

The background of the book cover features a complex network of glowing nodes and connections, resembling a graph or a neural network. The nodes are small, glowing spheres of various colors (blue, orange, yellow) scattered across a dark blue gradient background. Numerous thin, glowing lines connect these nodes, creating a web-like structure that radiates from the center towards the edges.

PAULO CÉSAR RODACKI GOMES

GRAFOS

CONCEITOS FUNDAMENTAIS, ALGORITMOS E APLICAÇÕES

Paulo César Rodacki Gomes

Grafos: conceitos fundamentais, algoritmos e aplicações

IFC

Blumenau, 2022

**INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E
TECNOLOGIA CATARINENSE**

REITORA

SÔNIA REGINA DE SOUZA FERNANDES

PRÓ-REITORA DE ENSINO

JOSEFA SUREK DE SOUZA

**PRÓ-REITORA DE PESQUISA,
PÓS-GRADUAÇÃO E INOVAÇÃO**

FÁTIMA PERES ZAGO DE OLIVEIRA

PRÓ-REITOR DE EXTENSÃO

FERNANDO JOSÉ TAQUES

**PRÓ-REITORA DE
DESENVOLVIMENTO
INSTITUCIONAL**

JAMILE DELAGNELO FAGUNDES DA SILVA

**PRÓ-REITOR DE
ADMINISTRAÇÃO**

STEFANO MORAES DEMARCO

EDITORA IFC

COORDENAÇÃO

LEILA DE SENA CAVALCANTE

CONSELHO EDITORIAL

FÁTIMA PERES ZAGO DE OLIVEIRA

LEILA DE SENA CAVALCANTE

GICELE VERGINE VIEIRA

REGINALDO LEANDRO PLÁCIDO

KÁTIA LINHAUS DE OLIVEIRA

SUELY APARECIDA DE JESUS

MONTIBELLER

HYLSON VESCOVI NETTO

HÉLIO MACIEL GOMES

SANDRO AUGUSTO RHODEN

IZACLAUDIA SANTANA DAS NEVES

MARIO WOLFART JÚNIOR

BRUNO PANSERA ESPINDOLA

JONATHAN ACHE DIAS

ELIANA TERESINHA QUARTIERO

LILIANE CERDÓTES

MARCIO PEREIRA SOARES

ILLYUSHIN ZAAK SARAIVA

ALCIONE TALASKA

DÉBORA DE LIMA VELHO JUNGES

EMANUELE CRISTINA SIEBERT

ANA NELCINDA GARCIA VIEIRA

ANDERSON SARTORI



CONTATO:

Rua das Missões, nº 100 – Ponta Aguda – Blumenau/SC – CEP: 89.051-000
Fone: (47) 3331-7850 | E-mail: editora@ifc.edu.br

PROJETO GRÁFICO
PAOLO MALORGIO STUDIO LTDA

IMAGEM DA CAPA
JOSÉ DEJANIR DE CASTRO JR.

DIAGRAMAÇÃO
PAOLO MALORGIO STUDIO LTDA

REVISÃO TEXTUAL
ALESSANDRA KLEIN

Todos os direitos de publicação reservados. Proibida a venda.
Os textos assinados, tanto no que diz respeito à linguagem como ao conteúdo, são de inteira responsabilidade dos autores e não expressam, necessariamente, a opinião do Instituto Federal Catarinense. É permitido citar parte dos textos sem autorização prévia, desde que seja identificada a fonte. A violação dos direitos do autor (Lei nº 9.610/1998) é crime estabelecido pelo artigo 184 do Código Penal.

**Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)**

Gomes, Paulo César Rodacki
Grafos [livro eletrônico] : conceitos fundamentais, algoritmos e aplicações / Paulo César Rodacki Gomes. -- Blumenau, SC : Editora do Instituto Federal Catarinense, 2022.
PDF

Bibliografia.
ISBN 978-65-88089-12-5

1. Algoritmos 2. Ciência da computação
3. Estrutura de dados (Ciência da computação)
4. Grafos - Teoria I. Título.

22-125299

CDD-511.5

Índices para catálogo sistemático:

1. Teoria dos grafos 511.5

Eliete Marques da Silva - Bibliotecária - CRB-8/9380

Sumário

Prefácio	5
1 CONCEITOS FUNDAMENTAIS	7
1.1 Introdução	7
1.2 Grafos	8
1.3 Grafos dirigidos	10
1.4 Laços e arestas paralelas	12
1.5 Parâmetros quantitativos	13
1.6 Grafos rotulados e não-rotulados	15
1.7 Isomorfismo	16
1.8 Subgrafos	17
1.9 Classes de grafos	18
1.10 Percursos, trilhas e caminhos	22
1.11 Aplicações	25
1.11.1 Labirintos	25
1.11.2 Problema dos 4 cubos	25
1.12 Exercícios	29
2 REPRESENTAÇÃO DE GRAFOS	41
2.1 Introdução	41
2.2 Operações em estruturas de dados de Grafos	42
2.3 Listas de arestas	42
2.4 Listas de adjacência	44
2.5 Mapas de adjacência	46
2.6 Matrizes de adjacência	47
2.7 Exercícios	49
3 BUSCA BÁSICA EM GRAFOS	53
3.1 Introdução	53
3.2 Busca em largura	54
3.2.1 O algoritmo de busca em largura	55
3.3 Busca em profundidade	57
3.3.1 Busca em profundidade com um vértice origem	57
3.3.2 Busca em profundidade com várias origens	58
3.4 Percorrendo a árvore de busca	59
3.5 Exemplos de aplicação de busca	60

3.5.1	Estudo de caso: busca em grafos de estados	60
3.5.2	Estudo de caso: ordenação topológica	61
3.6	Exercícios	64
4	CONEXIDADE	69
4.1	Introdução	69
4.2	Definições iniciais	69
4.3	Conexidade com algoritmo de busca	74
4.4	Conexidade por fusão de vértices	75
4.5	Conexidade por conjuntos disjuntos	77
4.6	Conexidade em grafos dirigidos	81
4.7	Exercícios	84
5	GRAFOS EULERIANOS E HAMILTONIANOS	87
5.1	Problema do Explorador e Problema do Turista	87
5.2	Grafos e Ciclos Eulerianos	88
5.2.1	Algoritmo de Fleury	91
5.2.2	Algoritmo de Hierholzer	92
5.2.3	Estudo de caso: dominó	96
5.3	Grafos e Ciclos Hamiltonianos	98
5.3.1	Estudo de caso: xadrez	100
5.3.2	Estudo de caso: <i>Gray codes</i>	102
5.4	Exercícios	104
6	ÁRVORES	111
6.1	Caracterização de árvores	111
6.2	Árvores geradoras	115
6.2.1	Estudo de caso: rigidez de treliças planas	116
6.3	Grafos valorados	119
6.4	Problema de árvore geradora de custo mínimo	120
6.4.1	Algoritmo de Prim	121
6.4.2	Algoritmo de Kruskal	123
6.5	Exercícios	128
7	CAMINHO MÍNIMO	133
7.1	Problema do caminho mínimo	133
7.1.1	Algoritmo de Dijkstra	133
7.1.2	Algoritmo de Floyd	135
7.1.3	Imprimindo caminhos na matriz de roteamento	139
7.2	Problema do Carteiro Chinês	139
7.2.1	Eulerização de grafos	140

7.2.2	Algoritmo para o problema do carteiro chinês	141
7.3	Problema do Caixeiro Viajante	144
7.3.1	Algoritmo força bruta	144
7.3.2	Heurística do vizinho mais próximo	145
7.3.3	Heurística da desigualdade do triângulo	147
7.3.4	Aplicações do problema do caixeiro viajante	148
7.4	Exercícios	150
	Lista de ilustrações	153
	Lista de quadros	157
	REFERÊNCIAS	159
	Índice	163

Prefácio

Grafos são poderosas abstrações matemáticas utilizadas para modelar e resolver uma grande variedade de problemas na Ciência da Computação, Engenharias, Ciências Naturais e Sociais. O presente trabalho traz tópicos introdutórios da Teoria dos Grafos, o qual é fruto de minhas notas de aula produzidas ao longo de cerca de 15 anos de trabalho como professor de Teoria dos Grafos em cursos de graduação em Ciência da Computação.

Ao longo desta jornada, percebeu-se que a maior parte das publicações em português davam grande ênfase a questões teóricas e ao formalismo matemático, sem, contudo, se aprofundarem em questões de implementação de algoritmos ou estruturas de dados de grafos. Por outro lado, publicações com enfoque em estruturas de dados em geral apresentam uma gama muito restrita de algoritmos, e quase sempre deixam de mostrar conceitos teóricos fundamentais de grafos.

Procurando preencher esta lacuna, decidimos produzir uma obra que trouxesse os principais algoritmos, juntamente com os conceitos fundamentais da Teoria dos Grafos necessários para uma melhor compreensão do tema como um todo. Não se trata de uma obra com enfoque teórico-matemático profundo, pois o objetivo principal é fornecer conceitos iniciais e básicos para que se comece a compreender os Grafos como ferramenta para solução de problemas práticos — com ou sem a implementação computacional de algoritmos.

Este livro é indicado como bibliografia básica inicial para o estudo de Teoria dos Grafos em cursos de graduação em Ciência da Computação e áreas afins, podendo ser útil também em cursos de pesquisa operacional, matemática discreta e estruturas de dados. Para cursos técnicos integrados ao ensino médio, concomitantes ou subsequentes, este material pode ser usado como bibliografia básica ou complementar em disciplinas optativas.

O conteúdo está dividido em sete capítulos, iniciando pelo capítulo de conceitos teóricos fundamentais. Logo em seguida, o segundo capítulo propõe estratégias para implementação de grafos como estruturas de dados. No capítulo seguinte, são apresentados os primeiros algoritmos — as buscas em largura e profundidade — juntamente com aplicações práticas destes tipos de buscas em grafos. Questões estruturais de grafos são tratadas no quarto capítulo, em especial a conexidade em grafos não-dirigidos e dirigidos. O quinto capítulo volta a ter um enfoque teórico mais detalhado para apresentar grafos e ciclos eulerianos e hamiltonianos, juntamente com estudos de caso envolvendo jogos de tabuleiro. Após este capítulo, são apresentadas as árvores como sendo tipos especiais de grafos, que estão relacionados ao problema prático das árvores geradoras de custo mínimo.

Por fim, o último capítulo traz alguns dos principais problemas de caminhamento em grafos, notadamente o problema do caminho mínimo. Fechando o capítulo, e este livro, são apresentados os problemas do carteiro chinês e do caixeiro viajante.

Ao final de cada capítulo são propostos exercícios e problemas de forma a complementar a compreensão do material exposto no capítulo. Estão incluídas também algumas questões de edições passadas de provas do POSCOMP¹ (Exame Nacional para Ingresso na Pós-graduação em Computação), promovido pela SBC - Sociedade Brasileira de Computação ([SBC, 2022](#)). Fica aqui registrado nosso agradecimento à SBC por ter permitido a inclusão de tais questões na presente obra.

Dedico esta obra a todos os meus alunos do passado, do presente e do futuro, pois sua necessidade de conhecimento, suas dúvidas e suas dificuldades foram os principais motivadores para que este projeto se concretizasse.

O autor.

¹ Nas seções de exercícios, essas questões sempre iniciam com a legenda “(POSCOMP – XXXX)”, em que XXXX é o ano de realização da prova.

1 Conceitos Fundamentais

Neste primeiro capítulo veremos uma introdução aos conceitos fundamentais da Teoria dos Grafos, a começar pelo próprio conceito de grafo. Iniciaremos mostrando fatos históricos desta área de estudo, que remonta ao século XVIII. Veremos também alguns exemplos simples de aplicações da Teoria dos Grafos na modelagem de problemas de logística, química, eletrônica, e ciência da computação.

1.1 Introdução

A primeira notícia que se tem do uso de grafos refere-se ao ano de 1736 ([BIGGS; LLOYD; WILSON, 1986](#)), quando Leonhard Euler publicou um artigo intitulado “*Solutio problematis ad geometriam situs pertinentis*” (“A solução de um problema relacionado à geometria de posição”) no qual ele discutia a solução do problema das “sete pontes de Königsberg”. A cidade de Königsberg na Prússia continha uma ilha central chamada Kneiphof. Em volta dela corria o rio Pregel, que depois era dividido em duas partes pela ilha, conforme pode ser visto no mapa da [Figura 1.1](#).

Figura 1.1 – Gravura da cidade de Königsberg, no século XVIII

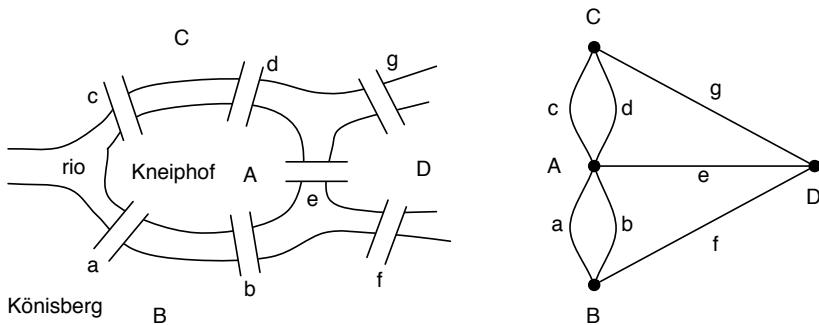


Fonte: [Paoletti \(2006\)](#)

As quatro partes da cidade, que chamaremos de A , B , C e D eram conectadas por sete pontes (a, b, c, d, e, f, g), conforme mostrado no diagrama da parte esquerda da [Figura 1.2](#). O problema consistia em determinar se é possível sair de qualquer ponto da cidade de Königsberg, atravessar cada uma das 7 pontes somente uma vez e retornar ao ponto de partida. Em 1730, Euler demonstrou que era impossível e o fez por meio de

um diagrama. Para tal, não importava qual o caminho a ser percorrido dentro da cidade, mas sim quando o caminho incluía a passagem por uma ponte. Euler reduziu cada uma das 4 regiões de terra a pontos, e ligou os pontos com linhas, para representar cada uma das pontes. Isto resultou no diagrama do lado direito da [Figura 1.2](#). Este diagrama é tido como a primeira referência que se tem aos grafos como se conhece hoje, mas o que vem a ser exatamente um grafo? Veremos uma definição formal na próxima seção.

Figura 1.2 – Diagrama representando o problema das pontes de Königsberg



Fonte: o autor.

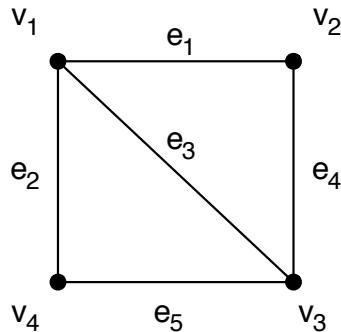
1.2 Grafos

Um grafo é um conceito, uma estrutura, uma ideia, mais precisamente uma abstração matemática que modela relações simétricas dois a dois entre determinados elementos de um conjunto. Apesar desta definição aparentemente simples, os grafos podem representar uma gama enorme de problemas teóricos e práticos. Dependendo do problema, podemos reduzi-lo a um modelo de grafo, e eventualmente, utilizar teoremas e algoritmos dentro da própria teoria para obter conclusões e soluções dos problemas. A seguir, temos uma definição mais formal de um grafo.

Definição 1.1 (Grafo). *Um grafo $G = (V, E)$ é definido por um conjunto não vazio V de vértices e um conjunto E de arestas, em que cada aresta $e \in E$ é definida por um par não-ordenado de vértices (u, v) , sendo ambos u e $v \in V$.*

Para fins didáticos, normalmente os grafos são representados graficamente por desenhos formados por pontos representando os vértices e linhas representando as arestas. Devido à definição apresentada acima, podemos concluir que as linhas (arestas) nunca ligam mais que dois vértices. Numa representação gráfica de um grafo, geralmente a disposição geométrica dos vértices e arestas não é importante, mas sim a conectividade entre estes elementos, isto é, qual vértice está ligado com qual aresta. A [Figura 1.3](#) mostra um exemplo de representação gráfica de um grafo com 4 vértices e 5 arestas:

Figura 1.3 – Exemplo de grafo não dirigido



Fonte: o autor.

Um grafo também pode ser descrito por seus dois conjuntos de vértices e arestas. Para o exemplo da [Figura 1.3](#), teríamos os conjuntos: $V = \{v_1, v_2, v_3, v_4\}$ e $E = \{(v_1, v_2), (v_1, v_4), (v_1, v_3), (v_2, v_3), (v_4, v_3)\}$. Note que, como cada aresta é formada por um par não-ordenado de vértice, tanto faz definirmos $e_1 = (v_1, v_2)$ ou $e_1 = (v_2, v_1)$.

Exercício 1.1. Desenhe o grafo não-dirigido abaixo, sendo dados os conjuntos de vértices e arestas: $V = \{1, 2, 3, 4, 5\}$ e $E = \{(1, 2), (1, 4), (1, 5), (2, 3), (3, 4), (4, 4)\}$.

Cabe ressaltar que a terminologia utilizada na Teoria dos Grafos não é padronizada e cada autor acaba usando seus próprios termos, o que pode causar uma certa confusão. Alguns autores, por exemplo, chamam arestas de arcos e vértices de nós, o que é válido desde que seja mantida consistência ao longo da obra. No presente texto, optamos, na medida do possível, por utilizar os termos mais amplamente utilizados na bibliografia.

Abaixo, seguem algumas definições básicas a respeito da estrutura dos grafos:

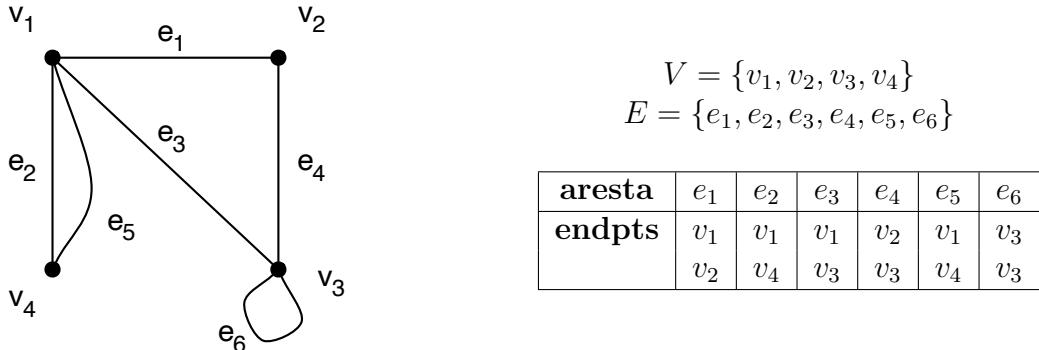
Definição 1.2 (Incidência). É a propriedade de uma aresta estar conectada a um vértice.

Definição 1.3 (Adjacência). É a propriedade de dois vértices estarem conectados por intermédio de uma aresta. Dois vértices adjacentes também podem ser chamados de *vizinhos*.

Exemplo 1.1. Na [Figura 1.4](#), as arestas e_1 e e_4 são incidentes ao vértice v_2 ; o vértice v_2 é adjacente aos vértices v_1 e v_3 , e vice-versa. No caso de digrafos a relação de adjacência não é simétrica. No digrafo da [Figura 1.5](#), o vértice v_1 é adjacente a v_2 , mas v_2 não é adjacente a v_1 .

A tabela mostrada na [Figura 1.4](#) é chamada de tabela de incidência, ela representa a **função de incidência** do grafo. A função de incidência, denotada por $\text{endpts}(e)$ mapeia

Figura 1.4 – Grafo não dirigido - incidência e adjacência



Fonte: o autor.

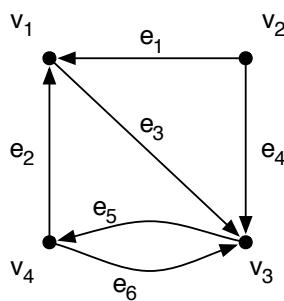
cada aresta e para um subconjunto de um ou dois elementos de V . Na [Figura 1.4](#) temos, por exemplo, $\text{endpts}(e_1) = \{v_1, v_2\}$ e $\text{endpts}(e_6) = \{v_3\}$.

1.3 Grafos dirigidos

Dependendo do tipo de problema a ser modelado, a relação entre elementos do problema (vértices) pode não ser simétrica. Neste caso, os grafos são “dirigidos”, isto é, as arestas apresentam um sentido que vai do vértice origem para o vértice destino. Estes grafos também são chamados de **digrafos**.

Definição 1.4 (Digrafo ou Grafo Dirigido). *Um grafo dirigido $G = (V, E)$ é definido por um conjunto não vazio V de vértices e um conjunto E de arestas, em que cada aresta $e \in E$ é definida por um par ordenado de vértices (u, v) , sendo ambos u e $v \in V$. O vértice u é o vértice origem, e v é o vértice destino da aresta e .*

Figura 1.5 – Exemplo de grafo dirigido (ou digrafo)



Fonte: o autor.

Na representação gráfica de digrafos, as arestas são desenhadas como setas, apontando na direção origem-destino, conforme ilustra o exemplo da [Figura 1.5](#). Para este grafo,

os conjuntos de vértices e arestas são: $V = \{v_1, v_2, v_3, v_4\}$ e $E = \{(v_2, v_1), (v_4, v_1), (v_1, v_3), (v_2, v_3), (v_3, v_4), (v_4, v_3)\}$. Note que agora, sendo um grafo dirigido, $e_1 = (v_2, v_1) \neq (v_1, v_2)$.

Para digrafos, o conceito de **adjacência** é um pouco diferente daquele visto anteriormente. Aqui, dizemos que um vértice v_j é adjacente a um vértice v_i se houver pelo menos uma aresta (v_i, v_j) . Portanto, na [Figura 1.6](#), o vértice v_j é adjacente a v_i , porém v_i não é adjacente a v_j .

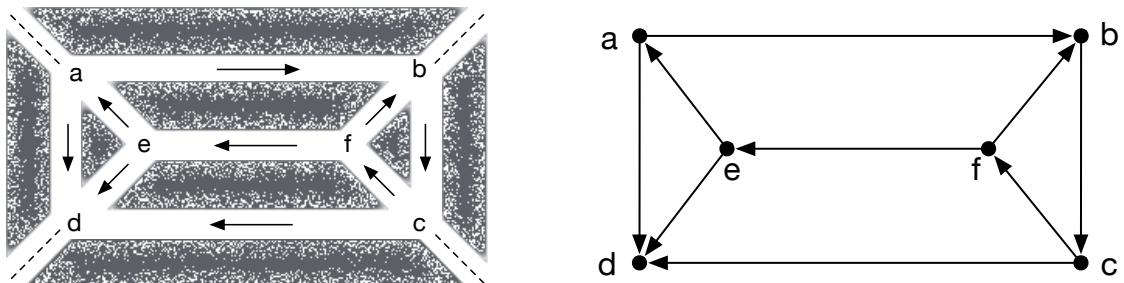
[Figura 1.6 – Adjacência de vértices em digrafos.](#)



Fonte: o autor.

Exemplo 1.2. Considere o seguinte sistema de ruas de mão única de uma cidade, conforme ilustrado na [Figura 1.7](#). Como as ruas são de mão única, não poderíamos representar este sistema como um grafo (não dirigido). Entretanto, se considerarmos que a direção das ruas pode ser representada por arestas dirigidas, podemos modelar este sistema como um grafo dirigido conforme a figura abaixo:

[Figura 1.7 – Exemplo de modelagem de sistema de trânsito](#)



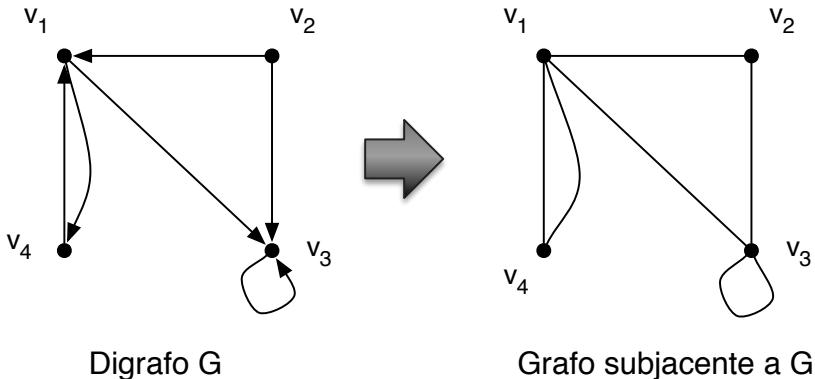
Fonte: o autor.

Exercício 1.2. Desenhe o seguinte grafo dirigido, sendo dados seus conjuntos de vértices e arestas: $V = \{1, 2, 3, 4\}$ e $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$.

Definição 1.5 (Grafo subjacente). Dado um grafo dirigido G , seu grafo subjacente é um grafo não dirigido formado pelo mesmo conjunto de vértices de G , porém com cada aresta dirigida de G substituída por uma aresta não-dirigida. A [Figura 1.8](#) exibe um exemplo de grafo subjacente obtido a partir de um grafo dirigido G .

Definição 1.6 (Grafo transposto). Dado um grafo dirigido $G = (V, E)$, seu grafo transposto é definido por $G^T = (V, E^T)$, em que $E^T = \{(u, v) : (v, u) \in E\}$.

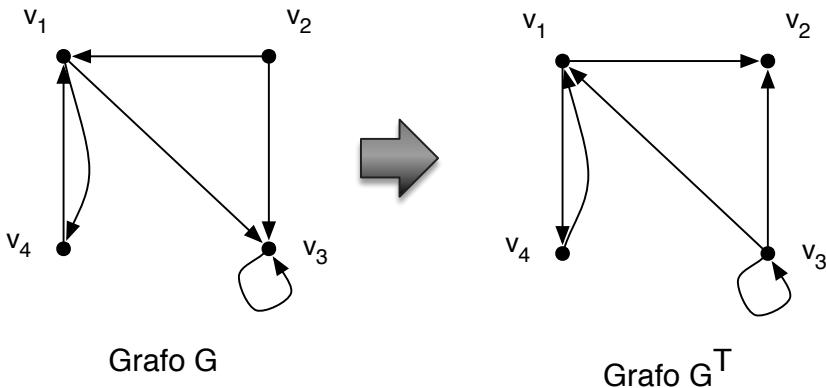
Figura 1.8 – Grafo subjacente a um grafo dirigido



Fonte: o autor.

