_1, v_1\}, \{u_1, w_1\}, \{v_1, w_1\}, \{v_1, x_1\}, \{w_1, y_1\}, \{x_1, y_1\}, \{x_1, z_1\}\}$;

(ii) G_2 *tal que* $V(G_2) = \{v_2, u_2, w_2, x_2, y_2, z_2\}$ *e*
$E(G_2) = \{\{u_2, v_2\}, \{u_2, w_2\}, \{v_2, w_2\}, \{v_2, x_2\}, \{w_2, y_2\}, \{x_2, y_2\}, \{y_2, z_2\}\}$;

(iii) G_3 *tal que* $V(G_3) = \{v_3, u_3, w_3, x_3, y_3, z_3\}$ *e*
$E(G_3) = \{\{u_3, v_3\}, \{u_3, w_3\}, \{v_3, w_3\}, \{v_3, x_3\}, \{w_3, y_3\}, \{x_3, y_3\}, \{u_3, z_3\}\}$.

Exercício 40. *Decida se o grafo representado pelo diagrama na Figura 2.7 abaixo é o grafo de Petersen.*

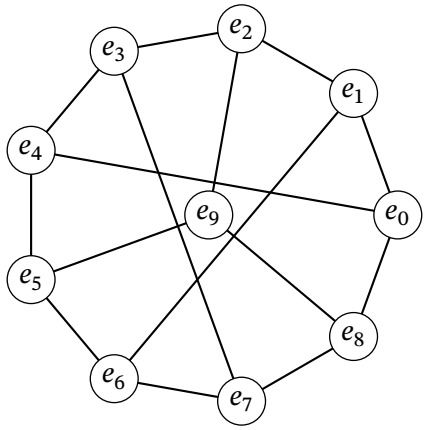


Figura 2.7: esse é o grafo de Petersen?

Exercício 41. *Decida se os dois grafos* 4*-regular representados pelos dois diagramas da Figura 2.8 abaixo são isomorfos.*

Exercício 42. *Mostre que existem* 11 *grafos não-isomorfos com* 4 *vértices.*

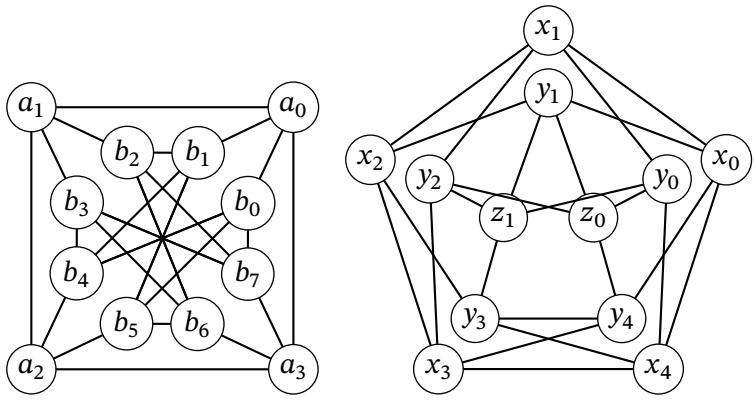


Figura 2.8: esses grafos são isomorfos?

Exercício 43. *Sejam G e H grafos isomorfos e $f : G \to H$ um isomorfismo. É verdade que $G[U]$ é isomorfo a $H[f(U)]$ para todo $U \subseteq V(G)$? Justifique.*

Exercício 44. *Mostre que o grafo de Petersen é isomorfo ao complemento do grafo linha do K^5.*

Exercício 45. *Um* **automorfismo** *de um grafo é um isomorfismo do grafo sobre ele mesmo. Quantos automorfismos tem um grafo completo?*

Exercício 46. *Mostre que o conjunto de automorfismos de um grafo com a operação de composição de funções definem um grupo.*

Exercício 47. *Qual o número de grafos distintos sobre um conjunto de vértices V de tamanho n?*

Exercício 48. *Prove que há pelo menos $2^{\binom{n}{2}}(n!)^{-1}$ grafos não isomorfos sobre um conjunto de vértices V de tamanho n.*