Em outras palavras, um grafo transposto a um grafo dirigido é obtido invertendo-se o sentido de todas as suas arestas. A [Figura 1.9](#) ilustra um exemplo de grafo transposto a um grafo dirigido.

Figura 1.9 – Grafo transposto a um grafo dirigido



Fonte: o autor.

1.4 Laços e arestas paralelas

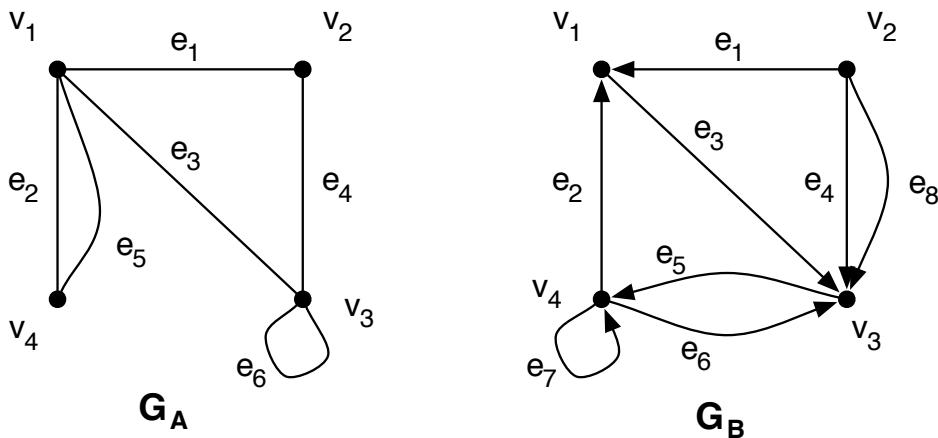
Eventualmente grafos e digrafos podem apresentar alguns tipos especiais de arestas:

Definição 1.7 (Laço). *Um laço é uma aresta que conecta um vértice a ele mesmo. Os laços também podem ser chamados de **loops** ou **self-loops**.*

Definição 1.8 (Arestas paralelas). *Arestas paralelas são conjuntos de duas ou mais arestas que conectam o mesmo par de vértices.*

Exemplo 1.3. Na [Figura 1.10](#), as arestas e_2 e e_5 do grafo não-dirigido G_A são arestas paralelas, e a aresta e_6 é um laço. No digrafo G_B , as arestas e_4 e e_8 são paralelas e a aresta e_7 é um laço. Note que as arestas e_5 e e_6 do digrafo G_B não são paralelas, pois por serem dirigidas, não conectam exatamente o mesmo par de vértices.

Figura 1.10 – Laços e arestas paralelas



Fonte: o autor.

De acordo com a existência destes tipos especiais de arestas, podemos classificar os grafos e digrafos em dois grandes grupos: grafos simples e multigrafos.

Definição 1.9 (Grafo simples). *Um grafo simples (ou digrafo simples) é um grafo que não contém laços e nem arestas paralelas.*

Definição 1.10 (Multigrafo). *Um multigrafo é um grafo (ou digrafo) que contém pelo menos um laço ou um conjunto de arestas paralelas.*

1.5 Parâmetros quantitativos

Normalmente, precisamos utilizar parâmetros quantitativos a respeito de um grafo, consequentemente, do problema modelado pelo grafo. Abaixo estão descritos os principais parâmetros utilizados neste livro.

Definição 1.11 (Ordem). A **ordem** de um grafo representa a cardinalidade do conjunto V . Em outras palavras, a ordem representa a quantidade de vértices do grafo, a qual é denotada por $|V|$ ou pela letra n .

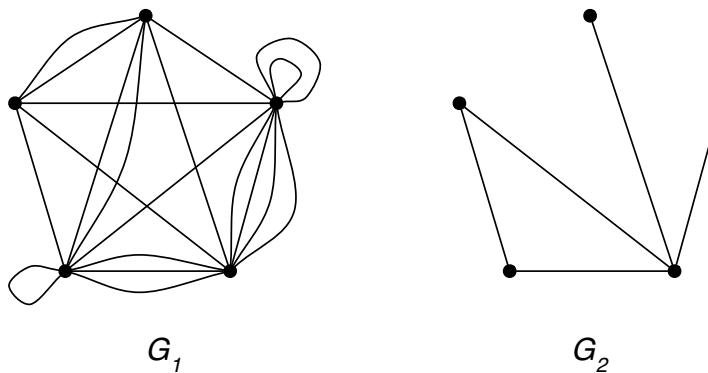
Definição 1.12 (Tamanho). O **tamanho** de um grafo representa a cardinalidade do conjunto E . Em outras palavras, o tamanho representa a quantidade de arestas do grafo, a qual é denotada por $|E|$ ou pela letra m .

Definição 1.13 (Densidade). A **densidade** de um grafo é dada em função da relação entre sua ordem e seu tamanho, ou seja, representa a proporção entre quantidade de arestas e vértices.

Em relação a esta proporção que define a densidade, um grafo é dito **denso** quando se tem muitas arestas para uma determinada quantidade de vértices. Se, ao contrário, o grafo tiver poucas arestas para uma determinada quantidade de vértices, ele é chamado de grafo **esparso**. A densidade não é uma medida exata, mas considera-se um grafo denso quando a quantidade de arestas é próxima ou maior que $|V|^2$. Por outro lado, o grafo é considerado esparso se o número de arestas for muito menor que $|V|^2$.

Exemplo 1.4. A [Figura 1.11](#) mostra dois grafos distintos, cada um deles tem 5 vértices. Por outro lado, o grafo G_1 possui 20 arestas enquanto o grafo G_2 tem apenas 5. Comparando os dois grafos, podemos concluir que, G_1 é denso e o grafo G_2 pode ser considerado esparso.

Figura 1.11 – Grafos denso e esparso



Fonte: o autor.

Em algumas situações, é importante saber quantas ligações de arestas existem em um determinado vértice. Por exemplo, considere um grafo representando um mapa rodoviário, com vértices para cidades e arestas para estradas que ligam diretamente duas cidades. A quantidade de arestas ligando um determinado vértice representa a quantidade de estradas pelas quais se pode sair da cidade. Esta quantidade é o grau do vértice.

Definição 1.14 (Grau de vértice). O grau de um vértice é igual à quantidade de arestas incidentes ao vértice, com cada laço sendo contado duas vezes. O grau de um vértice pode ser denotado por $\text{grau}(v)$.

Definição 1.15 (Sequência de graus). Dado um grafo não dirigido, sua sequência de graus é obtida listando-se os graus de todos os seus vértices em ordem não-decrescente, com repetições caso necessário.

Exemplo 1.5. O grafo G_A da Figura 1.10, tem $|V| = n = 4$, $|E| = m = 6$, $\text{grau}(v_1) = 4$ e $\text{grau}(v_2) = 2$, $\text{grau}(v_3) = 4$ e $\text{grau}(v_4) = 2$. Note que este grafo tem ordem e tamanho iguais a 4 e 6, respectivamente ($|V| = n = 4$ e $|E| = v = 6$). Podemos observar que a soma dos graus de todos os vértices é igual a 12, ou seja exatamente $2m$. Sua sequência de graus é $(2, 2, 4, 4)$.

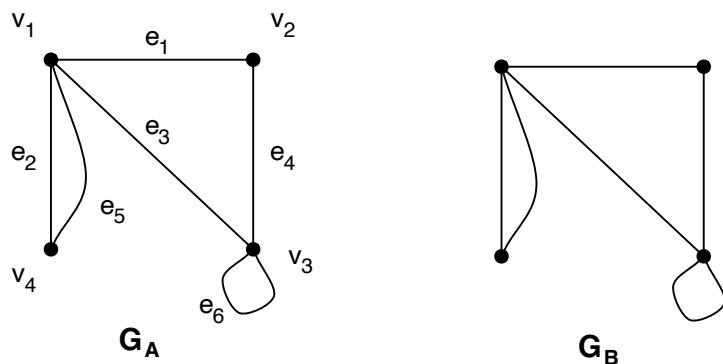
Definição 1.16 (Grau de vértice em digrafos). Em grafos dirigidos, o conceito de grau de vértice é um pouco diferente: o grau de entrada de um vértice é igual à quantidade de arestas dirigidas que chegam no vértice e o grau de saída é igual à quantidade de arestas que partem do vértice. Estes parâmetros são denotados por $\text{grau}_e(v)$ e $\text{grau}_s(v)$, respectivamente.

Exemplo 1.6. O digrafo G_B da Figura 1.10, tem $|V| = n = 4$, $|E| = m = 8$, $\text{grau}_e(v_2) = 0$, $\text{grau}_s(v_2) = 3$, $\text{grau}_e(v_4) = 2$ e $\text{grau}_s(v_4) = 3$.

1.6 Grafos rotulados e não-rotulados

Na Figura 1.12, o grafo G_A é rotulado. Isto significa que seus vértices e arestas tem algum tipo de identificação, ou rótulo. Em várias situações, não é necessário identificar cada vértice e aresta individualmente, neste caso os grafos são chamados não-rotulados. Como exemplo, temos o grafo G_B da Figura 1.12. Para fins de estudo teórico, muitas vezes não é necessário rotular os vértices. Por outro lado, a identificação explícita de vértices e arestas se faz necessária quando estamos modelando um problema real ou então quando estamos demonstrando ou executando algum algoritmo. Ao longo do texto, vamos apresentar grafos rotulados e não-rotulados de acordo com a necessidade ou a conveniência.

Figura 1.12 – Grafos rotulado e não-rotulado



Fonte: o autor.

1.7 Isomorfismo

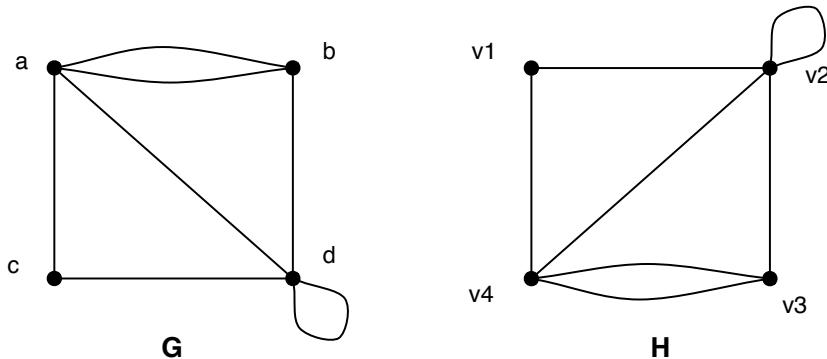
Quando um grafo é rotulado, normalmente seus vértices representam elementos de um problema concreto, portanto não é totalmente correto afirmarmos que dois grafos são iguais, se os nomes de seus vértices são diferentes. A rigor, um grafo rotulado seria fruto da modelagem matemática de determinado problema.

Por outro lado, em algumas situações, é importante verificar semelhanças estruturais entre diferentes grafos. Assim, ao invés de uma igualdade absoluta entre dois grafos, vamos procurar uma igualdade *estrutural*. Este tipo de conceito é chamado *isomorfismo*.

Definição 1.17 (Isomorfismo). *Dois grafos G e H são isomorfos se H puder ser obtido renomeando-se os vértices de G . Isto é, se existir correspondência de um para um entre os vértices de G e H de tal forma que o número de arestas conectadas a cada par de vértices em G é igual ao número de arestas conectadas aos pares de vértices correspondentes em H . Tal correspondência de um para um constitui o isomorfismo, é denotada por $G \simeq H$.*

Exemplo 1.7. Na [Figura 1.13](#), os grafos G e H são isomorfos (ou $G \simeq H$). Note que não podemos dizer que são iguais porque seus conjuntos de vértices e arestas são diferentes (por exemplo $V_G = \{a, b, c, d\}$ e $V_H = \{v_1, v_2, v_3, v_4\}$, mas eles tem estruturas iguais.

Figura 1.13 – Isomorfismo $G \simeq H$



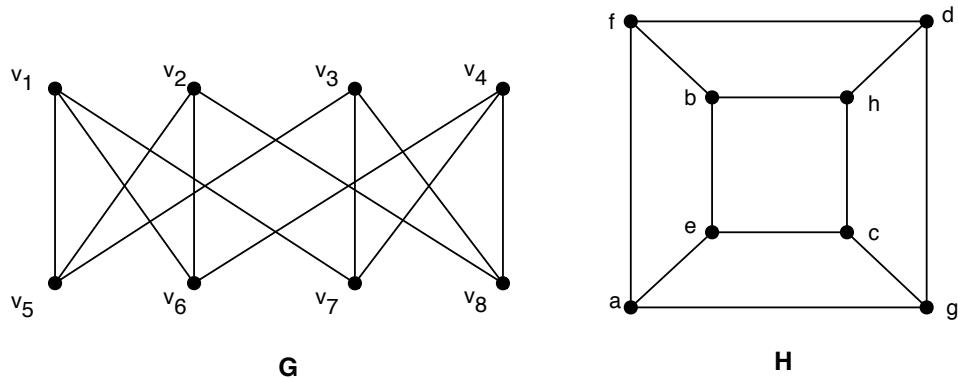
Fonte: o autor.

Podemos provar isto encontrando uma função bijetora f mapeando os vértices de um grafo para o outro. Assim, poderíamos estabelecer a seguinte correspondência:

$$a \rightarrow v_4, \quad b \rightarrow v_3, \quad , \quad c \rightarrow v_1, \quad d \rightarrow v_2$$

No grafo G , o vértice c é adjacente aos vértices a e d mas não é adjacente ao vértice b . Se observarmos o grafo H , constatamos que $v_1 = f(c)$ é adjacente aos vértices $v_2 = f(d)$ e $v_4 = f(a)$, mas não é adjacente ao vértice $v_3 = f(b)$. Neste sentido, a bijeção $f : V_G \rightarrow V_H$ também deve ser consistente na bijeção das adjacências.

Exercício 1.3. Dados os grafos G e H abaixo, prove que $G \simeq H$.



Sugestão: procure encontrar um mapeamento de V_G para V_H .

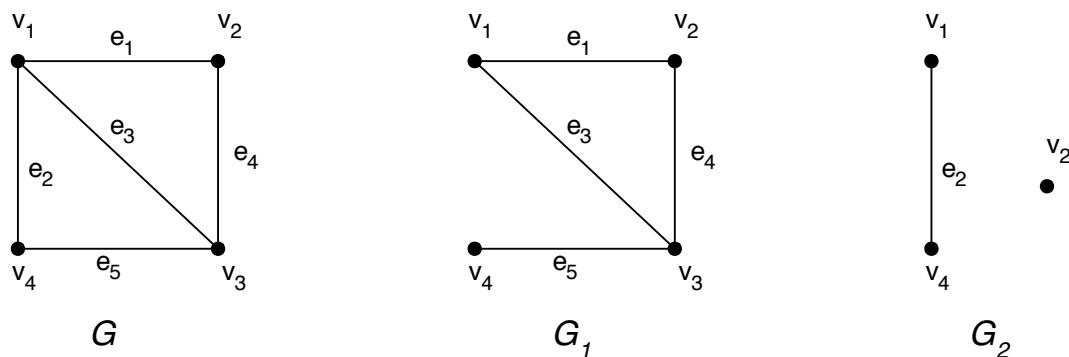
1.8 Subgrafos

Muitas vezes queremos abordar partes menores de um problema. O conceito de subgrafos está diretamente relacionado ao conceito de subconjuntos de um conjunto, ou subgrupos de um grupo. Na teoria dos grafos podemos utilizar a seguinte definição:

Definição 1.18 (Subgrafo). Um subgrafo de um grafo $G = (V, E)$ é um grafo cujos vértices são vértices de G e cujas arestas são arestas de G .

Exemplo 1.8. A Figura 1.14 exemplifica este conceito. O grafo G é composto pelos conjuntos $V = \{v_1, v_2, v_3, v_4\}$ e $E = \{e_1, e_2, e_3, e_4, e_5\}$. O primeiro subgrafo G_1 , é formado pelos seguintes subconjuntos de V e E : $V_1 = \{v_1, v_2, v_3, v_4\}$ e $E_1 = \{e_1, e_3, e_4, e_5\}$; o segundo subgrafo G_2 , é formado por $V_2 = \{v_1, v_2, v_4\}$ e $E_2 = \{e_2\}$.

Figura 1.14 – Dois subgrafos de um grafo G



Fonte: o autor.

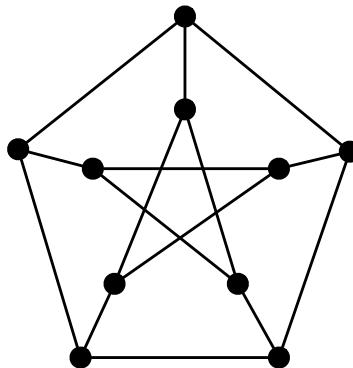
1.9 Classes de grafos

Existem alguns tipos de grafos que compartilham certas características estruturais em comum, tais grafos podem ser classificados em grupos com mesmas características, o que geralmente facilita o estudo teórico de propriedades e algoritmos. A seguir, veremos algumas dessas classes.

Definição 1.19 (Grafo regular). *Um grafo é dito **regular** se todos os seus vértices tem o mesmo grau. Um grafo é chamado de **r-regular** ou **regular de grau r** se o grau de cada um de seus vértices é igual a r.*

Exemplo 1.9. A [Figura 1.15](#) mostra um exemplo de grafo 3-regular. Este grafo é chamado de **Grafo de Petersen** ([PETERSEN, 1898](#)), ele tem propriedades interessantes sob vários aspectos da Teoria dos Grafos, sendo comumente utilizado para exemplificar teoremas e testar conjecturas. O nome deste grafo é uma homenagem ao matemático dinamarquês Julius Petersen e foi apresentado em um artigo científico de sua autoria, em 1898.

Figura 1.15 – Grafo de Petersen



Fonte: o autor.

Teorema 1.1. *Seja G um grafo r -regular com n vértices. Então G tem $nr/2$ arestas.*

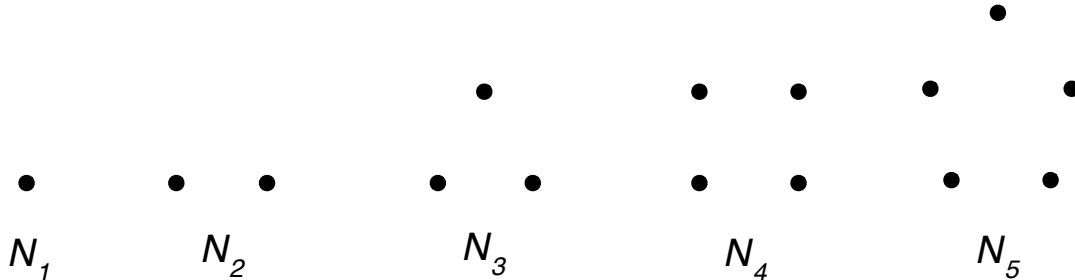
Prova. *Seja G um grafo com n vértices, cada um com grau r , então a soma dos graus de todos os vértices é igual a nr . Pelo lema do aperto de mão, o número de arestas de um grafo é igual à metade desta soma, portanto igual a $nr/2$.*

Definição 1.20 (Grafo nulo). *Grafo nulo é um grafo sem arestas. O grafo nulo é denotado por N_n em que n representa o número de vértices do grafo¹.*

¹ Um grafo nulo sempre tem vértices isolados, o qual denomina-se grafo 0-regular.

Exemplo 1.10. A [Figura 1.16](#) mostra os grafos nulos N_1 , N_2 , N_3 , N_4 e N_5 .

Figura 1.16 – Grafos nulos



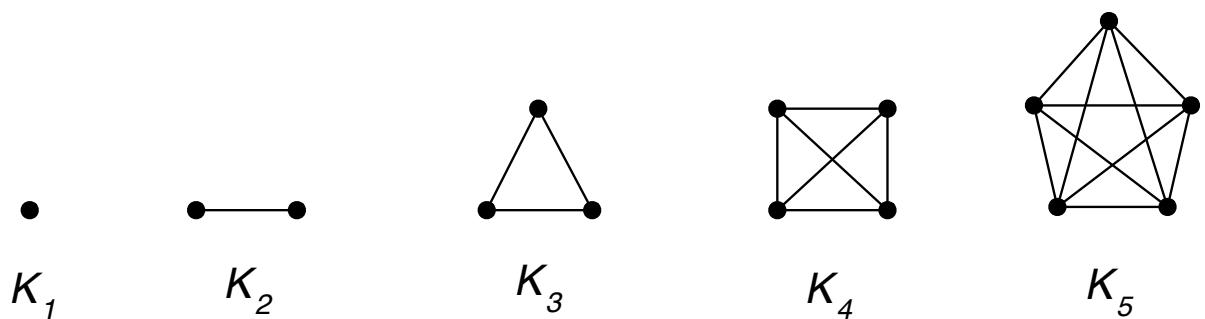
Fonte: o autor.

Definição 1.21 (Grafo completo). *Grafo completo é um grafo simples em que cada par de vértices distintos é adjacente. Nós denotamos o grafo completo com n vértices por K_n .*

Os grafos completos são bastante interessantes no estudo de teoria dos grafos. Note que um grafo completo com n vértices, é o grafo simples mais denso possível para esta quantidade de vértices. É também um grafo $(n - 1)$ -regular.

Exemplo 1.11. A [Figura 1.17](#) ilustra exemplos dos grafos completos K_1 , K_2 , K_3 , K_4 e K_5 .

Figura 1.17 – Grafos completos



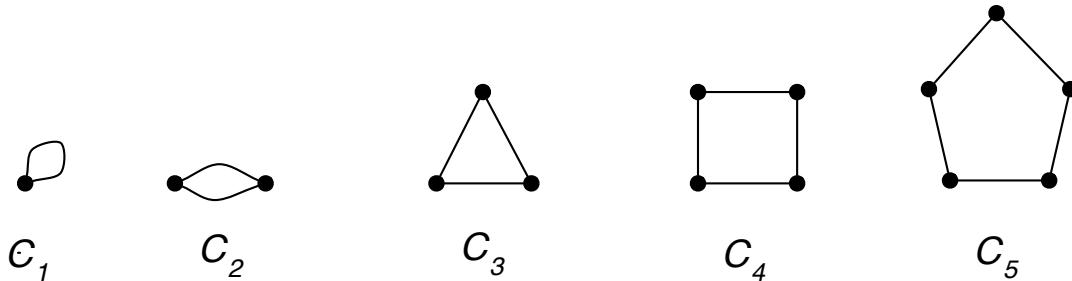
Fonte: o autor.

Exercício 1.4. Desenhe os grafos completos K_6 , K_7 e K_8 .

Definição 1.22 (Grafo ciclo). *Grafo ciclo é um grafo constituído por apenas um ciclo de vértices e arestas². O grafo ciclo com n vértices é denotado por C_n .*

Exemplo 1.12. A [Figura 1.18](#) ilustra exemplos dos grafos ciclo C_1 , C_2 , C_3 , C_4 e C_5 .

Figura 1.18 – Grafos ciclo



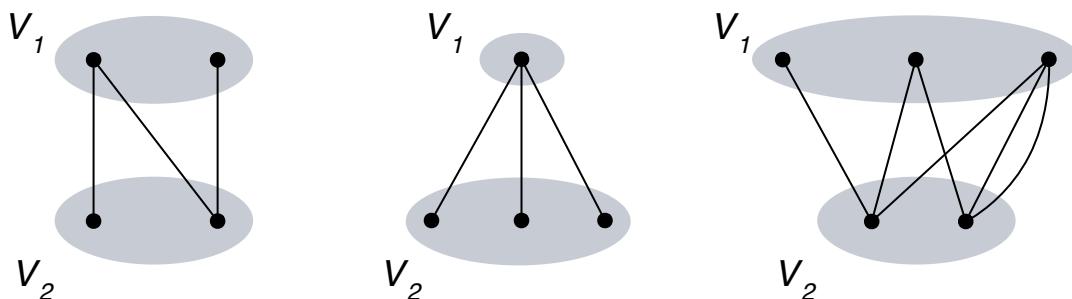
Fonte: o autor.

Definição 1.23 (Grafo bipartido). *Um grafo bipartido $G = (V_1 \cup V_2, E)$ é definido por dois conjuntos de vértices V_1 e V_2 , em que $V_1 \cap V_2 = \emptyset$ e um conjunto E de arestas, tal que, para cada aresta $e \in E$, tem-se que $e = (u, v)$ e $u \in V_1$ e $v \in V_2$. Os conjuntos V_1 e V_2 são chamados de subconjuntos de bipartição de G .*

Note que a definição 1.23 implica em nunca haver arestas entre vértices dos conjuntos V_1 e V_2 ou laços. Porém é possível haver arestas paralelas.

Exemplo 1.13. A [Figura 1.19](#) mostra três exemplos de grafos bipartidos. Os conjuntos de vértices V_1 e V_2 para cada grafo são indicados em cinza.

Figura 1.19 – Grafos bipartidos



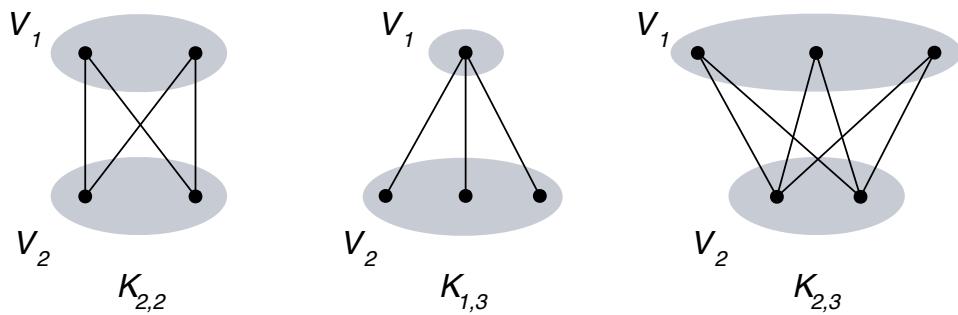
Fonte: o autor.

² Note que um grafo ciclo com n vértices sempre é um grafo 2-regular com n arestas.

Definição 1.24 (Grafo bipartido completo). Um grafo bipartido completo $K_{r,s}$ é um grafo simples bipartido no qual todos os pares de vértices u e v tais que $u \in V_1$ e $v \in V_2$ são adjacentes.

Exemplo 1.14. A [Figura 1.20](#) mostra exemplos dos grafos bipartidos completos $K_{2,2}$, $K_{1,3}$ e $K_{2,3}$, respectivamente.

Figura 1.20 – Grafos bipartidos completos



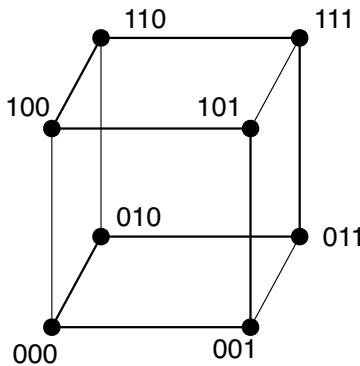
Fonte: o autor.

Exercício 1.5. Desenhe todos os grafos bipartidos completos $K_{r,s}$ com $r \leq 4$ e $s \leq 4$.

Definição 1.25 (Grafo cubo). Um grafo cubo Q_k , também chamado de hipercubo k -dimensional é um grafo cujo conjunto de vértices é formado por palavras binárias de k bits de tal forma que existe uma aresta entre dois vértices se e somente se eles forem diferentes em apenas um bit.

Exemplo 1.15. A [Figura 1.21](#) mostra exemplos do grafo cubo Q_3 com os respectivos valores de 3 bits associados aos vértices.

Figura 1.21 – Grafo cubo Q_3



Fonte: o autor.

Exercício 1.6. Desenhe os grafos cubo Q_1 , Q_2 e Q_4 . Dica: construa o grafo Q_4 a partir de dois grafos Q_3 .

1.10 Percursos, trilhas e caminhos

Muitas aplicações na teoria dos Grafos consistem em transitar pelo grafo passando por vértices e arestas. Eventualmente problemas de grafos envolvem noções de distância entre elementos do grafo, mensuradas em quantidades de vértices, arestas ou até em outros parâmetros relevantes para determinado problema. A seguir veremos algumas definições com o intuito de apresentar estes conceitos.

Definição 1.26 (Percorso). *Em um grafo não dirigido, podemos definir um percurso $P = \langle v_0, e_1, v_1, e_2, \dots, v_{n-1}, e_n, v_n \rangle$ como sendo uma sequência de vértices e arestas tal que o vértice final de uma aresta no percurso é igual ao vértice inicial da próxima aresta.*

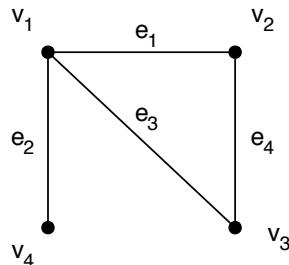
Dependendo da aplicação, pode ser suficiente representar percursos de forma concisa (ou abreviada), representando-os apenas por sequências de vértices ou de arestas, por exemplo: $P = \langle v_0, v_1, \dots, v_n \rangle$ ou $P = \langle e_1, e_2, \dots, e_n \rangle$. Também podemos representar percursos de v_0 para v_n como $v_0 \rightsquigarrow v_n$.

Definição 1.27 (Comprimento). *Em um percurso, seu comprimento é igual à quantidade de arestas percorridas no percurso. O comprimento do percurso é dado por $\delta(s, v)$. Se não existir percurso possível entre os vértices v_0 e v_n , então $\delta(v_0, v_n) = \infty$.*

Definição 1.28 (Percorso fechado). *Um percurso é dito **fechado** se seu primeiro vértice é igual ao último, isto é: $v_0 = v_n$. Caso contrário, o percurso é **aberto**.*

Exemplo 1.16. Na [Figura 1.22](#), $P = \langle v_1, e_2, v_4, e_2, v_1, e_1, v_2, e_4, v_3, e_3, v_1, \rangle$ é um exemplo de percurso fechado com comprimento igual a 5 em um grafo de 4 vértices. Note que a sequência $\langle v_1, v_4, v_3, v_2 \rangle$ **não** é um percurso, pois não há aresta entre v_4 e v_3 .

Figura 1.22 – Percurso em um grafo



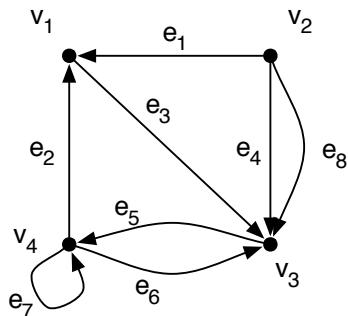
Fonte: o autor.

Definição 1.29 (Percorso dirigido). Analogamente, em um grafo dirigido, um percorso dirigido $P = \langle v_0, e_1, v_1, e_2, \dots, v_{n-1}, e_n, v_n \rangle$ é uma sequência de vértices e arestas dirigidas tal que o vértice final de uma aresta dirigida no percurso é igual ao vértice inicial da próxima aresta dirigida.

Em um percorso dirigido, é necessário respeitar a direção de cada aresta definida pelos seus vértices de origem e destino. Ou seja, não podemos “caminhar na contramão”.

Exemplo 1.17. Na [Figura 1.23](#), $P = \langle v_1, e_3, v_3, e_5, v_4, e_7, v_4, e_6, v_3 \rangle$ é um exemplo de percorso dirigido aberto com comprimento igual a 4. Note que a sequência $\langle v_1, e_3, v_3, e_4, v_2 \rangle$ não é um percorso, pois não podemos caminhar no sentido contrário das arestas (aresta e_4).

Figura 1.23 – Percurso dirigido em um grafo



Fonte: o autor.

Definição 1.30 (Trilha). Em um grafo, uma trilha é um percurso sem repetição de arestas.

Definição 1.31 (Caminho). Em um grafo, um caminho é uma trilha sem repetição de vértices (exceto pelos vértices inicial e final, eventualmente).

Definição 1.32. Um percurso, trilha ou caminho é **trivial** se tiver somente um vértice e nenhuma aresta.

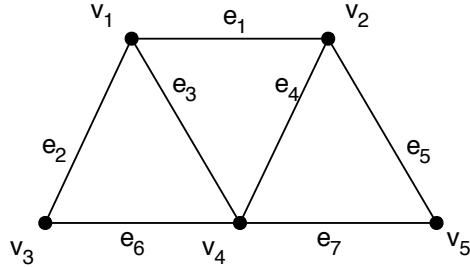
Não existe consenso entre autores a respeito da terminologia para percurso, trilha e caminho. Por exemplo, alguns autores usam caminho simples ao invés de caminho e caminho ao invés de trilha.

Definição 1.33 (Ciclo). Em um grafo, um ciclo é um caminho fechado com mais de um vértice, ou seja, é um caminho fechado não trivial.

Definição 1.34 (Círculo). Em um grafo, um circuito é uma trilha fechada.

Exemplo 1.18. Na Figura 1.24, temos os seguintes exemplos (note que um percurso não é uma trilha e uma trilha não é um caminho): *Percorso:* $P = \langle e_1, e_4, e_3, e_1, e_5 \rangle$; *Trilha:* $T = \langle e_3, e_4, e_5, e_7, e_6 \rangle$; *Caminho:* $C_a = \langle e_5, e_7, e_6, e_2 \rangle$; *Ciclo:* $C_f = \langle e_5, e_7, e_6, e_2, e_1 \rangle$ e *Círculo:* $C_c = \langle e_3, e_4, e_5, e_7, e_6, e_2 \rangle$.

Figura 1.24 – Caminho, trilha, ciclo e circuito em um grafo

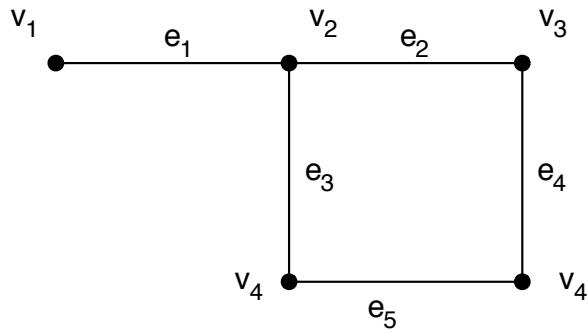


Fonte: o autor.

Definição 1.35 (Caminho mínimo). *Dados dois vértices s e v , o caminho mínimo $s \rightsquigarrow v$ é definido como o caminho de menor comprimento dentre todos os caminhos existentes entre s e v . Note que, dependendo da estrutura do grafo, pode haver mais de um caminho mínimo $s \rightsquigarrow v$.*

Exemplo 1.19. Na Figura 1.25, existem dois caminhos entre os vértices v_1 e v_4 , são eles: $C_1 = \langle v_1, e_1, v_2, e_2, v_3, e_3, v_4 \rangle$, com comprimento igual a 4 e $C_2 = \langle v_1, e_1, v_2, e_3, v_4 \rangle$, com comprimento igual a 2. Como não é possível encontrar um caminho menor que C_2 , este é o caminho mínimo entre os vértices v_1 e v_4 .

Figura 1.25 – Exemplo menor caminho



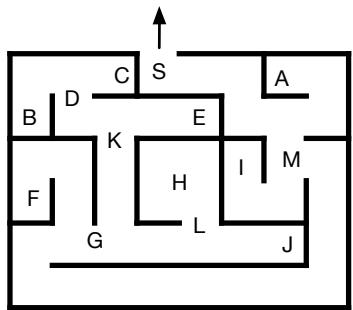
Fonte: o autor.

1.11 Aplicações

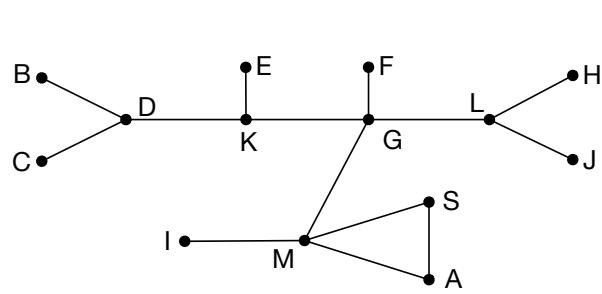
1.11.1 Labirintos

Um labirinto é um conjunto complexo de caminhos, corredores ou túneis onde é muito fácil se perder. A [Figura 1.26a](#) mostra um exemplo de um labirinto. Podemos utilizar um grafo para representar os becos sem saída e as possibilidades de escolhas de caminhos no labirinto em cada bifurcação. Por exemplo, na bifurcação K, podemos ir para D, E ou G.

Figura 1.26 – Exemplo de modelagem de labirinto



(a) Exemplo de labirinto



(b) Grafo do labirinto

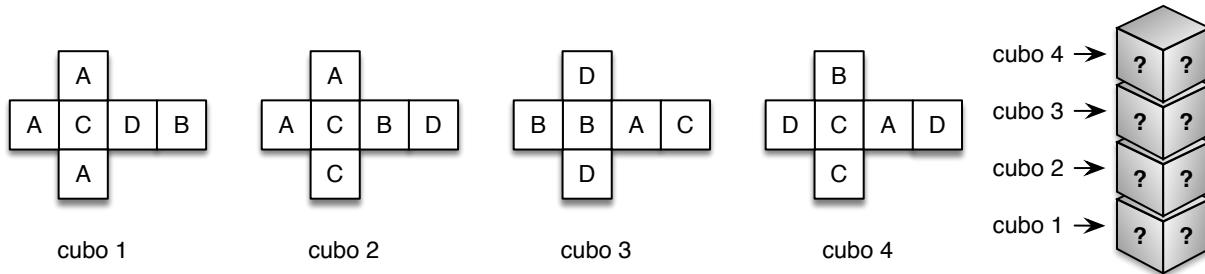
Fonte: o autor.

Para modelar o grafo, os vértices representam os becos sem saída e as bifurcações. As arestas ligam os vértices se e somente se for possível ir de um ponto a outro no labirinto sem passar por nenhuma bifurcação no meio do caminho. A [Figura 1.26b](#) mostra o grafo que representa o labirinto da figura anterior. Se quisermos, por exemplo encontrar um caminho do local B até a saída do labirinto, basta encontrar um caminho entre os vértices B e S no grafo.

1.11.2 Problema dos 4 cubos

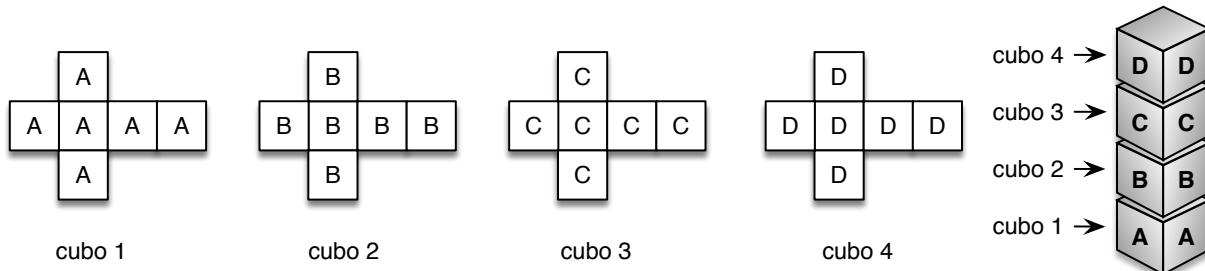
O problema dos quatro cubos, descrito por [Aldous, Best e Wilson \(2003\)](#), é um jogo do tipo quebra-cabeças no qual são fornecidos 4 cubos com faces marcadas pelas letras A, B, C ou D. Dados os 4 cubos, o problema consiste em empilhá-los de forma que em cada uma das 4 faces da pilha (esquerda, direita, frente e atrás) apareçam as 4 letras, para isso é necessário encontrar uma orientação para cada cubo. A [Figura 1.27](#) exibe uma possível instância deste problema: à esquerda são mostrados 4 cubos planificados e à direita é mostrada a pilha que deve ser construída.

Figura 1.27 – Instância do problema dos 4 cubos

Fonte: adaptado de [Aldous, Best e Wilson \(2003\)](#).

Observe que algumas instâncias do problema não tem solução, ou resultam em uma solução trivial. Na [Figura 1.28](#), cada cubo dispõe em todos os seus lados a marcação de apenas uma letra. Como cada um deles tem uma letra diferente, a solução do problema é trivial. Basta empilhá-los sem qualquer preocupação com a orientação de cada um e cada face da pilha vai exibir as 4 letras.

Figura 1.28 – Instância do problema dos 4 cubos com solução trivial

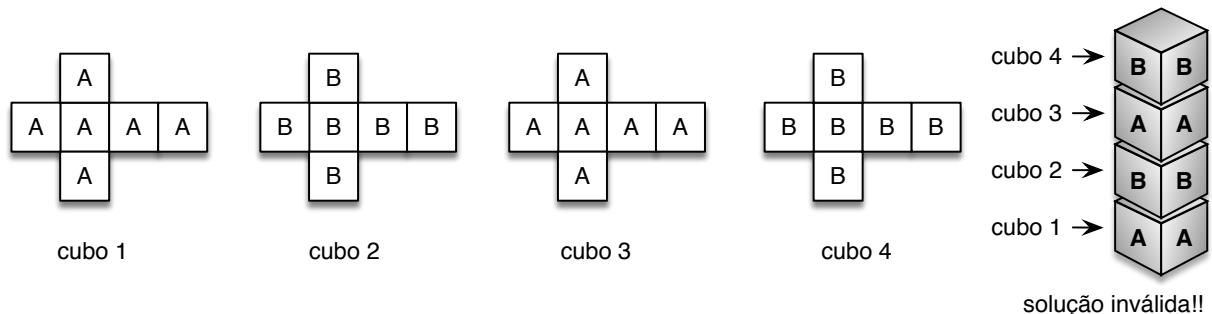
Fonte: adaptado de [Aldous, Best e Wilson \(2003\)](#).

Por outro lado, a [Figura 1.29](#) exibe uma instância sem solução. Neste caso, os cubos 1 e 3 tem todos os lados marcados com a letra A, os cubos 2 e 4 tem todos os lados marcados com a letra B. Com isso, este problema não tem solução, pois nunca será possível exibir as 4 letras diferentes na pilha de cubos.

Com este problema, podemos exemplificar uma aplicação de conceitos fundamentais da Teoria dos Grafos para solucionar uma situação prática. O primeiro passo é modelar o problema como um ou mais grafos, em seguida verificar alguma propriedade, aplicar algum teorema ou executar um algoritmo para buscar a solução desejada.

Precisamos encontrar elementos que, combinados dois a dois, formariam vértices e arestas. No caso do problema dos 4 cubos, uma relação importante e relevante para a busca de uma solução seria o fato de que duas faces opostas de um cubo formam um par específico de letras. Por exemplo: os lados opostos do cubo 1 da [Figura 1.27](#) formam os

Figura 1.29 – Instância do problema dos 4 cubos sem solução

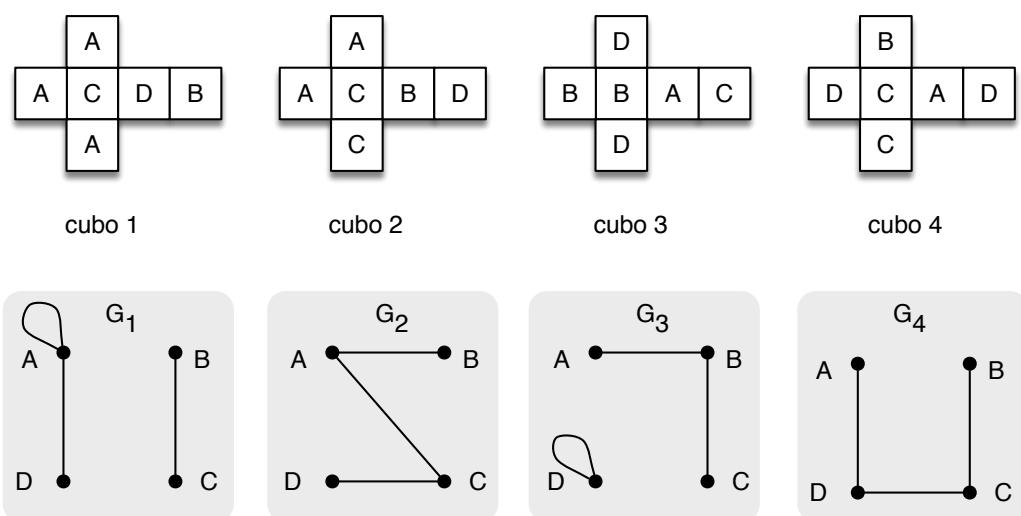


Fonte: adaptado de [Aldous, Best e Wilson \(2003\)](#).

pares (A, A), (C, B) e (A, D). Em razão disso, ao escolhermos uma face para um dos lados da pilha, já estamos escolhendo a face (e a letra) que aparece no lado oposto. Lembre-se, precisamos orientar os cubos para que as 4 letras apareçam nos 4 lados da pilha de cubos. Isto é uma relação importante entre pares de elementos do problema, e podemos utilizá-la para fazer a modelagem utilizando grafos.

Tomando o exemplo da [Figura 1.27](#), vamos iniciar a modelagem do problema formando conjuntos de vértices com as 4 letras. Considerando que, a relação importante está entre pares de letras em lados opostos dos cubos, este será nosso critério para formar arestas. Ou seja, dadas duas letras, estas formarão uma aresta se aparecerem em lados opostos de um cubo. Com isso, os 4 cubos da [Figura 1.27](#) formam os 4 grafos mostrados na [Figura 1.30](#).

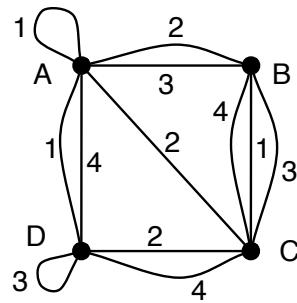
Figura 1.30 – Modelagem de um grafo para cada cubo



Fonte: o autor.

A seguir, se unirmos estes 4 grafos identificando o grafo de procedência de cada aresta com números (cubos 1 ao 4), formamos o grafo da [Figura 1.31](#), o qual engloba todas as opções de lados opostos de cubos.

Figura 1.31 – União dos grafos dos 4 cubos

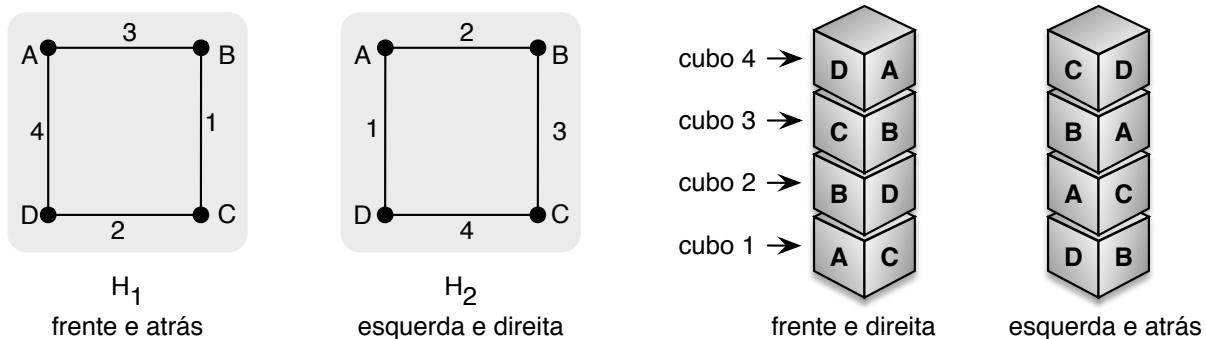


Fonte: o autor.

Visto que, um cubo tem 3 pares de faces opostas, para encontrar uma solução devemos escolher para cada cubo os dois pares de faces que ficarão nos lados da frente e de trás da pilha, bem como esquerda e direita. Para isso, é necessário extrair dois subgrafos H_1 e H_2 do grafo da [Figura 1.31](#) com as seguintes propriedades: (i) cada subgrafo deve conter apenas uma aresta de cada cubo e (ii) cada subgrafo deve ser regular de grau 2.

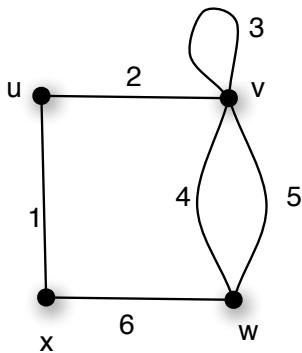
A primeira propriedade se deve ao fato de que é impossível colocar o mesmo cubo mais de uma vez na pilha, enquanto, a segunda assegura que uma letra apareça uma vez em cada um dos quatro lados da pilha. A [Figura 1.32](#) apresenta uma solução para o problema com os dois subgrafos e uma ilustração de como seria a pilha de cubos correspondente.

Figura 1.32 – Solução do problema dos 4 cubos



Fonte: o autor.

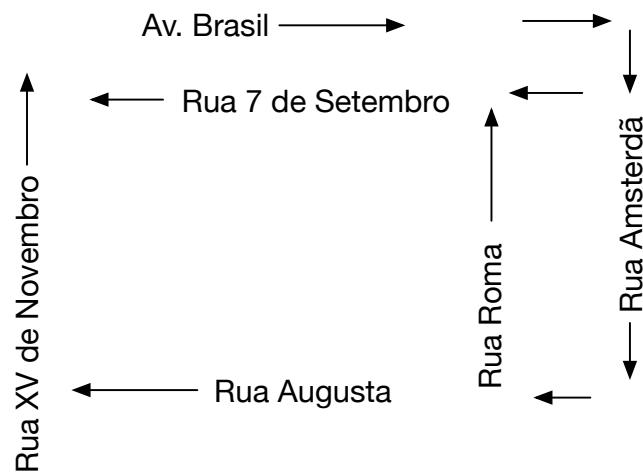
1.12 Exercícios

1. O que é um grafo? Responda com suas próprias palavras.
2. Para que serve um grafo?
3. Desenhe os grafos não-dirigidos abaixo, sendo dados seus conjuntos de vértices e arestas:
 - a) $V = \{1, 2, 3, 4, 5\}$ e $E = \{(1, 2), (1, 4), (1, 5), (2, 3), (3, 4), (4, 4)\};$
 - b) $V = \{1, 2, 3, 4, 5, 6\}$ e $E = \{(1, 2), (1, 4), (1, 4), (2, 3), (2, 5), (3, 5)\};$
 - c) $V = \{1, 2, 3, 4, 5, 6\}$ e
 $E = \{(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 5), (3, 6), (4, 5), (4, 6)\};$
 - d) $V = \{1, 2, 3, 4, 5\}$ e $E = \{(1, 2), (1, 4), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5)\};$
 - e) $V = \{1, 2, 3, 4\}$ e $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\};$
 - f) $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ e
 $E = \{(1, 2), (2, 2), (2, 3), (3, 4), (3, 5), (6, 7), (6, 8), (7, 8)\}.$
4. Dados os grafos da questão anterior, classifique-os como simples ou multigrafos.
5. Quais são as diferenças entre grafos simples e multigrafos?
6. Desenhe um exemplo de grafo simples dirigido e um não dirigido.
7. Desenhe um exemplo de multigrafo dirigido e um não dirigido.
8. Dado o grafo abaixo, quais afirmações estão corretas?
 - a) Os vértices v e w são adjacentes;
 - b) Os vértices v e x são adjacentes;
 - c) A aresta 2 é incidente ao vértice u ;
 - d) A aresta 5 é incidente ao vértice x .
9. Desenhe um grafo que represente o seguinte panorama de amizade entre quatro pessoas:
 - a) João é amigo de Carolina e de Maria, mas não de Marco;
 - b) Marco é amigo de Carolina, mas não de Maria;
 - c) Carolina é amiga de Maria.
10. Suponha que há seis pessoas em uma festa e que nenhuma delas é totalmente desconhecida do grupo. Prove que é sempre possível haver três pessoas que se

conhecem mutuamente e três pessoas das quais nenhuma delas conhece as outras duas. (Dica: represente as seis pessoas como vértices de um grafo e considere as 5 possíveis arestas emergindo de um desses vértices).