Exercício 49. *Um grafo G é* **vértice-transitivo** *se para quaisquer $u, v \in V(G)$ existe um automorfismo f de G com $f(v) = u$. Analogamente, G é* **aresta-transitivo** *se para quaisquer arestas $\{x, y\}, \{z, w\} \in E(G)$ existe um automorfismo f de G tal que $\{f(x), f(y)\} = \{z, w\}$.*

Dê um exemplo de grafo vértice-transitivo. Dê um exemplo de grafo aresta-transitivo. Dê um exemplo de grafo aresta-transitivo mas não vértice-transitivo.

Exercício 50. *Considere o grafo $G = (V, E)$, com $V = [n]$ e*

$$E = \{\{1,2\},\{2,3\},\{2,5\},\{3,4\},\{3,5\},\{4,6\},\{5,6\}\}.$$

(a) Escreva a matriz de adjacências A_G.
(b) Escreva uma representação de G por listas de adjacências, usando uma lista ligada para cada vértice. A ordem dos vizinhos em cada lista é livre.
(c) Quantos nós possuem, no total, as listas de adjacências? Relacione sua resposta com $|E|$.

(d) *Suponha que se queira determinar se $i, j \in E$. Explique como realizar essa consulta nas duas representações e justifique as complexidades $O(1)$ e $O(|V|)$ apresentadas no texto.*

Exercício 51. *Seja $G = (V, E)$ um grafo com $|V| = n$ e $|E| = m$.*

(a) *Mostre que uma representação de G por matriz de adjacências utiliza $\Theta(n^2)$ posições, enquanto uma representação por listas de adjacências utiliza $\Theta(n + m)$ posições.*

(b) *Para cada uma das situações abaixo, indique qual representação é mais adequada, matriz ou listas de adjacências, e justifique sua escolha comparando o custo das operações relevantes.*

- *$m = \Theta(n)$ e o algoritmo precisa percorrer todos os vizinhos de cada vértice.*
- *$m = \Theta(n^2)$ e o algoritmo realiza muitas consultas do tipo "$i, j \in E$?".*
- *$m = \Theta(n)$ e o algoritmo precisa apenas escrever todas as arestas.*
- *$m = \Theta(n^2)$ e o algoritmo precisa escrever todas as arestas.*

(c) *Explique por que, apesar das diferenças de eficiência, as duas representações não alteram a classificação de um algoritmo como sendo de tempo polinomial.*

Exercício 52. *Sejam G um grafo sobre o conjunto de vértices $[n]$, A a sua matriz de adjacências e $b(i, j)$ as entradas da matriz A^2. Que parâmetro de G está associado ao número $b(i, i)$, para cada $i \in V(G)$? Qual a relação entre $b(i, j)$ e, $N_G(i)$ e $N_G(j)$ quando $i \neq j$?*

Exercício 53. *Seja G um grafo qualquer sobre o conjunto de vértices $[n]$ e A sua matriz de adjacências. Como é possível determinar o número de triângulos de um grafo G qualquer a partir de A^3?*

Exercício 54. *O* **traço** *de uma matriz A, denotado por $\mathrm{tr}(A)$, é a soma dos elementos na diagonal da matriz. Quanto vale o traço de uma matriz de adjacências? Quanto vale $\mathrm{tr}(A^2)$ em função de $E(G)$?*

Exercício 55. *Por A ser uma matriz simétrica, sabemos da Álgebra Linear que todos os seus autovalores são números reais. Determine os autovalores da matriz de adjacências do grafo completo K^n sobre o conjunto de vértices $[n]$. Mostre que se G é um grafo r-regular sobre o conjunto de vértices $[n]$, então r é uma autovalor de A.*

Exercício 56. *Sejam G um grafo sobre o conjunto de vértices $[n]$ e A sua matriz de adjacências. Prove que se os autovalores de A são distintos então o grupo dos automorfismos (Exercício 46) é comutativo.*