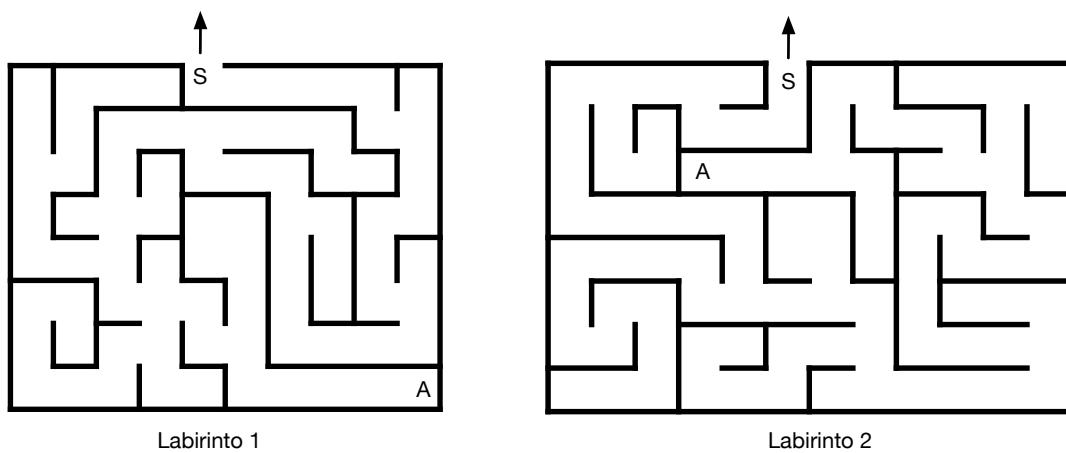
11. A [Figura 1.33](#) representa um sistema de ruas de mão única. Desenhe um grafo dirigido que sirva de modelo para este sistema.

Figura 1.33 – Exemplo sistema de ruas de mão única



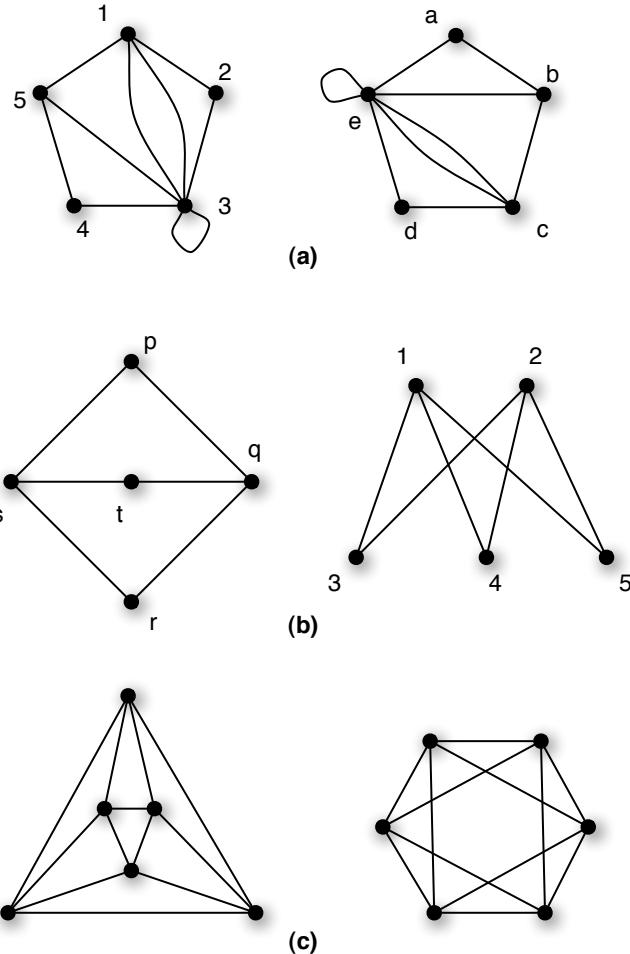
12. A [Figura 1.34](#) mostra dois labirintos. Para cada um deles, modele um grafo para representar as possibilidades de caminho e encontre um caminho que leve do local A até a saída do labirinto.

Figura 1.34 – Exercício labirintos



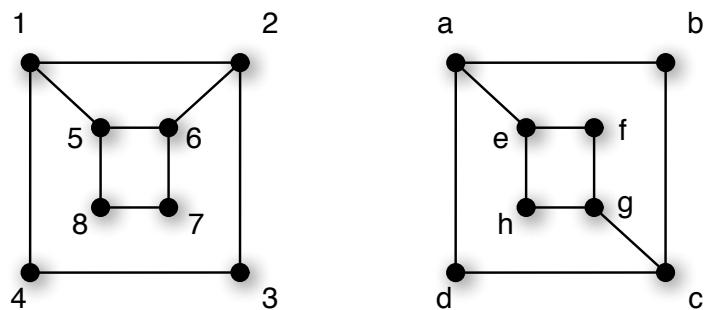
13. Mostre que os pares de grafos da Figura 1.35 são isomorfos. Faça isso por meio da demonstração de correspondência entre os vértices dos grafos.

Figura 1.35 – Exercício sobre isomorfismo



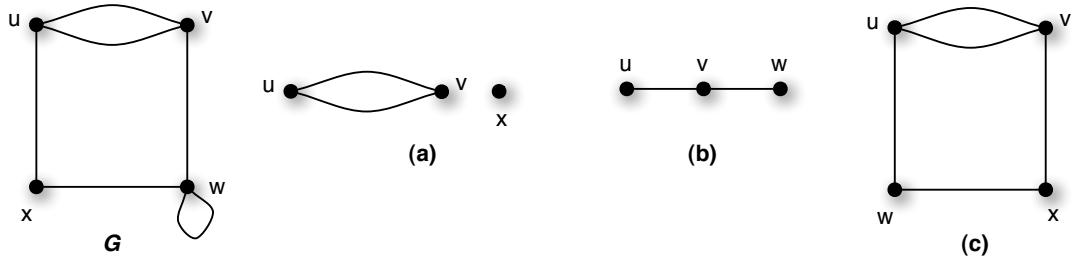
14. Os dois grafos da Figura 1.36 são isomorfos? Em caso afirmativo, encontre uma correspondência de um para um entre os vértices do primeiro com os vértices do segundo grafo. Caso contrário, explique porque tal correspondência não existe.

Figura 1.36 – Exercício isomorfismo



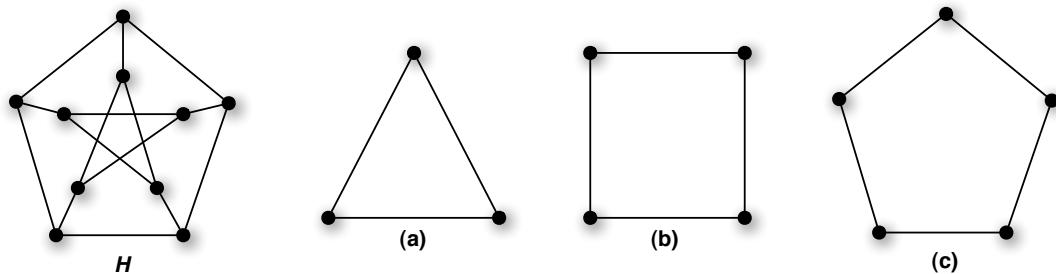
15. Na [Figura 1.37](#), quais dos grafos (a, b, c), são subgrafos do grafo G ?

Figura 1.37 – Exercício isomorfismo



16. Na [Figura 1.38](#), quais dos grafos (a, b, c), são subgrafos do grafo H ?

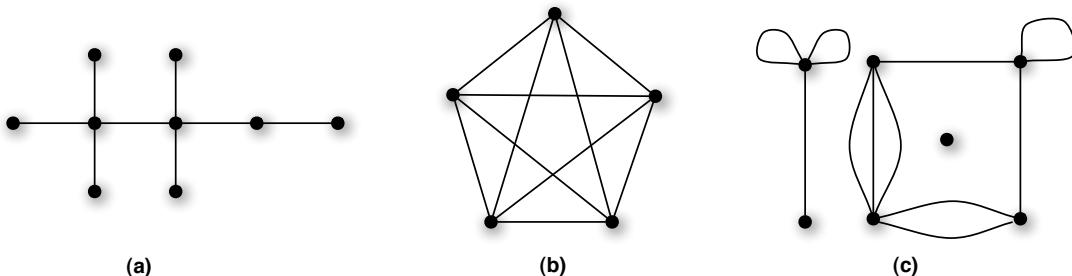
Figura 1.38 – Exercício isomorfismo



17. Desenhe um grafo simples com a seguinte sequência de graus: $(1, 1, 2, 3, 3, 4, 4, 6)$.
18. Desenhe um grafo simples, com a seguinte sequência de graus: $(3, 3, 3, 3, 3, 5, 5, 5)$.
19. Diga se cada uma das sentenças abaixo é verdadeira ou falsa. Justifique sua resposta com uma prova ou contraexemplo.
- Se dois grafos possuem a mesma sequência de graus, então eles são isomorfos.
 - Se dois grafos são isomorfos, então eles possuem a mesma sequência de graus.

20. Escreva a sequência de graus de cada um dos grafos (a, b, c) da [Figura 1.39](#).

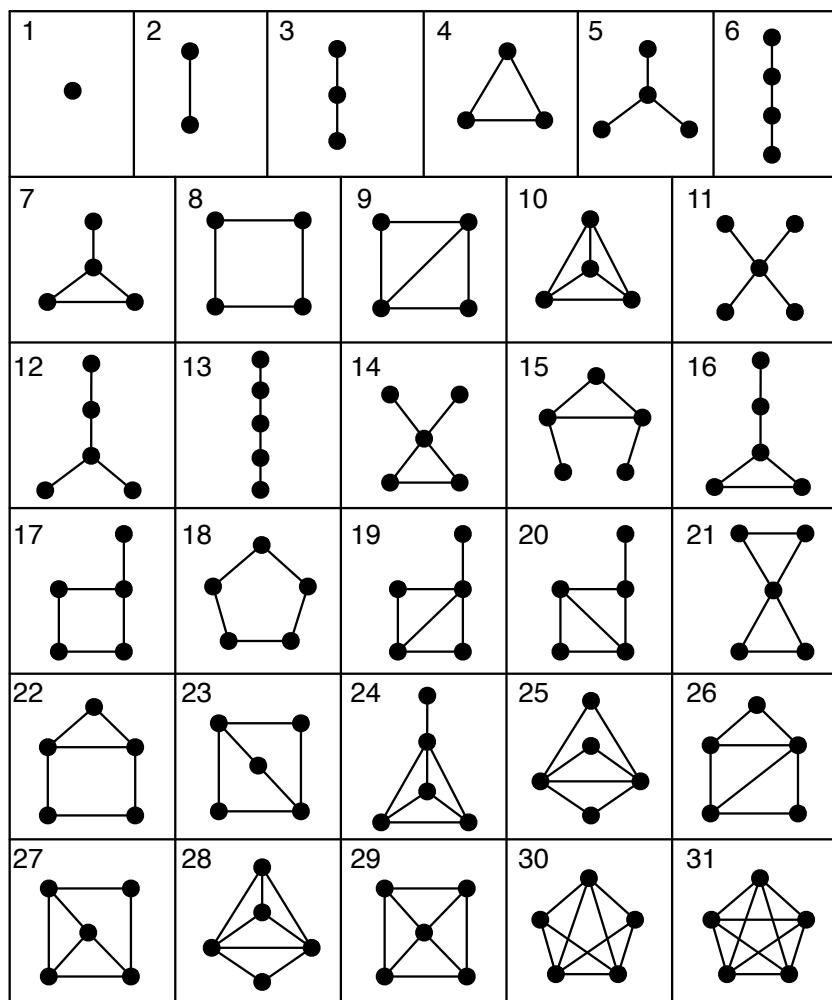
Figura 1.39 – Exercício sequência de graus



21. O que é um grafo completo?
22. O que é um grafo bipartido?

23. O que é um grafo bipartido completo?
24. Sob que circunstâncias devemos usar um grafo dirigido ao invés de um grafo não-dirigido?
25. A Figura 1.40 mostra todos os grafos conexos não rotulados com até 5 vértices. Nesta figura, localize todos os grafos regulares e todos os grafos bipartidos.

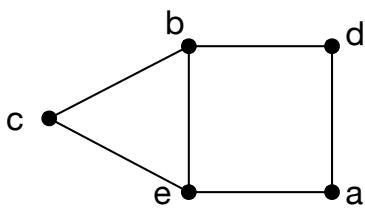
Figura 1.40 – Grafos conexos não rotulados com até 5 vértices



26. Quantas arestas tem um grafo nulo N_n ? (Obs.: n é a quantidade de vértices)
27. Quantas arestas tem um grafo ciclo C_n ?
28. Quantas arestas tem um grafo bipartido completo $K_{r,s}$?
29. Quantas arestas tem um grafo completo K_n ?
30. Qual é o grafo simples mais denso que existe? Justifique sua resposta.
31. Desenhe todos os grafos completos com até 8 vértices.

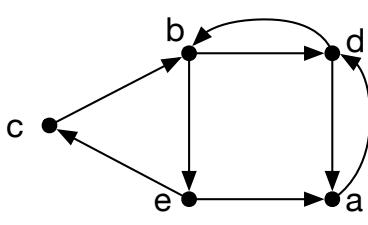
32. Desenhe todos os grafos ciclo com até 6 vértices.
33. Desenhe os seguintes grafos bipartidos completos: $K_{1,1}$, $K_{1,2}$, $K_{1,3}$, $K_{2,2}$, $K_{2,3}$, $K_{2,4}$, $K_{3,3}$, $K_{3,4}$, $K_{4,4}$.
34. Um **grafo tripartido completo** $K_{r,s,t}$ consiste em três conjuntos de vértices (de tamanhos r , s e t), com uma aresta ligando dois vértices se e somente se eles estiverem em conjuntos diferentes. Desenhe os grafos $K_{2,2,2}$ e $K_{3,3,2}$; e determine o número de arestas do grafo $K_{3,4,5}$.
35. Muitas vezes os grafos são usados na implementação de jogos por computador. Por exemplo: as 28 pedras de um jogo de dominó poderiam ser modeladas como um grafo. Como você faria essa modelagem (quais elementos do dominó são vértices? E quais são arestas?)
36. Desenhe um exemplo (se existir) de cada um dos grafos abaixo:
- um grafo bipartido que é um grafo regular de grau 5;
 - um grafo cubo com 11 vértices;
 - quatro grafos regulares de grau 4.

37. Quais das seguintes sequências de vértices representam um percurso no grafo abaixo? Quais percursos são caminhos? Quais percursos são ciclos? Quais são os comprimentos daqueles que são percursos?



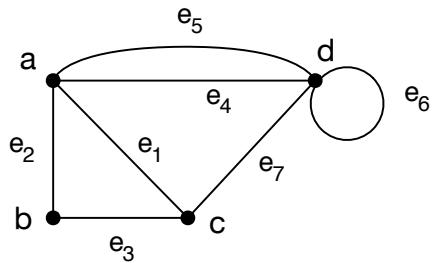
- $\langle a, e, b, c, b \rangle$
- $\langle a, e, a, d, b, c, a \rangle$
- $\langle e, b, a, d, b, e \rangle$
- $\langle c, b, d, a, e, c \rangle$

38. Quais das seguintes sequências de vértices representam um percurso no grafo abaixo? Quais percursos são caminhos? Quais percursos são ciclos? Quais são os comprimentos daqueles que são percursos?



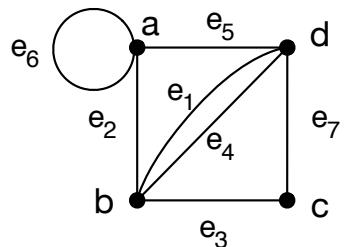
- $\langle d, b, e, c, b \rangle$
- $\langle d, a, d, a, d \rangle$
- $\langle d, a, b, e, d \rangle$
- $\langle d, b, e, c, b, a, d \rangle$

39. No grafo abaixo, encontre:



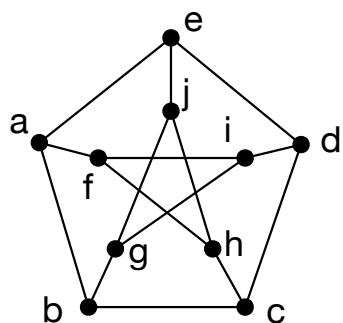
- a) um percurso fechado que não é uma trilha;
- b) uma trilha fechada que não é um ciclo; e
- c) todos os ciclos de tamanho 5 ou menos.

40. No grafo abaixo, encontre:



- a) um percurso de tamanho 7 entre os vértices b e d;
- b) um caminho de comprimento máximo.

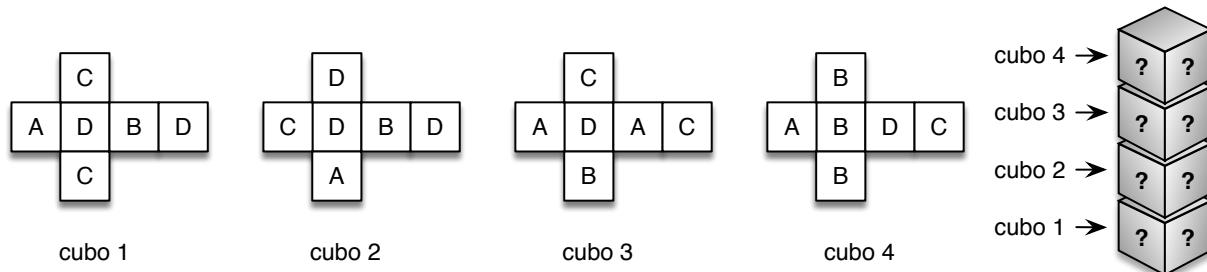
41. No grafo abaixo, encontre:



- a) uma trilha de tamanho 5;
- b) um caminho de tamanho 9;
- c) ciclos de tamanhos 5, 6, 8 e 9.

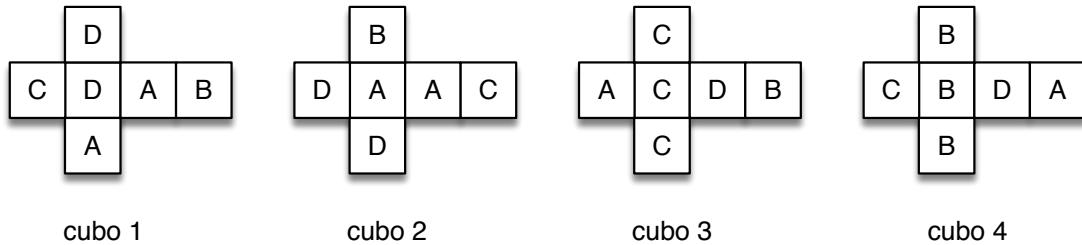
42. O jogo dos 4 cubos é um quebra-cabeças em que são dados 4 cubos com suas faces coloridas de vermelho (*R-red*), verde (*G-green*), azul (*B-blue*) e amarelo (*Y-yellow*). A solução consiste em empilhar os 4 cubos de forma a cada uma das cores aparecer em cada um dos lados da pilha. Dados os 4 cubos da Figura 1.41 (planificados), use os conceitos de Teoria dos Grafos vistos até agora para encontrar uma solução do problema.

Figura 1.41 – Exercício 4 cubos



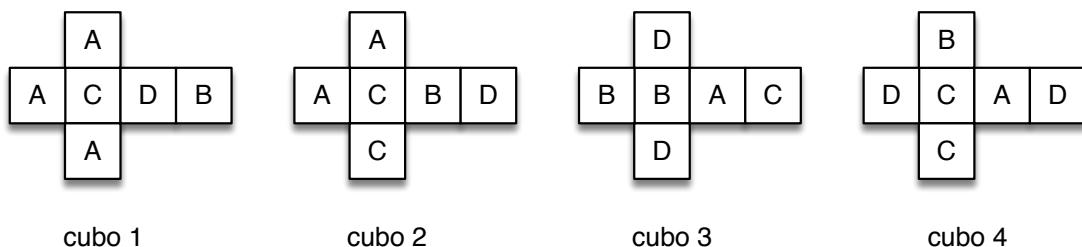
43. Mostre que não existe solução para o problema dos quatro cubos ilustrado na [Figura 1.42](#).

Figura 1.42 – Exercício 4 cubos sem solução



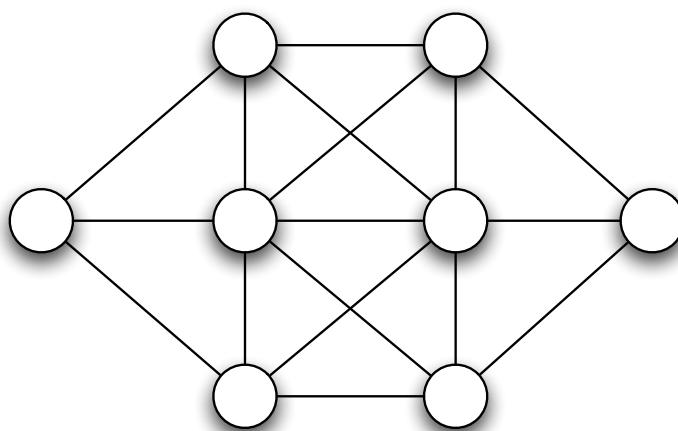
44. Prove que o problema dos quatro cubos ilustrado na [Figura 1.43](#) tem uma única solução.

Figura 1.43 – Exercício 4 cubos com solução única

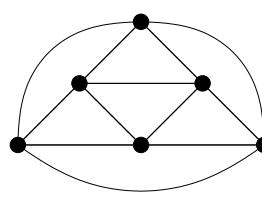


45. O problema dos oito círculos consiste em colocar as letras A, B, C, D, E, F, G e H nos oito círculos da [Figura 1.44](#) de tal forma que nenhuma letra pode ficar adjacente a outra letra se elas forem vizinhas na ordenação alfabética. Tente encontrar uma solução para o problema escrevendo as letras no grafo abaixo.

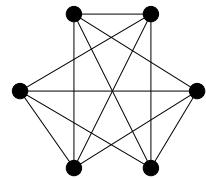
Figura 1.44 – Problema dos oito círculos



46. (PosComp – 2005) Os grafos $G = (V_G, E_G)$ e $H = (V_H, E_H)$ são isomorfos. Assinale a alternativa que justifica esta afirmação.

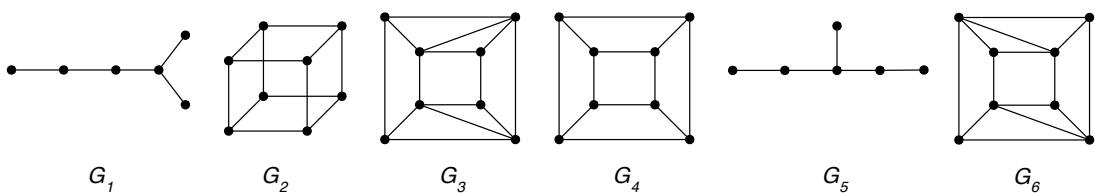


G



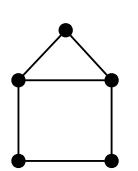
H

- a) As sequências de grau dos vértices G e H são iguais.
 - b) Os grafos têm o mesmo número de vértices e o mesmo número de arestas.
 - c) Existe uma bijeção de V_G em V_H que preserva adjacências.
 - d) Cada vértice de G e de H pertence a exatamente quatro triângulos distintos.
 - e) Ambos os grafos admitem um circuito que passa por cada aresta exatamente uma vez.
47. (PosComp – 2007) Considere os seis grafos G_1 , G_2 , G_3 , G_4 , G_5 e G_6 mostrados a seguir.

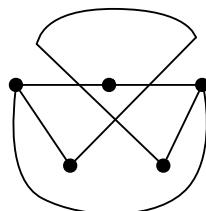


Pode-se afirmar que os únicos pares de grafos isomorfos entre si são:

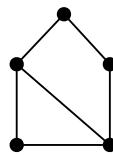
- a) G_1 e G_5 ; G_3 e G_6
 - b) G_3 e G_4 ; G_2 e G_6
 - c) G_1 e G_5
 - d) G_2 e G_4
 - e) G_3 e G_6
48. (PosComp – 2008) Considere os grafos I, II, III, IV e V, mostrados abaixo:



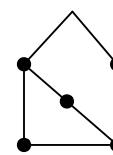
I



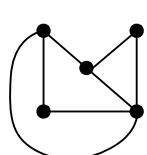
II



III



IV



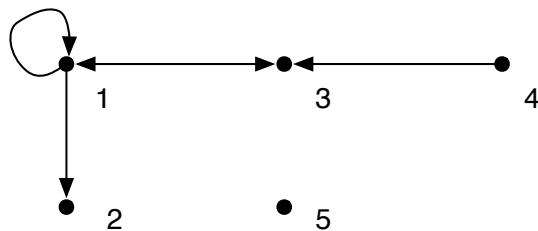
V

São grafos isomorfos:

- a) todos acima apresentados.
 - b) apenas I e III.
 - c) apenas II e V.
 - d) apenas III e IV.
 - e) apenas I, II e III.
49. (PosComp – 2010) Dados dois grafos não orientados $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$:
- $$G_1 : V_1 = \{a, b, c\}, E_1 = \{(a, b), (b, c), (a, c)\}$$
- $$G_2 : V_2 = \{d, e\}, E_2 = \{(d, e)\}$$
- Qual alternativa apresenta corretamente o grafo $G_r = (V, E)$ resultante da soma dos grafos G_1 e G_2 ?

- a) $G_r : V = \{a, b, c, d, e\}, E = \{(a, b), (b, c), (a, c), (d, e)\}$
- b) $G_r : V = \{a, b, c, d, e\}, E = \{(a, d), (a, e), (b, d), (b, e), (c, d), (c, e), (d, e)\}$
- c) $G_r : V = \{a, b, c, d, e\},$
 $E = \{(a, b), (b, c), (a, c), (a, d), (a, e), (b, d), (b, e), (c, d), (c, e)\}$
- d) $G_r : V = \{a, b, c, d, e\},$
 $E = \{(a, b), (b, c), (a, c), (a, d), (a, e), (b, d), (b, e), (c, d), (c, e), (d, e)\}$
- e) $G_r : V = \{a, b, c, d, e\}, E = \{(a, b), (b, c), (c, d), (d, e), (e, a)\}$

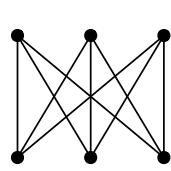
50. (PosComp – 2011) Considere o grafo a seguir:



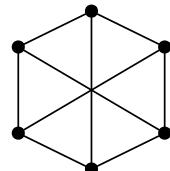
O grafo representa a relação:

- a) $R = \{(1, 1), (1, 2), (1, 3), (3, 1), (4, 3)\}$
- b) $R = \{(1, 1), (1, 2), (1, 3), (3, 1), (3, 4)\}$
- c) $R = \{(1, 1), (1, 3), (2, 1), (3, 1), (3, 4)\}$
- d) $R = \{(1, 1), (1, 2), (1, 3), (3, 4), (4, 3)\}$
- e) $R = \{(1, 1), (1, 3), (2, 1), (3, 1), (4, 3)\}$

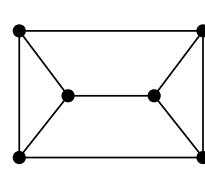
51. (PosComp – 2015) Considere os grafos a seguir:



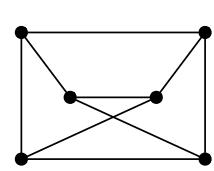
G_1



G_2



G_3



G_4

Pela análise desses grafos, verifica-se que:

- a) G_3 e G_4 são grafos completos.
- b) G_1 e G_2 são grafos isomorfos.
- c) G_3 e G_1 são grafos bipartidos.
- d) G_2 e G_3 são grafos planares.
- e) G_4 e G_1 são multigrafos.

2 Representação de grafos

Agora que temos um conhecimento geral dos principais conceitos e definições sobre grafos, partiremos para o desenvolvimento de estruturas de dados para implementação computacional de grafos. O tema é bastante amplo e detalhes de implementação podem variar dependendo do tipo de grafo ou do contexto de aplicação. Entretanto, longe de esgotar a discussão sobre o assunto, este capítulo pretende mostrar questões gerais básicas que podem ser úteis para a maior parte das implementações computacionais para solucionar problemas envolvendo grafos.

Começamos com uma contextualização básica e enumeramos as operações mais comuns que uma estrutura de dados de grafos deveria fornecer. São mostrados quatro tipos básicos de estruturas de dados: listas de arestas, listas de adjacência, mapas de adjacência e matrizes de adjacência. Por fim, propõe-se alguns exercícios de implementação.

2.1 Introdução

Até o presente momento, foram apresentadas duas formas de representação de grafos: a representação por definição dos conjuntos de vértices e arestas (conjuntos V e E) e a representação gráfica. Estas duas formas de representação são úteis para o estudo teórico de assuntos relacionados a grafos, mas são inadequadas para a descrição e implementação de algoritmos. Para tal, veremos neste capítulo outras quatro formas de representação em termos de estruturas de dados: as listas de arestas, as listas de adjacência, os mapas de adjacência e as matrizes de adjacência.

Para iniciarmos a discussão sobre implementação de tais estruturas, precisamos estabelecer as principais entidades envolvidas. Temos então vértices, arestas e grafos. Considerando o uso de linguagens orientadas a objetos, tais entidades poderiam ser implementadas como classes, por exemplo.

Em todas as quatro estruturas devemos manter uma coleção de vértices, os quais são objetos que contém referências para dados relevantes do problema modelado (por exemplo, os identificadores dos vértices). Por outro lado, as estruturas diferem bastante na forma com a qual tratam as arestas. Consequentemente, o desempenho das diferentes operações sobre a estrutura de dados também varia de acordo com o tipo de estrutura.

A implementação de algoritmos em grafos envolve uma série de operações sobre a estrutura de dados. As operações incluem consultas, criação e remoção de elementos dos grafos. Algumas das operações mais comuns são: obter a ordem e o tamanho do grafo, retornar uma coleção de todos os vértices ou de todas as arestas do grafo, retornar uma

coleção de todas as arestas incidentes a um vértice em grafo não dirigido, retornar uma coleção de arestas de saída ou de chegada em um vértice de grafo dirigido, retornar o grau de um vértice (ou grau de entrada e saída em grafos dirigidos), inserir e remover vértices e arestas, entre outras.

Antes de vermos em maiores detalhes cada uma dessas estruturas, vamos definir as principais operações que elas precisam implementar.

2.2 Operações em estruturas de dados de Grafos

Sendo os grafos definidos como conjuntos de vértices e arestas, é natural que implementações computacionais modelem esta abstração como três tipos de dados fundamentais: Vértices, Arestas e Grafos (e/ou Digrafos a depender de decisões de projeto). Vértices são os tipos de dados mais elementares, geralmente contendo pelo menos referência para informações adicionais vinculadas e dependentes do contexto de aplicação. O vértice, por exemplo, pode estar associado a um identificador ou um determinado código referente aos dados do problema modelado etc. Uma aresta também pode estar associada a um objeto contendo informações oriundas do problema modelado, tais como: número de voo, distância, custo, códigos identificadores diversos, entre outras.

A seguir, definimos uma série de operações comuns em algoritmos que operam em grafos. Dependendo do contexto pode-se implementar todas elas para que se tenha uma estrutura dados abstrata para grafos, ou então apenas parte delas de forma a implementar algum algoritmo específico. O [Quadro 1](#) apresenta as operações e suas descrições. É importante frisar que não estão especificados em detalhes os tipos de dados de parâmetros de entrada e valores de retorno, podendo estes serem definidos em função de requisitos específicos de cada aplicação.

2.3 Listas de arestas

Esta estrutura de dados é, provavelmente, a mais simples de implementar, por outro lado é a menos eficiente. Sua principal característica do ponto de vista estrutural é manter uma lista não ordenada de arestas. Trata-se de uma estrutura enxuta, porém a operação de localização de uma aresta em particular não é eficiente. A localização de todas as arestas incidentes a um vértice também é ineficiente.

A [Figura 2.1](#) mostra esquematicamente a organização dos objetos vértices e arestas na memória. Temos uma coleção iterável não-ordenada V contendo referências para os vértices e outra coleção iterável não-ordenada E contendo referências para os objetos aresta. As referências estão representadas como setas na figura. Estas coleções podem ser listas duplamente encadeadas. Cada aresta tem referências para os vértices aos quais

Quadro 1 – Operações mais comuns em estruturas de dados de Grafos

Operação	Descrição
getOrdem()	retorna quantidade de vértices
getTamanho()	retorna quantidade de arestas
vertices()	retorna uma iteração de todos os vértices do grafo
arestas()	retorna uma iteração de todas as arestas do grafo
insereV()	instancia um novo vértice e o adiciona ao grafo
removeV(v)	remove o vértice v e todas as suas arestas incidentes
insereA(u, v)	instancia uma nova aresta incidente aos vértices u e v , e a adiciona ao grafo
removeA(e)	remove a aresta e
adj(v)	retorna uma iteração com todos os vértices adjacentes ao vértice u
getA(u, v)	retorna uma referência para a aresta de u para v ou null se os vértices não forem adjacentes. Para grafos não dirigidos, <code>getA(u, v)</code> e <code>getA(v, u)</code> produzem o mesmo resultado
grauE(v)	retorna o grau de entrada do vértice v em grafos dirigidos
grauS(v)	retorna o grau de saída do vértice v em grafos dirigidos
grau(v)	retorna o grau do vértice v em grafos não dirigidos. Alternativamente, pode-se usar os dois métodos anteriores em grafos não dirigidos de forma que <code>grauE(v)</code> e <code>grauS(v)</code> retornem o mesmo resultado
verticesA(e)	retorna o par de vértices que conectados à aresta e . Se o grafo for dirigido, o primeiro vértice do par é a origem e o segundo é o destino da aresta
oposto(v,e)	para um vértice v incidente à aresta e , retorna o outro vértice incidente à aresta
arestasE(v)	retorna uma iteração de todas as arestas de entrada do vértice v
arestasS(v)	retorna uma iteração de todas as arestas de saída do vértice v

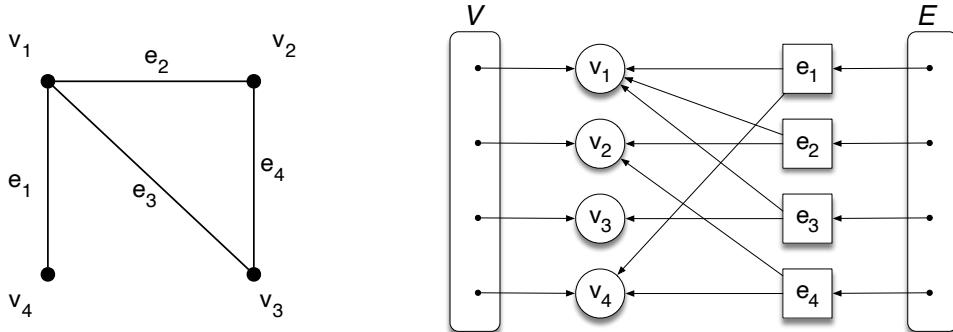
Fonte: Autor.

ela é incidente. Isto permite que as operações `verticesA(e)` e `oposto(v,e)` sejam executadas em tempo constante. Note que os vértices não dispõem de referências para suas arestas incidentes.

Considerando n e m , respectivamente, a ordem e o tamanho do grafo, a estrutura de listas de arestas consome espaço de memória $O(m + n)$, pois cada vértice ou aresta consome espaço $O(1)$ e as listas V e E usam espaço linearmente proporcional às suas quantidades de elementos. Tendo em vista que as operações `vertices()` e `arestas()` iteram sobre as listas V e E , elas têm desempenho $O(n)$ e $O(m)$, respectivamente.

A maior desvantagem desta estrutura decorre do fato de não termos referências das arestas nos objetos vértice. Com isso, as operações `getA(u, v)`, `grau(v)`, `grauE(v)`, `grauS(v)`, `arestasE(v)` e `arestasS(v)` têm desempenho $O(m)$, pois demandam uma inspeção exaustiva de

Figura 2.1 – Representação por lista de arestas de um grafo simples



Fonte: adaptado de [Goodrich, Tamassia e Goldwasser \(2013\)](#).

todas as arestas da lista E .

Quanto às operações de alteração do grafo, a adição de novos vértices e arestas leva tempo $O(1)$. A remoção de uma aresta pode ser realizada em $O(m)$ devido à necessidade de se percorrer a lista E para fazer a remoção do elemento correspondente. A operação $\text{removeV}(v)$ também é $O(m)$ porque a remoção de um vértice implica na remoção de todas as suas arestas incidentes.

2.4 Listas de adjacência

A estrutura de listas de adjacência de um grafo $G = (V, E)$ consiste em uma coleção não-ordenada iterável V de vértices. Cada vértice, por sua vez, deve manter uma lista não ordenada de todas as suas arestas incidentes. Isto permite que a determinação das arestas incidentes a um vértice seja mais eficiente. Dado um vértice v , sua lista de incidência é denotada por $I(v)$.

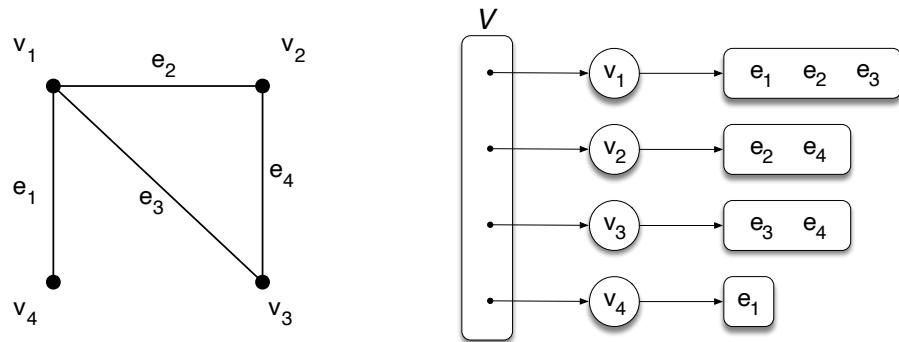
Cada lista $I(v)$ é composta por referências as arestas incidentes a v . As listas $I(v)$ podem ser compostas por coleções dinâmicas de arestas, normalmente listas duplamente encadeadas. No caso de grafos dirigidos, cada vértice mantém listas separadas para as suas arestas de entrada e de saída, respectivamente $I_e(v)$ e $I_s(v)$.

O acesso a elementos da estrutura inicialmente depende da implementação da coleção V . Em situações em que não há inserção e remoção de vértices após a criação do grafo, a coleção pode ser um array e os vértices podem possuir um identificador numérico indicando a sua posição no array. Desta forma, o acesso à lista de incidência de um determinado vértice é feito em tempo constante $O(1)$. Por outro lado, situações de natureza mais dinâmica com eventuais inserções e remoções de vértices, demandam uma implementação dinâmica para V , possivelmente por meio de listas duplamente encadeadas.

Neste caso, o acesso a determinada lista de incidência é feito em tempo $O(n)$.

A Figura 2.2 mostra a representação esquemática da estrutura de listas de adjacência para um grafo simples não dirigido. Assumimos que, cada aresta representada no diagrama é uma instância única de um objeto aresta e que todas as arestas estão organizadas em uma coleção não-ordenada iterável de arestas E .

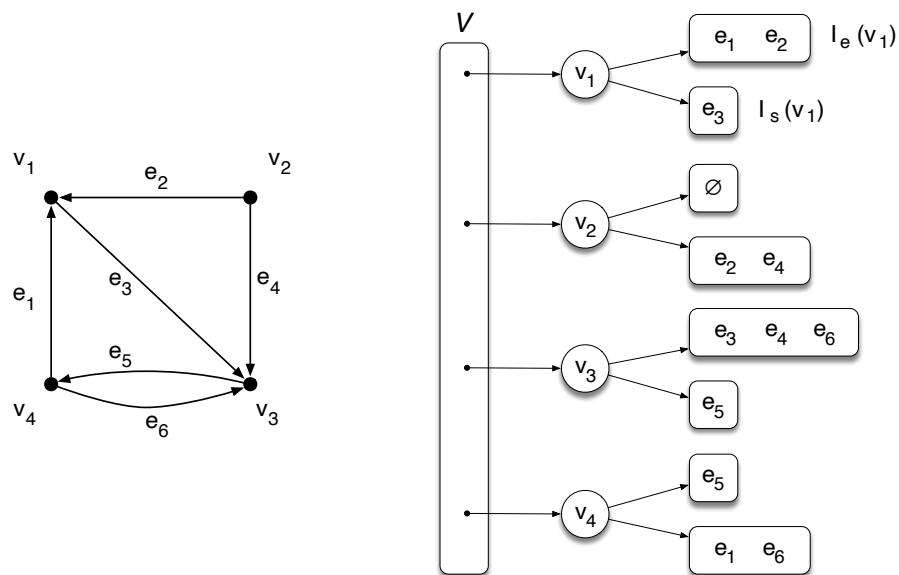
Figura 2.2 – Listas de adjacência de um grafo simples



Fonte: adaptado de Goodrich, Tamassia e Goldwasser (2013).

Conforme mencionado anteriormente, no caso de grafos dirigidos, cada vértice possui duas listas de incidência, sendo uma para as arestas de entrada e outra para as arestas de saída. A Figura 2.3 ilustra, esquematicamente, um exemplo deste tipo.

Figura 2.3 – Listas de adjacência de um grafo dirigido



Fonte: adaptado de Goodrich, Tamassia e Goldwasser (2013).

2.5 Mapas de adjacência

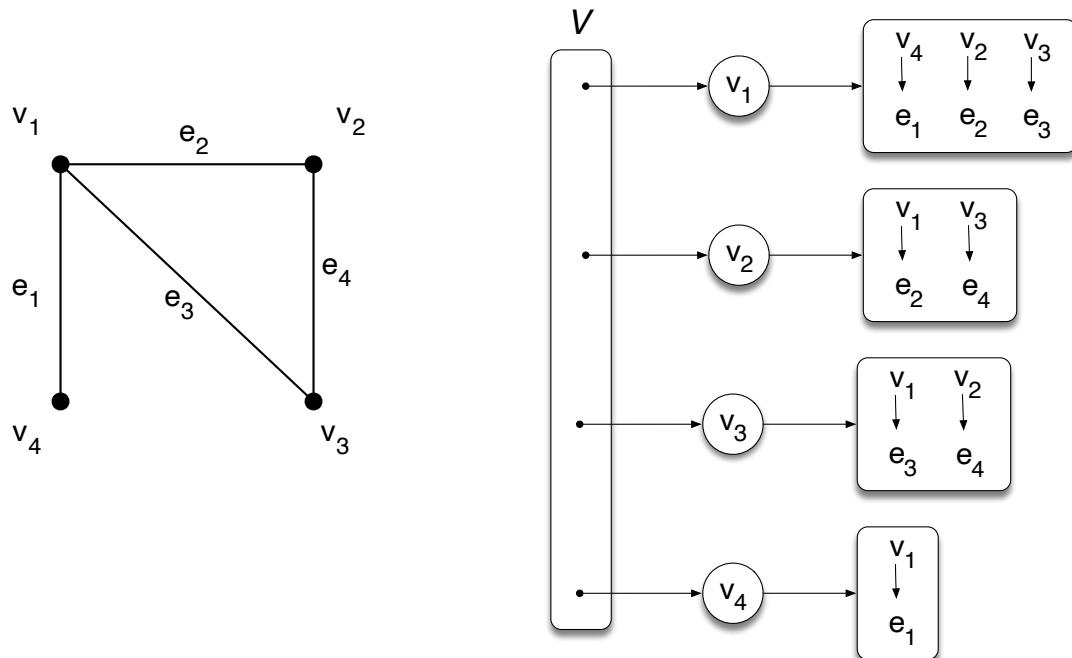
A estrutura de mapas de adjacência é semelhante à estrutura de listas de adjacência, porém ao invés de uma lista cada vértice mantém um mapa de arestas. As chaves do mapa são os vértices adjacentes ao vértice considerado. Esta estrutura permite uma maior eficiência ao acesso à uma determinada aresta feito pela operação $\text{getA}(u, v)$, desde que a implementação do mapa seja eficiente.

Ao usar mapas (ou *hash maps*) para implementar $I(v)$ de cada vértice v , fazemos o vértice incidente oposto de cada aresta ser a chave do par chave/valor do mapa. O uso de memória desta estrutura de mapas de adjacência continua sendo $O(m + n)$, visto que assim como nas listas de adjacência, cada $I(v)$ usa espaço $O(\text{grau}(v))$.

A maior vantagem dos mapas de adjacência sobre as listas de adjacência é que a operação $\text{getA}(u, v)$ compreende um desempenho esperado da ordem $O(1)$. Tratando-se, portanto, de uma boa estrutura de dados para implementação de grafos de uma forma geral.

A [Figura 2.4](#) mostra a representação esquemática da estrutura de mapas de adjacência para um grafo simples não dirigido.

Figura 2.4 – Mapas de adjacência de um grafo dirigido



Fonte: adaptado de [Goodrich, Tamassia e Goldwasser \(2013\)](#).

2.6 Matrizes de adjacência

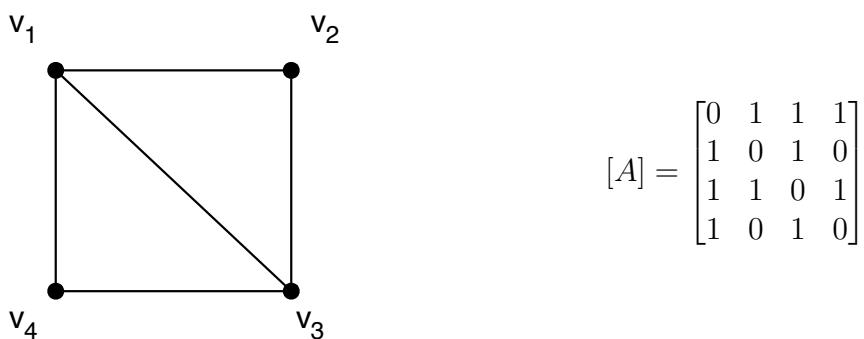
Uma matriz de adjacência, denotada por $[A]$, é uma matriz quadrada de tamanho $n \times n$. Para um grafo simples $G = (V, E)$, os elementos da matriz são definidos da seguinte forma:

$$a_{i,j} = \begin{cases} 1, & \text{se } (v_i, v_j) \in E; \\ 0, & \text{se } (v_i, v_j) \notin E. \end{cases} \quad (2.1)$$

Por esta definição, pensamos nos vértices como inteiros no conjunto $\{0, 1, 2, \dots, n-1\}$ e nas arestas como pares destes inteiros. Dado um par de vértices, podemos acessar uma aresta em tempo constante. Armazenando estas informações em uma matriz quadrada de tamanho $n \times n$, cada vértice do grafo fica associado a uma linha e uma coluna da matriz. Se houver aresta ligando dois vértices v_i e v_j quaisquer, os elementos correspondentes às linhas i e j da matriz possuem valor igual a 1. Os grafos da [Figura 2.5](#) e da [Figura 2.6](#) ilustram este caso.

Note que no caso de grafos não dirigidos a matriz é simétrica, guardando portanto a propriedade $[A] = [A]^T$, em consequência de $a_{i,j} = a_{j,i}$. Isto está de acordo com a definição de grafos não dirigidos vista no [Capítulo 1](#), pois este tipo de grafo modela relações simétricas entre elementos de um problema. Para o caso de digrafos, sempre temos $[A] \neq [A]^T$.

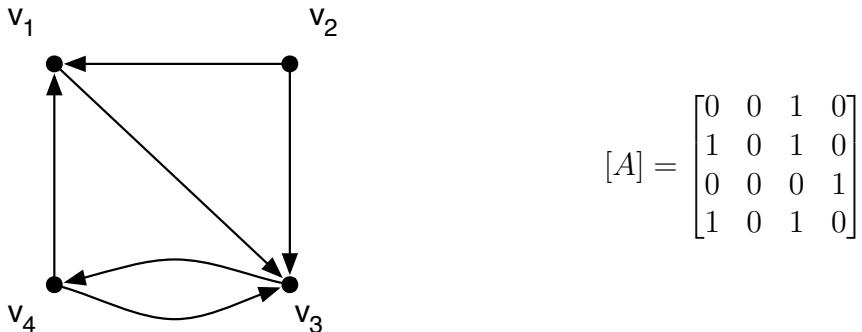
Figura 2.5 – Matriz de adjacência de grafo simples



Fonte: o autor.

É importante mencionar que estas matrizes apenas indicam a existência ou não de uma aresta conectando determinado par de vértices. Se optarmos por acessar os objetos aresta, podemos construir a matriz de forma que cada elemento $a_{i,j}$ guarde a referência para um objeto aresta (v_i, v_j) , ao invés de conter um número inteiro. Se não houver aresta entre determinado par de vértices, o elemento correspondente da matriz deve conter uma referência nula.

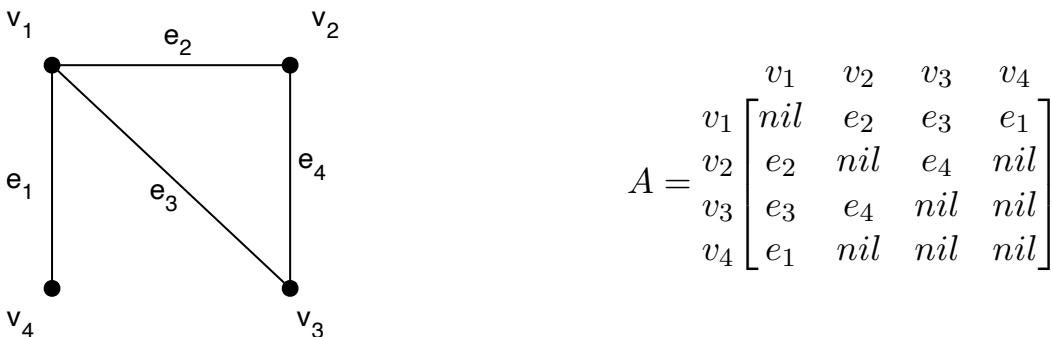
Figura 2.6 – Matriz de adjacência de grafo simples dirigido



Fonte: o autor.

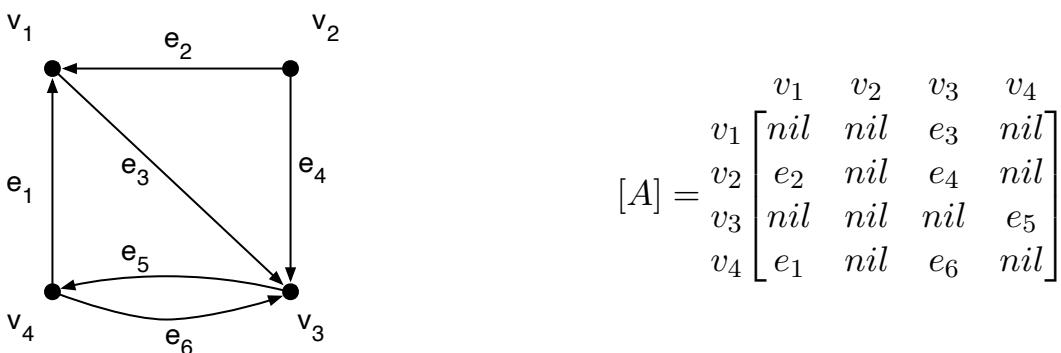
A [Figura 2.7](#) mostra, esquematicamente, como ficaria este tipo de matriz de adjacência para um grafo simples não dirigido.

Figura 2.7 – Matriz de adjacência de grafo simples



No caso de um grafo simples dirigido, teríamos matrizes não-simétricas com referências para as arestas, conforme ilustrado na [Figura 2.8](#).

Figura 2.8 – Matriz de adjacência de grafo simples dirigido



Fonte: o autor.

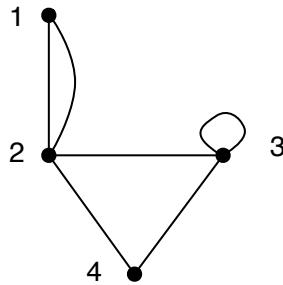
Os exemplos acima tratam apenas de grafos simples. Como cada par de vértice forma apenas uma aresta, é conveniente a utilização de matrizes. Para multigrafos, especialmente no caso de arestas paralelas, é preciso adaptar a estrutura para comportar mais de uma

referência a arestas para cada par de vértice. É possível pensar em estratégias combinando listas ou mapas de adjacência com matrizes, em que cada elemento da matriz estaria associado a uma lista ou mapa de arestas. Isto resolve o problema de representação de arestas paralelas, mas por outro lado perdemos uma das grandes vantagens de se usar matrizes de adjacência, que é o acesso a arestas em tempo constante.

2.7 Exercícios

1. Quais são as principais diferenças, em termos de vantagens e desvantagens, entre o uso de listas de adjacência e matrizes de adjacência para representação de grafos?
2. Escreva as listas e matrizes de adjacência dos grafos abaixo, sendo dados seus conjuntos de vértices e arestas. Faça a mesma coisa, considerando que os conjuntos abaixo definem grafos dirigidos.
 - a) $V = \{1, 2, 3, 4, 5\}$ e $E = \{(1, 2), (1, 4), (1, 5), (2, 3), (3, 4), (4, 4)\};$
 - b) $V = \{1, 2, 3, 4, 5, 6\}$ e $E = \{(1, 2), (1, 4), (1, 4), (2, 3), (2, 5), (3, 5)\};$
 - c) $V = \{1, 2, 3, 4, 5, 6\}$ e
$$E = \{(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 5), (3, 6), (4, 5), (4, 6)\};$$
 - d) $V = \{1, 2, 3, 4, 5\}$ e $E = \{(1, 2), (1, 4), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5)\};$
 - e) $V = \{1, 2, 3, 4\}$ e $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\};$
 - f) $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ e
$$E = \{(1, 2), (2, 2), (2, 3), (3, 4), (3, 5), (6, 7), (6, 8), (7, 8)\}.$$
3. (PosComp – 2006) A respeito da representação de um grafo de n vértices e m arestas é correto dizer que:
 - a) a representação sob a forma de matriz de adjacência exige espaço $\Omega(m^2)$.
 - b) a representação sob a forma de listas de adjacência permite verificar a existência de uma aresta ligando dois vértices dados em tempo $O(1)$.
 - c) a representação sob a forma de matriz de adjacência não permite verificar a existência de uma aresta ligando dois vértices dados em tempo $O(1)$.
 - d) a representação sob a forma de listas de adjacência exige espaço $\Omega(n + m)$.
 - e) todas as alternativas estão corretas.

4. (PosComp – 2013) Seja o grafo G a seguir:



Com base nesse grafo, considere as afirmativas a seguir.

I. O grafo G é conexo.

II. A matriz de adjacências do grafo G é dada por $[A] = \begin{bmatrix} 0 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$

III. O grau do vértice 2 é igual a 2.

IV. O grafo G é denominado como Grafo Simples.

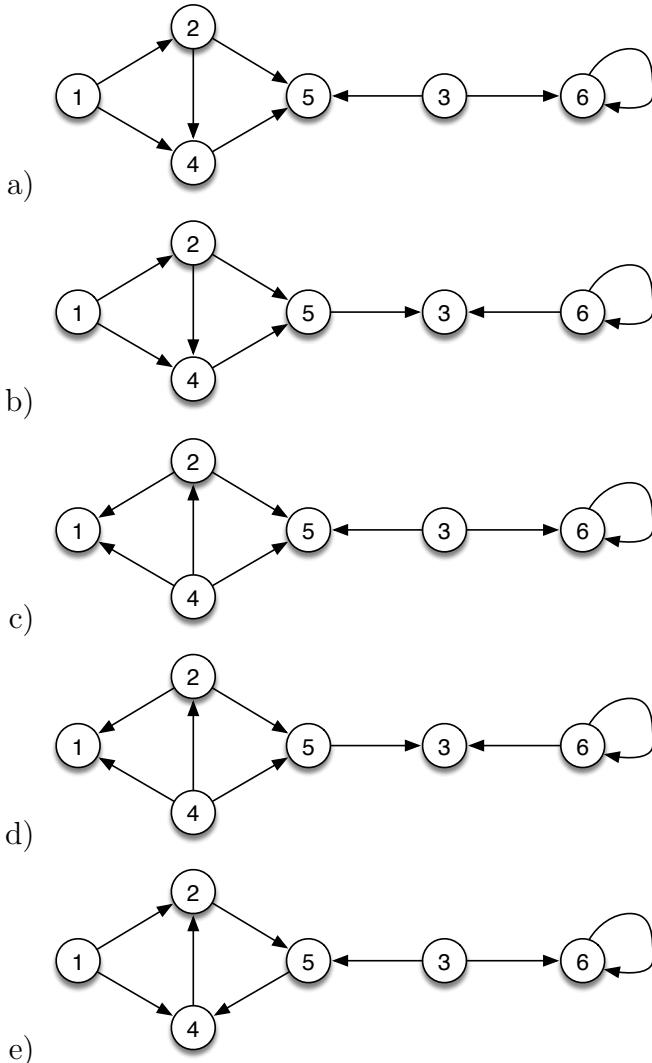
Assinale a alternativa correta.

- a) Somente as afirmativas I e II são corretas.
- b) Somente as afirmativas I e IV são corretas.
- c) Somente as afirmativas III e IV são corretas.
- d) Somente as afirmativas I, II e III são corretas.
- e) Somente as afirmativas II, III e IV são corretas.

5. (PosComp – 2015) Seja $G = (V, E)$ um grafo em que V é o conjunto de vértices e E é o conjunto de arestas. Considere a representação de G como uma matriz de adjacências $[A]$.

$$[A] = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \\ v_1 & 0 & 1 & 0 & 1 & 0 & 0 \\ v_2 & 0 & 0 & 0 & 0 & 1 & 0 \\ v_3 & 0 & 0 & 0 & 0 & 1 & 1 \\ v_4 & 0 & 1 & 0 & 0 & 0 & 0 \\ v_5 & 0 & 0 & 0 & 1 & 0 & 0 \\ v_6 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

O correspondente grafo orientado G é:



6. (PosComp – 2016) A matriz de um grafo $G = (V, A)$ contendo n vértices é uma matriz $n \times n$ de bits, em que $A[i, j]$ é 1 (ou verdadeiro, no caso de booleanos) se e somente se existir um arco do vértice i para o vértice j . Essa definição é uma:

- a) Matriz de adjacência para grafos não ponderados.
- b) Matriz de recorrência para grafos não ponderados.
- c) Matriz de incidência para grafos não ponderados.
- d) Matriz de adjacência para grafos ponderados.
- e) Matriz de incidência para grafos ponderados.

7. Implemente em uma linguagem de programação, uma estrutura de dados, ou conjunto de classes para armazenar e manipular grafos e digrafos. Avalie as vantagens e desvantagens da sua estrutura em relação a modelagem comentada na [seção 2.2](#).

3 Busca básica em grafos

O percorrimento dos vértices e arestas respeitando a estrutura dos grafos é chamada de “busca” e constitui uma operação básica em grafos dirigidos e não dirigidos. Este capítulo apresenta os dois métodos diferentes para busca: em largura e em profundidade. São mostrados exemplos de pseudocódigo que podem ser facilmente implementados em uma linguagem de programação. Ao final do capítulo, dois exemplos de aplicação direta de busca são apresentados e discutidos.

3.1 Introdução

Estruturas de dados frequentemente precisam ser percorridas para que diversas operações sejam efetuadas, tais como: inserir ou remover novos elementos ou simplesmente fazer leitura de dados armazenados na estrutura. Estruturas mais simples geralmente também são manipuladas com maior facilidade. Por exemplo, para atribuirmos um valor a uma variável simples, basta atribuí-lo diretamente utilizando o identificador da variável, conforme apresentado no exemplo abaixo.

```
int a;
a = 10;
```

No caso de vetores, seus elementos são acessados por meio de índices numéricos. Isto é possível, visto que os elementos de um vetor são dispostos em um bloco contínuo de memória. Assim, o identificador do vetor é uma referência para o endereço de memória do início deste bloco. Os demais elementos são acessados por meio da indexação numérica do identificador. Por exemplo, se quisermos percorrer um vetor de inteiros com 10 elementos, para atribuir valor nulo a todos eles, podemos simplesmente construir um laço de 10 repetições utilizando o contador do laço como índice para os elementos do vetor.

```
int v[10]; // declaração do vetor
int i;      // contador
for (i = 0; i < 10; i++)
    v[i] = 0;
```

Para estruturas alocadas dinamicamente, tais como listas encadeadas, o processo se torna mais complicado, pois cada elemento é armazenado em um endereço de memória que não necessariamente está em uma posição contígua à do elemento anterior. Por esse motivo,

para se chegar a um determinado elemento de uma lista, deve ser feito o percorrimento da lista passando por todos os seus elementos desde o elemento inicial, até o elemento desejado.

No caso dos grafos, o problema de manipulação da estrutura de dados é ainda mais complexo, pois normalmente queremos percorrer a estrutura respeitando a sua topologia. Os algoritmos de busca constituem métodos para resolver este problema, promovendo a pesquisa em grafos. **Pesquisar** um grafo significa visitar seus vértices passando sistematicamente por suas arestas, ou seja, não é apenas listar os vértices de um grafo em determinada ordem, mas sim atingir cada vértice obedecendo a estrutura do grafo, definida pelas arestas. Existem basicamente dois métodos de busca: a busca em largura e a busca em profundidade.

O fato dos métodos de busca servirem para explorar a estrutura de um grafo nos traz algumas consequências importantes. Em primeiro lugar, os algoritmos de busca servem de base para construção de outros algoritmos mais especializados, pois a solução de vários problemas envolvendo grafos requer que seus vértices sejam visitados. Exemplos destes algoritmos são o algoritmo de Dijkstra, para cálculo de caminhos de custo mínimo, o algoritmo de Prim, para determinação da árvore geradora de custo mínimo e o algoritmo de componentes fortemente conexas, entre outros.

Como aplicação direta dos algoritmos de busca para solução de problemas, temos a busca em grafos de estados, na área de Inteligência Artificial e a ordenação topológica de grafos acíclicos dirigidos. Estes exemplos serão apresentados e discutidos ao final deste capítulo.

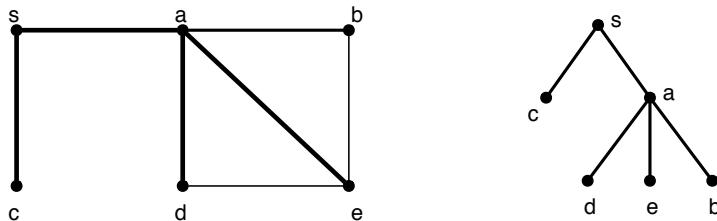
3.2 Busca em largura

O método de busca em largura explora o grafo a partir de um vértice inicial arbitrário. A partir daí, a busca vai se alastrando uniformemente pelo grafo. Em cada grande iteração do algoritmo, são visitados todos os vértices a uma mesma distância (em quantidade de arestas) do vértice inicial. Assim, se imaginássemos um grande grafo desenhado no chão e derramássemos um balde de tinta sobre o vértice inicial, a tinta se espalharia uniformemente a partir desse ponto, da mesma forma que a busca em largura o faz.

A busca em largura tem uma propriedade interessante: para atingir cada vértice do grafo, a busca percorre os caminhos mais curtos desde o vértice inicial, medidos em quantidade de arestas percorridas. O conjunto de caminhos percorridos forma uma estrutura de árvore, cuja raiz é o vértice inicial. Esta árvore é comumente chamada de **árvore de busca em largura**. Desta forma, é interessante que as implementações computacionais de busca em largura armazenem esta árvore enquanto exploram a estrutura do grafo. A

[Figura 3.1](#) ilustra um exemplo simples de grafo. Se tomarmos o vértice s como vértice inicial, teríamos a árvore de busca mostrada ao lado do grafo.

Figura 3.1 – Exemplo de árvore de busca em largura



Fonte: o autor.

A árvore de busca em largura é uma árvore n -ária, pois não sabemos a priori quantos filhos cada nó da árvore terá. Nas implementações de árvores n -árias, geralmente cada nó possui uma lista encadeada de referências para os nós filhos. No caso de árvores de busca, como o objetivo principal é apenas armazenar os caminhos percorridos, podemos encarar as coisas por outro ângulo: não sabemos quantos filhos terá cada nó, mas podemos ter certeza de que cada nó tem apenas um único pai. Assim, ao invés de armazenar os n filhos de cada nó, podemos armazenar somente uma referência para seu pai, resultando em uma estrutura de dados muito mais simples, que pode ser implementada com apenas um vetor de referências para vértices.

A este vetor, damos o nome de **vetor de roteamento** ou **vetor de predecessores** e o denotamos pelo símbolo $R(G)$. Assim, a árvore de busca em largura da [Figura 3.1](#) corresponde aos seguintes valores no vetor de roteamento: $\rho(s) = \text{nil}$, $\rho(a) = s$, $\rho(b) = a$, $\rho(c) = s$, $\rho(d) = a$ e $\rho(e) = a$.

De forma a facilitar o texto, podemos representar o vetor de roteamento $R(G)$ conforme mostrado no [Quadro 2](#).

Quadro 2 – Exemplo de vetor de roteamento $R(G)$

v	s	a	b	c	d	e
$\rho(v)$	nil	s	a	s	a	a

Fonte: o autor.

3.2.1 O algoritmo de busca em largura

A seguir, veremos o pseudocódigo do algoritmo de busca em largura, também conhecido como BFS (do inglês *Breath First Search*) ([CORMEN et al., 2001](#)). Na implementação deste pseudocódigo, cada vértice possui um atributo chamado *estado*, que indica

seu estado atual na busca (não visitado, ou visitado), além disso, a busca utiliza uma fila de vértices, que conduz a ordem de visitação deles. Os elementos do pseudocódigo são os seguintes:

- r : representa o vértice inicial;
- $estado(v)$: estado do vértice v ao longo da execução do algoritmo, que pode assumir um dos seguintes valores $\{NAO_VISITADO, VISITADO, ENCERRADO\}$. O valor $NAO_VISITADO$ significa que a busca ainda não atingiu este vértice. O valor $VISITADO$, indica que a busca já atingiu o vértice v , mas ele ainda tem vizinhos (vértices adjacentes) ainda não visitados. O valor $ENCERRADO$ indica que a visitação deste vértice está encerrada, isto é, tanto ele como todos os seus vizinhos já foram visitados pela busca;
- $\rho(v)$: referência ao vértice predecessor do vértice v na busca, representa seu pai na árvore de busca em largura;
- F : fila de vértices, que deve implementar as operações $F.INSERE(v)$, responsável pela inserção do vértice v no final da fila, e $F.REMOVE()$, que remove e retorna o primeiro vértice da fila F ;
- $d(v)$: atributo numérico que armazena a distância de s até v em quantidade de arestas percorridas.
- $adj(v)$: é uma iteração com todos os vértices adjacentes ao vértice v (veja seção 2.2).

Algoritmo: BuscaEmLargura(G, r)

```

para cada  $v \in V$  faça
     $estado(v) \leftarrow NAO\_VISITADO;$ 
     $\rho(v) \leftarrow nil;$ 
     $d(v) \leftarrow \infty;$ 

     $d(r) \leftarrow 0;$ 
     $estado(r) \leftarrow VISITADO;$ 
     $F \leftarrow \emptyset;$ 
     $F.INSERE(r);$ 

enquanto  $F \neq \emptyset$  faça
     $v_i \leftarrow F.REMOVE();$ 
    para cada  $v_j \in Adj(v_i)$  faça
        se  $estado(v_j) = NAO\_VISITADO$  então
             $F.INSERE(v_j);$ 
             $estado(v_j) \leftarrow VISITADO;$ 
             $\rho(v_j) \leftarrow v_i;$ 
             $d(v_j) \leftarrow d(v_i) + 1;$ 

         $cor(v_i) \leftarrow ENCERRADO;$ 
    
```

Algoritmo 3.1: Algoritmo de busca em largura ([CORMEN et al., 2001](#))

3.3 Busca em profundidade

A busca em profundidade, também chamada de *DFS* (do inglês *Depth First Search*), pesquisa o grafo de uma forma recursiva. Ela explora o grafo procurando sempre entrar mais profundamente em sua estrutura. Ao atingir um determinado vértice, a busca visita seu vizinho, depois o vizinho do vizinho e assim susseivamente. Quando a busca não consegue mais ir adiante, ela volta gradativamente aos passos anteriores até que consiga encontrar uma nova opção de caminho. Este procedimento de voltar atrás é comumente conhecido como *backtracking*.

3.3.1 Busca em profundidade com um vértice origem

Assim como na busca em largura, a busca em profundidade também gera uma árvore de busca, porém ela não tem a propriedade de gerar os caminhos mínimos.

Para maior clareza, o algoritmo de busca em profundidade é desmembrado em mais de um procedimento. O primeiro deles ([algoritmo 3.2](#)) apresenta o pseudocódigo do laço principal da busca em profundidade. De forma semelhante ao algoritmo de busca em largura, podemos escolher um vértice para iniciar a busca. Este algoritmo também atinge todos os vértices possíveis de serem atingidos a partir do vértice inicial s , porém **não há garantia** de que a árvore de busca gerada contenha os menores caminhos de s até cada vértice.

Algoritmo: BuscaEmProfundidade(G, r)

```

para cada  $v \in V$  faça
     $estado(v) \leftarrow NAO\_VISITADO;$ 
     $\rho(v) \leftarrow nil;$ 
tempo  $\leftarrow 0;$ 
VisitaVertice( $r$ );

```

Algoritmo 3.2: Algoritmo de busca em profundidade com um vértice de origem

Os elementos do pseudocódigo são os seguintes:

- r : representa o vértice inicial;
- $estado(v)$: este atributo representa o estado de visitação do vértice v ao longo da execução do algoritmo, podendo assumir um dos seguintes valores $\{NAO_VISITADO, VISITADO, ENCERRADO\}$. O valor $NAO_VISITADO$ significa que a busca ainda não atingiu este vértice. O valor $VISITADO$, indica que a busca já atingiu o vértice v , mas ele ainda tem vizinhos (vértices adjacentes) ainda não visitados. O valor $ENCERRADO$ indica que a visitação deste vértice está encerrada, isto é, tanto ele como todos os seus vizinhos já foram visitados pela busca;

- $\rho(v)$: referência ao vértice predecessor do vértice v na busca, representa seu pai na árvore de busca em profundidade;
- *tempo*: Cormen et al. (2001) propõem o uso de um relógio lógico marcando a ordem de visitação dos vértices. Este dado pode ser utilizado na resolução de alguns problemas que envolvem a busca em profundidade como parte do método de resolução. O relógio lógico implementa uma técnica chamada de *time stamping*, que rotula com números inteiros uma sequência de eventos. No caso do algoritmo de busca em profundidade, estes valores variam de 0 a $2n$;
- $t_a(v)$: tempo de descoberta ou abertura do vértice v , representa o valor do tempo lógico quando o vértice é visitado pela primeira vez;
- $t_e(v)$: tempo de encerramento do vértice v , representa o valor do tempo lógico quando a visitação no vértice é encerrada.

O [algoritmo 3.3](#) mostra o pseudocódigo da função $VisitaVertice(v)$. Trata-se de uma função recursiva que efetivamente implementa o procedimento de busca em profundidade. e que vai visitar todos os vértices que puderem ser atingidos a partir do vértice inicial r . A recursividade da função caracteriza um empilhamento dos vértices a serem visitados, com isso a ordem de visitação é feita em profundidade, sempre visitando o primeiro vértice não-visitado adjacente ao vértice atual.

```

Algoritmo: VisitaVertice( $v_i$ )
   $estado(v_i) \leftarrow VISITADO$ ;
   $tempo \leftarrow tempo + 1$ ;
   $t_a(v_i) \leftarrow tempo$ ;
  para cada  $v_j \in Adj(v_i)$  faça
    se  $estado(v_j) = NAO\_VISITADO$  então
       $\rho(v_j) \leftarrow v_i$ ;
       $VisitaVertice(v_j)$ ;
   $estado(v_i) \leftarrow ENCERRADO$ ;
   $tempo \leftarrow tempo + 1$ ;
   $t_e(v_i) \leftarrow tempo$ ;

```

Algoritmo 3.3: Função VisitaVertice(v_i)

3.3.2 Busca em profundidade com várias origens

Em alguns casos, a busca a partir de um vértice inicial r não consegue atingir todos os vértices. Isto pode acontecer em grafos dirigidos e em grafos não-conexos ([capítulo 4](#)). Nestes casos, se for necessário visitar todos os vértices, pode-se iniciar sucessivas buscas em profundidade. Sempre que uma busca encerrar, inicia-se uma nova busca a partir de um vértice ainda não visitado. O [algoritmo 3.4](#) demonstra este procedimento.

Algoritmo: BuscaProfTodos(G)

```

para cada  $v \in V$  faz
  |  $estado(v) \leftarrow NAO\_VISITADO;$ 
  |  $\rho(v) \leftarrow nil;$ 
  |
  |  $tempo \leftarrow 0;$ 
para cada vértice  $v \in V$  faz
  | se  $estado(v) = NAO\_VISITADO$  então
    |   | VisitaVertice( $v$ );
  |

```

Algoritmo 3.4: Algoritmo BuscaProfTodos(G)

O laço principal da função BuscaProfTodos(G) percorre todos os vértices do grafo G , e chama VisitaVertice(v) sempre que o vértice corrente v for um vértice ainda não visitado. Ao final do laço principal, todos os vértices são visitados, e temos no vetor de roteamento uma árvore para cada uma das buscas que eventualmente foram feitas. No vetor de roteamento, os vértices que originaram as buscas são raízes das árvores, e seus predecessores têm valor *nil*.

3.4 Percorrendo a árvore de busca

Os caminhos gerados pelos algoritmos de busca, armazenados em vetores de roteamento, podem ser reconstituídos por meio do [algoritmo 3.5](#). Note que o algoritmo é recursivo e só funciona para caminhos começando no vértice inicial utilizado no algoritmo de busca que gerou o correspondente vetor de roteamento. Tal vértice sempre é a raiz r de uma árvore de busca

Algoritmo: ImprimeCaminho(G, r, v)

```

se  $v=r$  então
  |  $imprime(r);$ 
senão
  | se  $\rho(v) = nil$  então
    |   |  $imprime("não existe caminho de r para v");$ 
  | senão
    |   |  $ImprimeCaminho(G, r, \rho(v));$ 
    |   |  $imprime(v);$ 
  |

```

Algoritmo 3.5: Procedimento para imprimir caminhos

3.5 Exemplos de aplicação de busca

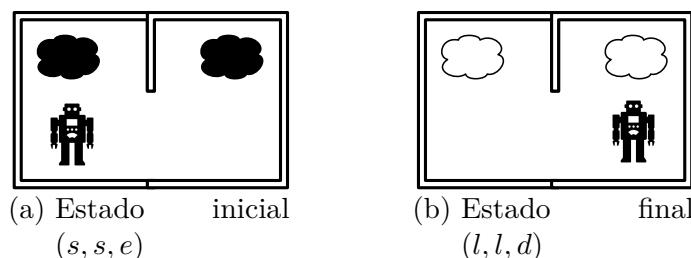
Conforme mencionado anteriormente, os algoritmos de busca em grafos podem ser utilizados diretamente na resolução de determinados problemas, mas também podem servir de ponto de partida para algoritmos mais especializados que precisem caminhar pelo grafo passando por arestas. A seguir, apresentamos dois estudos de caso que aplicam busca em grafos.

3.5.1 Estudo de caso: busca em grafos de estados

Um grafo de estados $G = (V, E)$ representa nas quais as transições entre possíveis estados de um problema são importantes. Um exemplo clássico na Ciência da Computação são as máquinas de estados finitos, que podem ser modeladas por tipos especiais de grafos chamados Autômatos Finitos Determinísticos ([HOPCROFT; MOTWANI; ULLMAN, 2006](#)). Um outro exemplo bastante popular é a programação de tarefas de um robô. Por exemplo, de forma bem simplificada, suponha que um robô aspirador de pó precisa limpar uma área dividida em dois cômodos: o quarto do lado direito (q_d) e o quarto esquerdo (q_e). O robô executa apenas três comandos: ir para o quarto esquerdo, ir para o quarto direito e limpar o quarto atual. Considerando que o robô se encontra inicialmente no quarto da esquerda e ambos os quartos estão sujos, nossa tarefa é montar um plano para que o robô limpe os dois quartos e, ao final, se posicione no quarto da direita.

Vamos montar este plano por meio de um grafo de estados. Cada vértice representa um estado do problema e cada aresta modela uma possível transição entre dois estados. Um estado do problema é caracterizado pela condição atual de limpeza de cada quarto (limpo ou sujo) e pela posição atual do robô (quarto esquerdo ou quarto direito). Podemos representar os vértices por uma tupla de 3 valores (Q_e, Q_d, P_r) , sendo Q_e e Q_d os conjuntos de estados de limpeza dos dois quartos, ou seja, com $Q_e = \{l, s\}$ e $Q_d = \{l, s\}$. P_r é o conjunto de possíveis localizações do robô, $P_r = \{e, d\}$. O estado inicial do problema, portanto, é representado pela tupla (s, s, e) e o estado final pela tupla (l, l, d) , conforme ilustrado na [Figura 3.2](#).

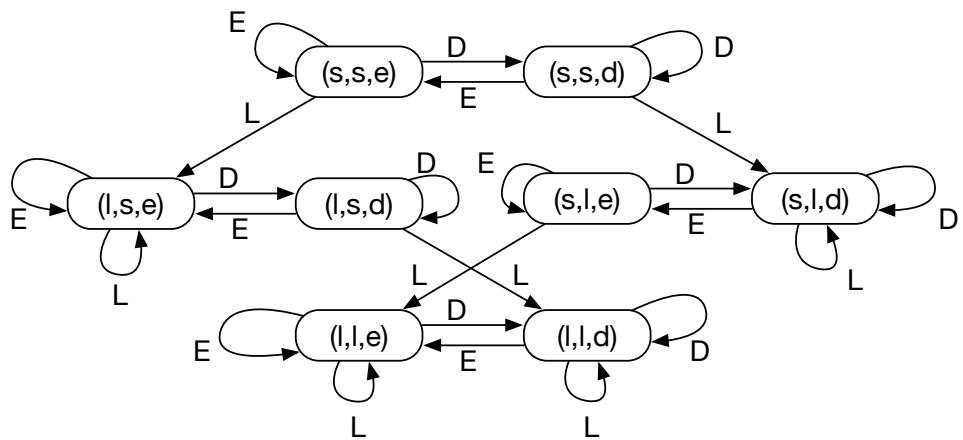
Figura 3.2 – Estados inicial e final do problema



Fonte: o autor.

Aqui surge uma questão interessante sobre a modelagem de problemas com grafos: nosso problema será modelado como um grafo dirigido ou não-dirigido? Na verdade, isso é decorrente da natureza das relações entre os elementos (vértices) do problema. Neste caso os únicos fatores que alteram o estado do problema e, consequentemente, serão as arestas do garfo, são os comandos aceitos pelo robô: ir para o quarto esquerdo, ir para o quarto direito e limpar. Como não existe o comando para devolver a sujeira, uma vez que o quarto foi limpo ele não volta mais ao estado sujo. Isso nos indica que se trata de um grafo dirigido, que pode ser visto na [Figura 3.3](#). Os rótulos E, D e L das arestas correspondem aos comandos “ir para a esquerda”, “ir para a direita” e limpar, respectivamente.

Figura 3.3 – Grafo de estados do robô



Fonte: o autor.

Finalmente, para encontrar uma sequência de comandos para montar o plano de atuação do robô, basta efetuar uma busca no grafo, iniciando pelo vértice (s, s, e) . A resposta pode ser encontrada percorrendo-se a árvore de busca para determinar a sequência de comandos que gera as transições de estado até o estado final. Por exemplo: $(s, s, e) \rightarrow$ limpa $\rightarrow (l, s, e) \rightarrow$ direita $\rightarrow (l, s, d) \rightarrow$ limpa $\rightarrow (l, l, d) \rightarrow$ esquerda $\rightarrow (l, l, e)$.

Note que em problemas deste tipo, tanto a busca em largura quanto a busca em profundidade podem ser utilizadas, entretanto, por garantir encontrar os caminhos com menor número de arestas, a busca em largura garante que a solução use a menor quantidade possível de transições de estado.

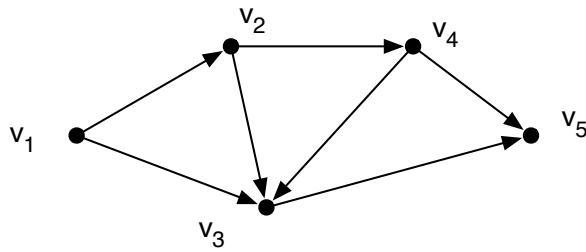
3.5.2 Estudo de caso: ordenação topológica

A ordenação topológica em um grafo acíclico dirigido é uma ordenação linear de vértices tal que para cada aresta dirigida (v_i, v_j) , o vértice v_i vem antes de v_j na ordenação. Este tipo de ordenação só é possível em grafos acíclicos dirigidos.

Definição 3.1 (Grafo acíclico dirigido). *Um grafo acíclico dirigido é um grafo dirigido sem ciclos dirigidos fechados.*

A [Figura 3.4](#) mostra um exemplo de grafo acíclico dirigido. Note que se caminharmos pelo grafo saindo de um vértice v arbitrário, nunca conseguimos retornar para v .

Figura 3.4 – Exemplo de grafo acíclico dirigido



Fonte: o autor.

Uma ordenação topológica pode ser vista como uma organização linear dos vértices de um grafo de tal forma que vértices subsequentes dependem dos vértices precedentes, com estas relações de dependência modeladas pelas arestas. Isto é particularmente útil quando o grafo modela conjuntos interdependentes de atividades, como nos diagramas PERT, usados para planejamento e controle de projetos ([KERZNER, 2005](#)).

O algoritmo para ordenação topológica utiliza a busca em profundidade, e pode ser descrito pelos seguintes passos:

Entrada: um grafo acíclico dirigido G .

Passos:

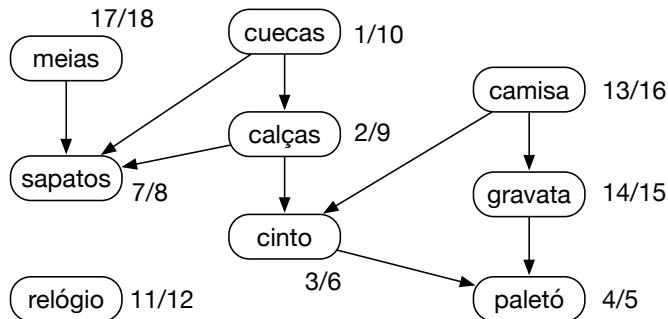
1. Execute a busca em profundidade no grafo — $BuscaProfTodos(G)$;
2. À medida que cada vértice for encerrado (isto é, quando for computado seu tempo de encerramento $t_e(v)$), insira o vértice no início de uma lista encadeada;
3. Retorne a lista encadeada de vértices.

Saída: uma lista encadeada ordenada de vértices.

Para ilustrar um exemplo de ordenação topológica, vamos organizar a sequência de passos para um homem se vestir com terno e gravata, incluindo sapatos e relógio. No grafo acíclico dirigido da [Figura 3.5](#) os vértices são as peças de roupa e as arestas mostram as relações de dependência entre as peças. Por exemplo, só se pode calçar os sapatos depois de calçar as meias. Por isso, existe uma aresta dirigida ligando o vértice “meias” até o vértice “sapatos” indicando que os “sapatos” dependem das “meias”. Os números próximos

a cada vértice representam os seus tempos de abertura e fechamento em uma execução da busca em profundidade.

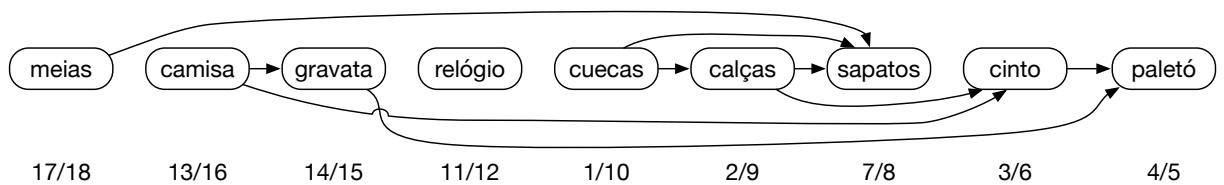
Figura 3.5 – Grafo acíclico dirigido



Fonte: adaptado de [Cormen et al. \(2001\)](#).

A [Figura 3.6](#) mostra a ordenação topológica obtida com a busca em profundidade realizada. A sequência de vértices encontrada é decorrente da ordem de visitação dos vértices, que em última instância, é também função de como os vértices estão dispostos nas estruturas de dados que modelam o grafo. Note que, por mais incomum que possa parecer esta sequência de passos para uma pessoa se vestir, ela é semanticamente correta, conforme evidenciado pelas arestas dirigidas na [Figura 3.6](#).

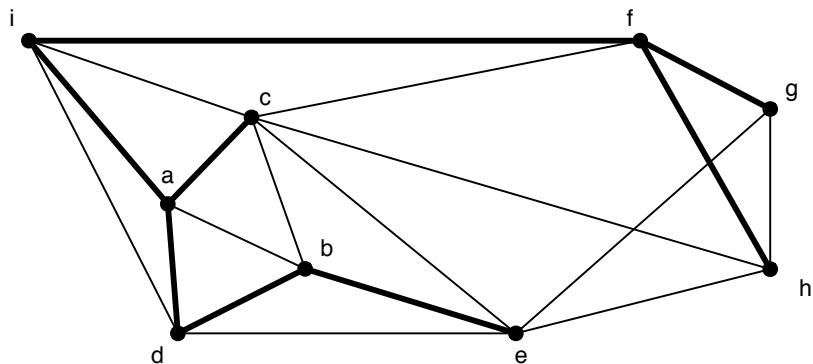
Figura 3.6 – Resultado da ordenação topológica



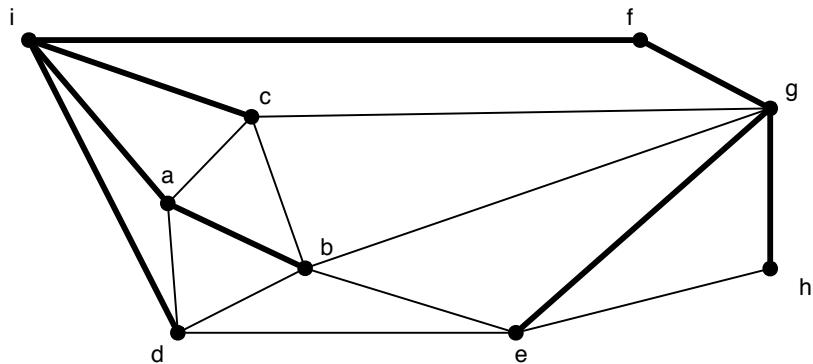
Fonte: o autor.

3.6 Exercícios

1. A figura abaixo mostra uma árvore de busca contendo caminhos do vértice a para todos os demais vértices. Preencha a tabela abaixo com os respectivos valores do vetor de roteamento que representa a árvore.



2. A figura abaixo mostra uma árvore de busca contendo caminhos do vértice f para todos os demais vértices. Preencha a tabela abaixo com os respectivos valores do vetor de roteamento que representa a árvore.



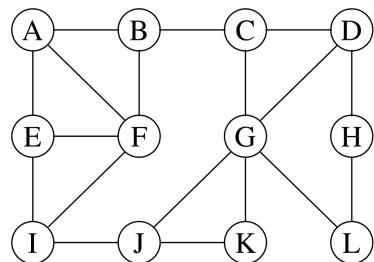
3. Dado o vetor de roteamento abaixo, escreva a sequência de vértices de cada caminho (se existir caminho):

v	a	b	c	d	e	f	g	h	i	j
$\rho(v)$	b	c	d	e	h	d	nil	g	j	h

- a) Caminho $g \rightsquigarrow c$: _____
- b) Caminho $b \rightsquigarrow f$: _____
- c) Caminho $g \rightsquigarrow i$: _____
4. Dado o vetor de roteamento abaixo, escreva a sequência de vértices de cada caminho (se existir caminho):

v	a	b	c	d	e	f	g	h	i	j	f	l	m	n
$\rho(v)$	j	j	f	i	h	nil	m	i	nil	f	j	k	h	g

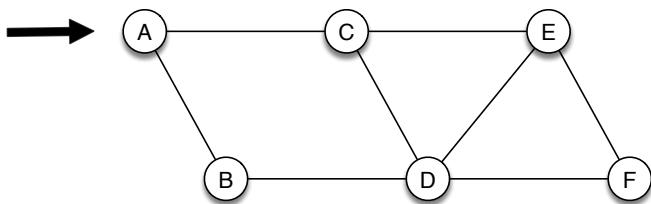
- a) Caminho $i \rightsquigarrow n$: _____
- b) Caminho $f \rightsquigarrow k$: _____
- c) Caminho $g \rightsquigarrow h$: _____
5. A figura abaixo mostra um grafo não dirigido, e um vetor de roteamento gerado neste grafo ao se executar um algoritmo de busca.



v	a	b	c	d	e	f	g	h	i	j	f	l
$\rho(v)$	nil	a	b	c	i	e	d	l	j	g	l	g

- a) A árvore foi gerada por busca em largura ou profundidade? _____
- b) Qual o caminho $a \rightsquigarrow e$? _____
- c) Qual o comprimento do caminho $a \rightsquigarrow e$? _____
6. Elabore um algoritmo para verificar se um grafo possui ciclos.
7. Implemente o algoritmo de busca em largura ([algoritmo 3.1](#)).
8. Implemente o algoritmo de busca em profundidade ([algoritmo 3.2](#) e [algoritmo 3.3](#)).

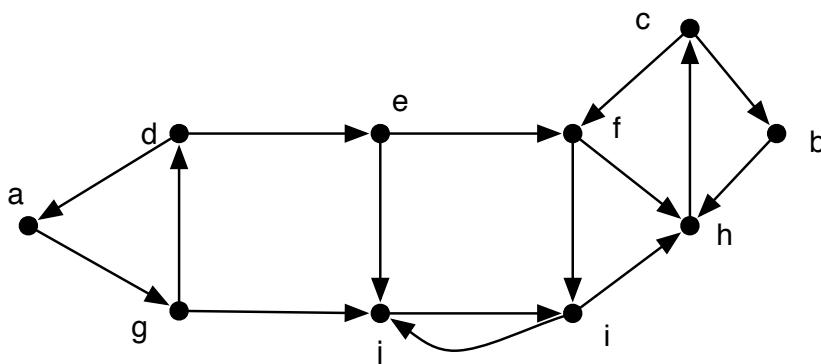
9. (PosComp – 2009) Considere o algoritmo de busca em largura em grafos. Dado o grafo a seguir e o vértice A como ponto de partida, a ordem em que os vértices são descobertos é dada por:



- a) A B C D E F
- b) A B D C E F
- c) A C D B F E
- d) A B C E D F
- e) A B D F E C

10. Dado o grafo G dirigido abaixo, mostre passo a passo a execução de $BuscaProfTodos(G)$, considerando que as listas de adjacência estão em ordem alfabética:

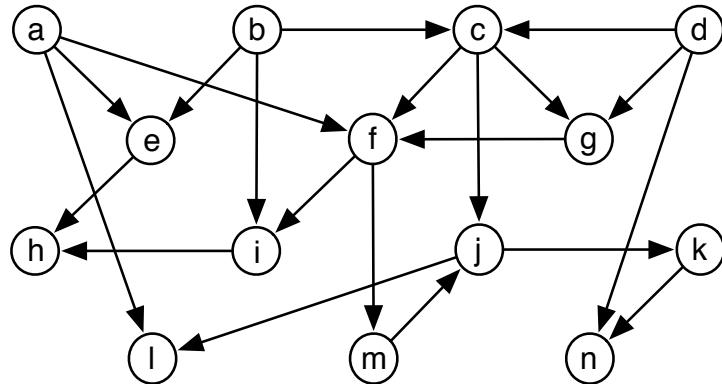
- a) Mostre a sequencia de chamadas a $VisiteVertice$;
- b) Mostre os tempos de abertura e encerramento de todos os vértices no grafo G ;
- c) Mostre o vetor de roteamento final.



11. (PosComp – 2011) Seja G um grafo conexo com n vértices. Considere duas rotulações dos vértices de G obtidas por duas buscas em G , uma em largura, $l()$, e outra em profundidade, $p()$, ambas iniciadas no vértice v . Em cada rotulação, os vértices receberam um número de 1 a n , o qual representa a ordem em que foram alcançados na busca em questão. Assim, $l(v) = p(v) = 1$; enquanto $l(x) > 1$ e $p(x) > 1$ para todo vértice x diferente de v . Considere dois vértices u e w de G e denote por $d(u, w)$ a distância em G de u até w . Com base nesses dados, assinale a alternativa correta.

- a) Se $l(u) < l(w)$ e $p(u) < p(w)$, então $d(v, u) < d(v, w)$.
- b) Se $l(u) < l(w)$ e $p(u) > p(w)$, então $d(v, u) = d(v, w)$.
- c) Se $l(u) > l(w)$ e $p(u) < p(w)$, então $d(v, u) \leq d(v, w)$.
- d) Se $l(u) > l(w)$ e $p(u) > p(w)$, então $d(v, u) < d(v, w)$.
- e) Se $l(u) < l(w)$ e $p(u) > p(w)$, então $d(v, u) \leq d(v, w)$.

12. (PosComp – 2015) Centenas de problemas computacionais são expressos em termos de grafos, e os algoritmos para resolvê-los são fundamentais para a computação. O algoritmo de busca em:
- a) largura utiliza pilha, enquanto o de busca em profundidade utiliza fila.
 - b) largura é o responsável pela definição do vértice inicial.
 - c) profundidade é utilizado para obter uma ordenação topológica em um dígrafo acíclico.
 - d) largura explora as arestas a partir do vértice mais recentemente visitado.
 - e) profundidade expande a fronteira entre vértices conhecidos e desconhecidos uniformemente.
13. Faça a ordenação topológica do dígrafo abaixo. Utilize o algoritmo de busca em profundidade e mostre a lista de vértices ordenados.



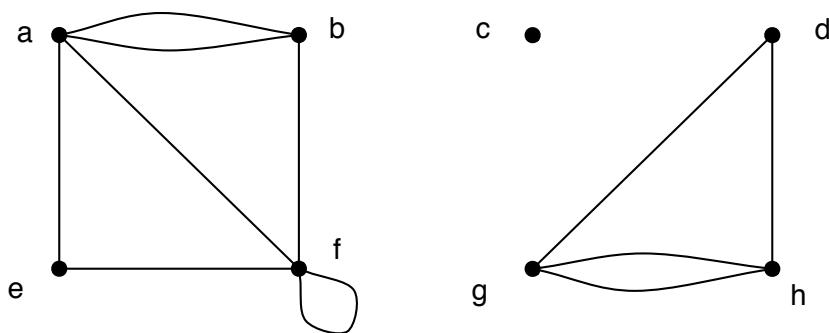
4 Conexidade

A conexidade trata de questões referentes a partes de grafos estarem conectadas ou não. O capítulo inicia com uma contextualização geral sobre o tema, seguida por uma série de definições teóricas iniciais. São então apresentados diversos algoritmos para a avaliação de conexidade de grafos não dirigidos e dirigidos.

4.1 Introdução

A conexidade trata de questões referentes a partes de grafos estarem conectadas ou não. Para ilustrar o assunto, considere o grafo não dirigido $G = (V, E)$ em que $V = \{a, b, c, d, e, f, g, h\}$ e $E = \{(a, b), (a, b), (e, a), (f, a), (b, f), (f, f), (d, g), (h, g), (h, g), (d, h), (e, f)\}$. Uma representação gráfica deste grafo é ilustrada na [Figura 4.1](#). Intuitivamente, podemos observar que existem três partes distintas neste grafo, completamente separadas entre si. Aparentemente isto constituiria três grafos, mas na verdade trata-se de um grafo só (porque ele seria resultado da modelagem de um único problema). A conexidade refere-se a este tipo de situação.

Figura 4.1 – Exemplo de conexidade



Fonte: o autor.

4.2 Definições iniciais

A seguir, veremos algumas definições importantes para compreendermos os problemas de conexidade em grafos dirigidos e não dirigidos, e também os algoritmos que tratam esses problemas.

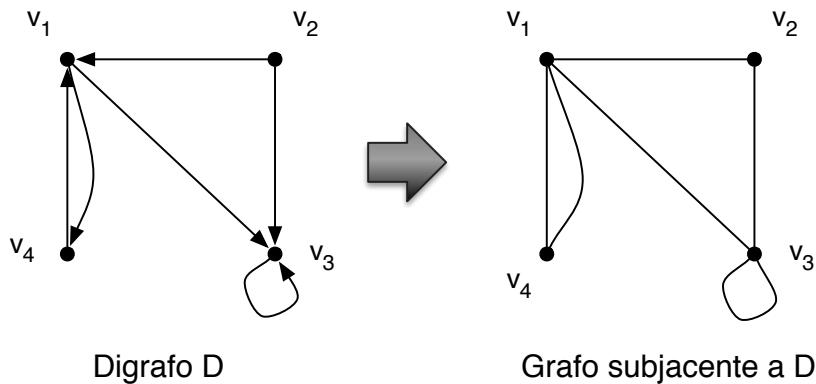
Definição 4.1 (Grafo conexo). *Um grafo G é conexo se, para todos os pares de vértices u e v do grafo, existe pelo menos um caminho de u para v . Caso contrário, o grafo é chamado não conexo.*

Definição 4.2 (Componente conexa). *Uma componente conexa de um grafo G é um subgrafo conexo maximal de G .*

O grafo da [Figura 4.1](#) possui três componentes conexas, que compreendem os seguintes conjuntos de vértices: $C_1 = \{a, b, e, f\}$, $C_2 = \{c\}$ e $C_3 = \{d, g, h\}$

Definição 4.3 (Grafo subjacente). *Dado um digrafo D , seu grafo subjacente é um grafo não dirigido obtido pela substituição de todas as arestas dirigidas de D por arestas não dirigidas. A [Figura 4.2](#) mostra um grafo dirigido $D = (V, E)$ e seu grafo subjacente.*

Figura 4.2 – Exemplo de grafo subjacente a um digrafo



Fonte: o autor.

Definição 4.4 (Digrafo conexo). *Um digrafo D é (fracamente) conexo se seu grafo subjacente for conexo.*

Se pensarmos do ponto de vista de programação de computadores, podemos considerar um grafo como sendo uma variável ou um objeto numa determinada linguagem de programação. Sendo assim, a configuração que resulta da adição ou remoção de um vértice ou aresta de um grafo G pode ser considerado um novo valor para G .

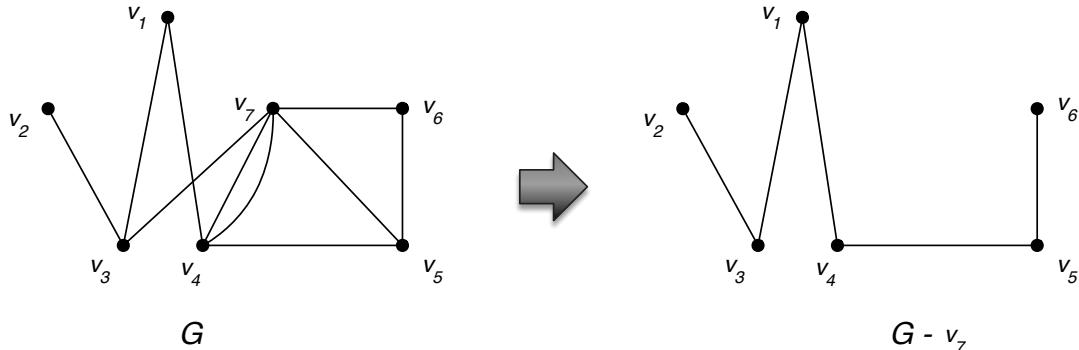
As definições abaixo estabelecem algumas operações simples em grafos e poderiam, por exemplo, serem implementadas em uma classe representativa de grafos.

Definição 4.5 (Subgrafo de remoção de vértice). *Se v é um vértice do grafo G , então o subgrafo de remoção de vértice $G - v$ é o subgrafo induzido pelo conjunto de vértices $V_G - v$, isto é:*

$$V_{G-v} = V_G - v \quad e \quad E_{G-v} = \{e \in E_G : v \notin \text{endpts}(e)\}$$

Exemplo 4.1. A *Figura 4.3* mostra o resultado da subtração do vértice v_7 do grafo G .

Figura 4.3 – Exemplo de remoção de vértice

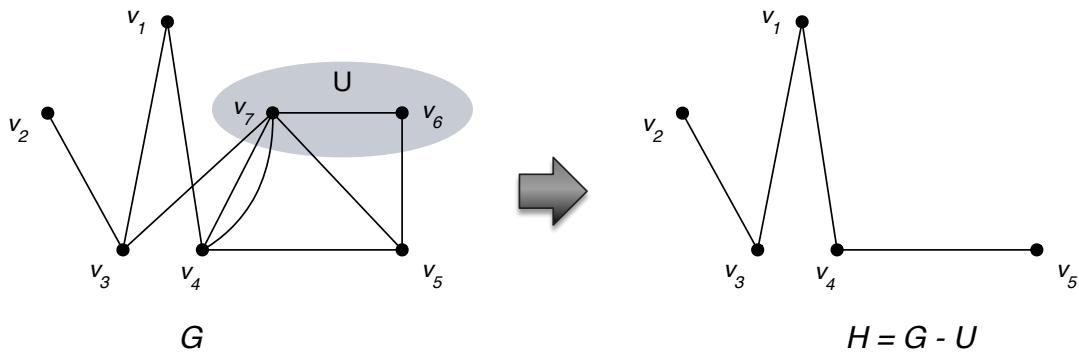


Fonte: o autor.

De forma geral, se tivermos um conjunto U tal que $U \subseteq V_G$, o resultado da aplicação sucessiva da remoção de todos os vértices de U em G é denotado por $G - U$.

Exemplo 4.2. Na *Figura 4.4*, temos o grafo U definido por $V_U = \{v_6, v_7\}$ e $E_U = (v_6, v_7)$. Ao aplicarmos a operação geral de subtração de grafos $G - U$, temos como resultado o grafo H .

Figura 4.4 – Exemplo de subtração de grafos



Fonte: o autor.

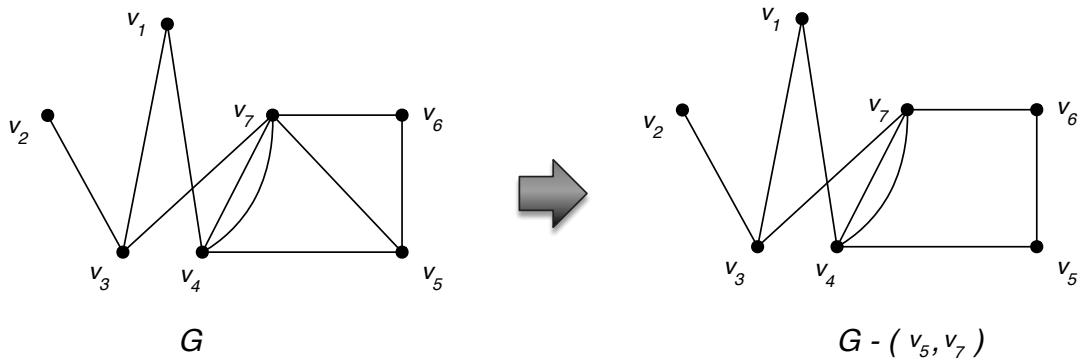
Definição 4.6 (Subgrafo de remoção de aresta). Se e é uma aresta do grafo G , então o subgrafo de remoção de aresta $G - e$ é o subgrafo induzido pelo conjunto de arestas $E_G - e$, isto é:

$$V_{G-e} = V_G \quad e \quad E_{G-e} = E_G - e$$

De forma geral, se tivermos um conjunto D tal que $D \subseteq E_G$, o resultado da aplicação sucessiva da remoção de todas as arestas de D em G é denotado por $G - D$.

Exemplo 4.3. Na [Figura 4.5](#), temos um exemplo da operação de remoção de aresta (v_5, v_7) do grafo G .

Figura 4.5 – Exemplo de remoção de aresta



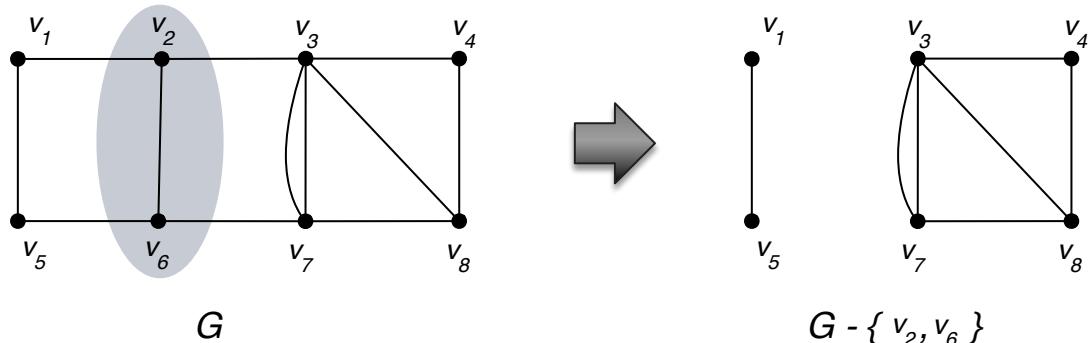
Fonte: o autor.

As definições de subgrafos de remoção de vértices e de arestas para grafos dirigidos são análogas às dos grafos não dirigidos.

Definição 4.7 (Corte de Vértices). *Um corte de vértices em um grafo G é é um conjunto de vértices U tal que $G - U$ tem mais componentes conexas que G .*

Exemplo 4.4. Na [Figura 4.6](#), temos um exemplo de corte de vértices representado pelo subconjunto de vértices $\{v_2, v_6\}$ do grafo G . Após a operação $G - \{v_2, v_6\}$ o grafo inicialmente conexo passa a ter duas componentes conexas. Portanto, o conjunto $\{v_2, v_6\}$ é um corte de vértices do grafo G . Note que o grafo G tem outro corte de vértices, que é o conjunto $\{v_3, v_7\}$.

Figura 4.6 – Exemplo de corte de vértices

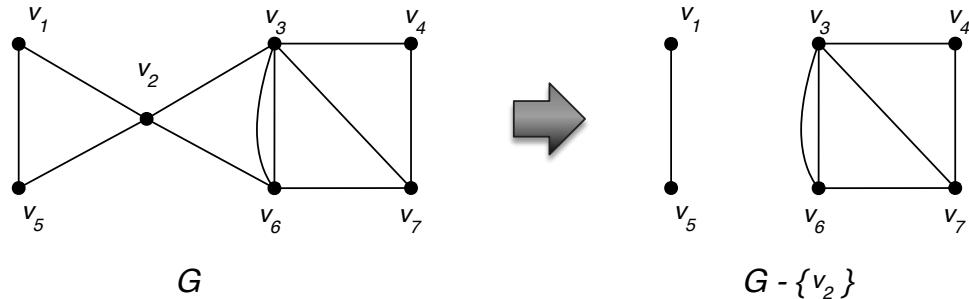


Fonte: o autor.

Definição 4.8 (Vértice de corte). Um vértice de corte é um corte de vértices constituído por apenas um vértice.

Exemplo 4.5. A [Figura 4.7](#) ilustra um exemplo de vértice de corte. Neste caso, a remoção do vértice v_2 do grafo conexo G faz com que ele passe a ter duas componentes conexas.

Figura 4.7 – Exemplo de vértice de corte



Fonte: o autor.

Definição 4.9 (Corte de arestas). Um corte de arestas em um grafo G é um conjunto de arestas D , tal que, $G - D$ possui mais componentes conexas que G .

Definição 4.10 (Aresta de corte ou “ponte”). Uma aresta de corte ou ponte é um corte de arestas constituído por apenas uma aresta.

As operações de remoção de vértices e de arestas nos mostram que alguns grafos são “mais conexos” que outros. Por exemplo, alguns grafos podem ser desconectados com a remoção de uma única aresta ou então um vértice. Por outro lado, há outros grafos que permaneceriam conectados a não ser que houvesse remoção de uma quantidade maior de arestas e/ou vértices. Os conceitos de conectividade de vértices e conectividade de arestas nos trazem parâmetros numéricos que indicam o grau de “conectividade” dos grafos. Estes conceitos são úteis para se avaliar o nível de vulnerabilidade de redes de informação, malhas rodoviárias ou qualquer outro tipo de rede de comunicação ou transporte.

Definição 4.11 (Conectividade de vértices). A conectividade de vértices de um grafo G , denotada por $\kappa(G)$ é o número mínimo de vértices cuja remoção pode desconectar G ou reduzi-lo a um grafo com apenas um vértice.

Portanto, se G possuir pelo menos um par de vértices não-adjacentes, então $\kappa(G)$ é igual ao tamanho de seu menor corte de vértices.

Definição 4.12 (k -conexão). Um grafo G é k -conexo (ou k -conectado) se G é conexo e $\kappa_v(G) \geq k$. Se G possui vértices não adjacentes, então G é k -conexo se cada um de seus cortes de vértices tiver pelo menos k vértices.

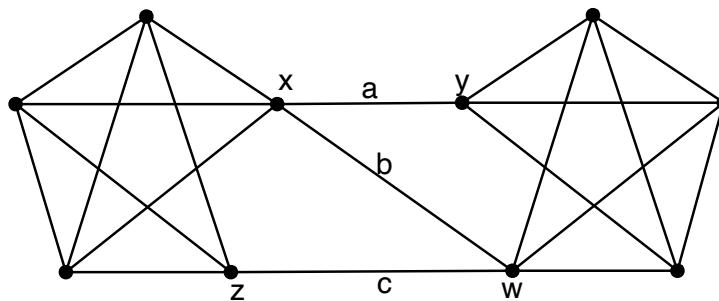
Definição 4.13 (Conectividade de arestas). A *conectividade de arestas* de um grafo G , denotada por $\gamma(G)$ é o número mínimo de arestas cuja remoção pode desconectar G .

Portanto, se G for um grafo conexo, então $\gamma(G)$ é igual ao tamanho de seu menor corte de arestas.

Definição 4.14 (k -conexão-de-arestas). Um grafo G é k -conexo-de-arestas (ou k -conectado-de-arestas) se G é conexo e se cada um de seus cortes de aresta tem no mínimo k arestas (isto é, $\gamma(G) \geq k$).

Exemplo 4.6. No grafo G da Figura 4.8, o conjunto de vértices $\{x, z\}$ é um dos seus três cortes de vértices com dois elementos. Neste grafo também é relativamente fácil perceber que não há vértice de corte. Portanto, $\kappa(G) = 2$. O conjunto de arestas $\{a, b, c\}$ é o único corte de arestas com 3 elementos no grafo G , e não há corte de arestas com menos de 3 elementos. Portanto, $\gamma(G) = 3$.

Figura 4.8 – Grafo G com $\kappa(G) = 2$ e $\gamma(G) = 3$



Fonte: o autor.

4.3 Conexidade com algoritmo de busca

Um problema comum é avaliar se determinado grafo é conexo e, não sendo, determinar a sua quantidade de componentes conexas. Pela definição 4.1, podemos inferir uma forma de fazer isso por meio do uso de algoritmos de busca, visto que sua principal função é encontrar caminhos nos grafos. Esta solução foi descrita por Hopcroft e Tarjan em 1973 ([HOPCROFT; TARJAN, 1973](#)).

Caso se queira utilizar o algoritmo de busca em profundidade, por exemplo, basta adaptá-lo para contabilizar a quantidade de buscas recursivas que são executadas. Cada uma delas corresponde a uma componente conexa no grafo.

O algoritmo 4.1 mostra o pseudocódigo desta solução. Da mesma forma que no algoritmo anterior, a variável N_{cc} é um valor inteiro que armazena a quantidade de

componentes conexas do grafo. Consequentemente, no vetor de roteamento construído pelo algoritmo, cada árvore corresponde a uma componente conexa. A função $VisitaVertice(v)$ pode ser vista na [subseção 3.3.1](#).

Algoritmo: ConexidadeBuscaProfundidade(G)

```

para cada  $v \in V$  faz
  |  $estadov) \leftarrow NAO\_VISITADO;$ 
  |  $\rho(v) \leftarrow nil;$ 

   $tempo \leftarrow 0;$ 
   $N_{cc} \leftarrow 0;$ 

  para cada vértice  $v \in V$  faz
    | se  $cor(v) = NAO\_VISITADO$  então
      |   |  $N_{cc} \leftarrow N_{cc} + 1;$ 
      |   |  $VisitaVertice(v);$ 

    | se  $N_{cc} > 1$  então
      |   | imprima(“Grafo não conexo, com  $N_{cc}$  componentes conexas”)
    | senão
      |   | imprima(“Grafo conexo”)
  
```

Algoritmo 4.1: Avaliação de conexidade com busca em profundidade

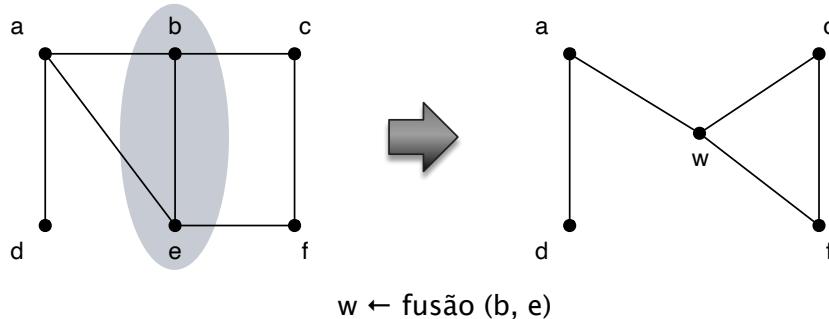
É possível fazer o mesmo tipo de adaptação no algoritmo de busca em largura. Com isso, já temos dois algoritmos para avaliar conexidade em grafos não-dirigidos. A seguir, veremos uma terceira opção utilizando estruturas de dados para conjuntos disjuntos de vértices.

4.4 Conexidade por fusão de vértices

A avaliação de conexidade pode ser feita pelo algoritmo apresentado por Goodman e Hedetniemi ([GOODMAN; HEDETNIEMI, 1977](#)). A ideia básica do algoritmo é reduzir cada componente conexa a um único vértice por meio da operação de fusão de vértices¹. Dados dois vértices adjacentes u e v a fusão é realizada pela remoção da(s) aresta(s) (u, v) e dos vértices u e v . É criado um novo vértice w que deve ser adjacente a todos os demais vértices inicialmente adjacentes a u e v . A [Figura 4.9](#) ilustra esta operação.

¹ Esta operação também é conhecida como *contração de aresta*.

Figura 4.9 – Operação de fusão de vértices



Fonte: o autor.

O algoritmo seleciona um vértice arbitrário e vai “colapsando” o grafo com sucessivas fusões de vértice enquanto houver vértice adjacente disponível para a operação. A partir daí, contabiliza-se uma componente conexa e o procedimento é iniciado novamente a partir de um novo vértice arbitrário, se houver. O pseudocódigo apresentado tem os seguintes elementos básicos:

- w : vértice arbitrário;
- H : cópia do grafo original G . Isto é necessário para preservar o grafo original, porque supõe-se que as operações de fusão destroem o grafo;
- N_{cc} : número de componentes conexas do grafo.

Algoritmo: ConexidadeFusao(G)

```

// Inicialização
 $H \leftarrow G$ 
 $N_{cc} \leftarrow 0$ 

// Fusão sucessiva de vértices
enquanto  $H \neq \emptyset$  faça
    Selecione um vértice  $w \in H$ 
    enquanto  $w$  for adjacente a algum vértice  $v \in H$  faça
         $w \leftarrow \text{fusao}(w, v)$ 
    // Remova o vértice  $w$  do grafo
     $H \leftarrow H - w$ 
    // Contabilize uma componente conexa
     $N_{cc} \leftarrow N_{cc} + 1$ 

    // Avaliação da conexidade
    se  $N_{cc} > 1$  então
        | imprima("Grafo não conexo, com  $N_{cc}$  componentes conexas")
    senão
        | imprima("Grafo conexo")

```

Algoritmo 4.2: Avaliação de conexidade por fusão de vértices

4.5 Conexidade por conjuntos disjuntos

Em [Cormen et al. \(2001\)](#), é apresentada uma solução para avaliação de conexidade baseada em estruturas de dados de conjuntos disjuntos de vértices, considerando que eles são representados por objetos em linguagem de programação. Conjuntos disjuntos são aqueles que não compartilham elementos, isto é, dados k conjuntos A_1, A_2, \dots, A_k tem se que a intersecção entre todos eles é um conjunto vazio: $A_1 \cap A_2 \cap \dots \cap A_k = \emptyset$.

Há diferentes formas de implementar este tipo de estrutura de dados, uma delas é usando listas encadeadas para representar os conjuntos. Entretanto, vamos apresentar uma implementação que armazena conjuntos em árvores utilizando referências para nós predecessores, de forma semelhante ao que fizemos com vetores de roteamento no [Capítulo 3](#). Cada árvore representa um conjunto e cada nó representa um elemento do respectivo conjunto. Para utilizar a estrutura na avaliação de conexidade, é necessário implementar as seguintes operações:

- *CriaConjunto(v)*: esta operação deve criar um conjunto com o vértice v e atribuir um identificador único para este conjunto, que pode ser uma referência para o próprio objeto vértice, pois sabemos que os vértices são únicos e v não estará em nenhum outro conjunto;
- *BuscaConjunto(v)*: retorna uma referência para o identificador do conjunto que contém o vértice v ;
- *Uniao(u, v)*: cria um novo conjunto por meio da união dos conjuntos que contém os vértices u e v , e atribui um identificador ao conjunto resultante.

Se estas operações forem implementadas com listas encadeadas para armazenar os conjuntos, a operação *CriaConjunto(v)* pode ser implementada em tempo $O(1)$, porém a operação *BuscaConjunto(v)* leva tempo $O(n)$ (sendo n a quantidade de vértices do grafo), visto que precisa percorrer linearmente as listas de cada conjunto até encontrar o vértice v e retornar o identificador do conjunto correspondente. A união de duas listas de vértices *Uniao(u, v)* pode ser feita em tempo constante, porém, como esta operação precisa chamar *BuscaConjunto(v)* duas vezes, seu desempenho acaba sendo também linear na quantidade de vértices do grafo: $O(n)$. Para este tipo de implementação, cada vértice v possui dois atributos:

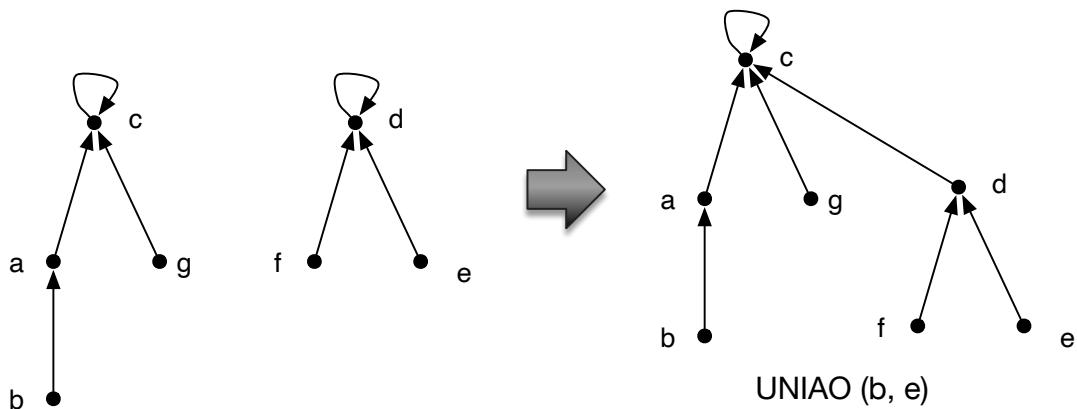
- $\rho(v)$: referência para o vértice pai de v na árvore que armazena um conjunto de vértices;
- $peso(v)$: o peso do vértice é um atributo numérico referente ao nível do vértice na árvore. Vértices folha possuem peso 0, e o vértice raiz vai ter como valor de peso a própria profundidade da árvore.

Em relação a uma implementação com listas encadeadas, a estrutura usando árvores se torna mais eficiente. A operação $CriaConjunto(v)$ leva tempo constante $O(1)$ ao criar um conjunto atribuindo v a seu próprio predecessor.

A operação $BuscaConjunto(v)$, por sua vez, precisa “subir” a árvore a partir de v em direção ao nó raiz e retornar a sua referência. Esta operação é linear na profundidade da árvore. Como veremos a seguir, o pseudocódigo contém uma heurística para compressão das árvores, reduzindo sua profundidade a cada operação de união de conjuntos. Isto é feito por meio do atributo *peso* dos vértices das duas árvores a serem unidas.

A operação $Uniao(u, v)$ pode ser feita atribuindo-se como predecessor de um vértice raiz o outro vértice raiz, conforme ilustrado na Figura 4.10. Aqui, as operações $BuscaConjunto(b)$ e $BuscaConjunto(e)$, retornam, respectivamente, referências para c e d , enquanto a operação $Uniao(b, e)$ forma um novo conjunto ao atribuir c como predecessor de d .

Figura 4.10 – Exemplo conjuntos disjuntos com árvores



Fonte: o autor.

A seguir, temos os pseudocódigos destas operações. A operação $CriaConjunto(v)$ forma um conjunto com apenas um objeto vértice simplesmente inicializando os seus atributos $\rho(v)$ e $peso(v)$. Ao aplicarmos esta operação para os n vértices de um grafo, criamos uma **floresta de árvores de conjuntos disjuntos**.

Algoritmo: $CriaConjunto(v)$

```

 $\rho(v) \leftarrow v$ 
 $peso(v) \leftarrow 0$ 

```

Algoritmo 4.3: Operação $CriaConjunto(v)$

A operação $BuscaConjunto(v)$ é um procedimento recursivo que inicia do vértice v e gradativamente vai subindo os nós até chegar ao nó raiz, o qual é retornado. Note

que, ao atribuir o retorno da chamada recursiva ao predecessor do vértice, o método promove uma adaptação da árvore, tornando-a menos profunda. Portanto, quanto mais este procedimento for chamado, mais otimizada fica a estrutura para executar também a união de dois conjuntos.

Algoritmo: *BuscaConjunto*(v)

```

se  $v \neq \rho(v)$  então
     $\rho(v) \leftarrow \text{BuscaConjunto}(\rho(v))$ 
retorna  $\rho(v)$ 
```

Algoritmo 4.4: Operação *BuscaConjunto*(v)

A operação de união usa um procedimento auxiliar *Link*(u, v) para efetivamente conectar as duas árvores. Se as árvores forem de profundidades diferentes, a árvore resultante menos profunda será aquela que mantiver como raiz o nó raiz da árvore original mais profunda. Esta heurística é implementada pelo procedimento *Link*(u, v).

Algoritmo: *Uniao*(u, v)

```
Link(BuscaConjunto( $u$ ), BuscaConjunto( $v$ ))
```

Algoritmo 4.5: Operação *Uniao*(u, v)

O procedimento *Link*(u, v) recebe a raiz de duas árvores e escolhe qual das duas vai se manter como raiz da árvore resultante. Isto é feito por meio da observação do atributo *peso* de cada um dos vértices raiz. A raiz com menor peso deve se tornar filha da raiz com peso original maior.

Algoritmo: *Link*(u, v)

```

se peso( $u$ ) > peso( $v$ ) então
     $\rho(v) \leftarrow u$ 
senão
     $\rho(u) \leftarrow v$ 
    se peso( $u$ ) = peso( $v$ ) então
         $\rho(v) \leftarrow u$ 
```

Algoritmo 4.6: Operação *Link*(u, v)

A utilização da estrutura de conjuntos disjuntos de vértices permita que utilizemos o algoritmo a seguir para construir uma floresta, na qual cada árvore representa uma componente conexa do grafo. O procedimento começa com n conjuntos de um vértice e vai verificando aresta por aresta para promover uniões dos conjuntos caso os dois vértices incidentes a uma aresta estejam em conjuntos diferentes.

Algoritmo: *ComponentesConexasConjuntos(G)*

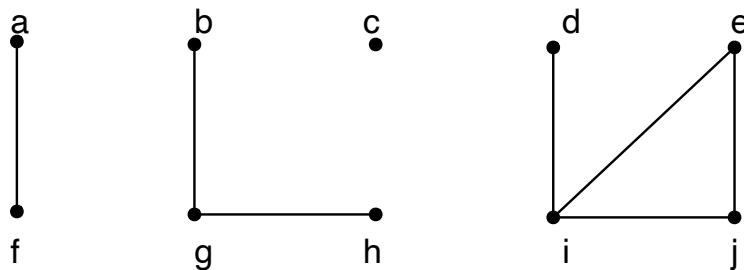
```

para cada vértice  $v \in V$  faça
    | CriaConjunto(v)
para cada aresta  $(u, v) \in E$  faça
    | | se BuscaConjunto(u)  $\neq$  BuscaConjunto(v) então
        | | | Uniao(u, v)
    
```

Algoritmo 4.7: Algoritmo *ComponentesConexasConjuntos(G)*

A [Figura 4.11](#) ilustra passo a passo a configuração dos conjuntos de vértices à medida que o algoritmo é executado no grafo mostrado.

Figura 4.11 – Exemplo de formação de conjuntos disjuntos



aresta	Coleção de conjuntos disjuntos									
inicio	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(e, j)	{a}	{b}	{c}	{d}	{e, j}	{f}	{g}	{h}	{i}	
(h, g)	{a}	{b}	{c}	{d}	{e, j}	{f}		{h, g}	{i}	
(f, a)	{a, f}	{b}	{c}	{d}	{e, j}			{h, g}	{i}	
(d, i)	{a, f}	{b}	{c}	{d, i}	{e, j}			{h, g}		
(b, g)	{a, f}		{c}	{d, i}	{e, j}			{h, g, b}		
(i, j)	{a, f}		{c}		{e, j, d, i}			{h, g, b}		
(e, i)	{a, f}		{c}		{e, j, d, i}			{h, g, b}		

Fonte: o autor.

Após a execução do algoritmo, pode-se investigar a configuração das componentes conexas por meio da estrutura de conjuntos. O procedimento *MesmaComponente(u, v)* ilustra uma forma de se consultar a estrutura.

Algoritmo: *MesmaComponente*(u, v)

```

se BuscaConjunto( $u$ ) = BuscaConjunto( $v$ ) então
    | retorna Verdadeiro
senão
    | retorna Falso
```

Algoritmo 4.8: Algoritmo *MesmaComponente*(u, v)

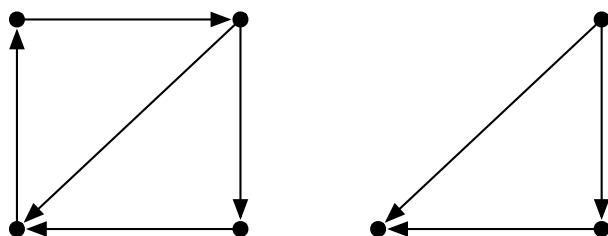
4.6 Conexidade em grafos dirigidos

Em grafos dirigidos, o conceito de conexidade também está relacionado à existência de caminhos entre vértices. Por exemplo, em termos gerais avaliaremos questões tais como se há “livre circulação” em determinadas regiões do grafo dirigido. Ou seja, se sairmos de um determinado vértice u e atingirmos um vértice v , é possível retornar a u , considerando os sentidos das arestas do grafo? Para tal avaliação, usaremos as definições a seguir, para a construção de um algoritmo baseado na Busca em Profundidade,

Definição 4.15 (Componente fortemente conexa). *Uma componente fortemente conexa de um digrafo $G = (V, E)$ é um conjunto maximal de vértices $C \subseteq V$, tal que, para cada par de vértices u e v em C , existe caminho do vértice u para o vértice v e vice-versa (de v para u).*

Definição 4.16 (Grafo fortemente conexo). *Um grafo dirigido é fortemente conexo se existir um caminho direto de cada vértice para cada vértice. A Figura 4.12 mostra um grafo fortemente conexo e um fracamente conexo, respectivamente.*

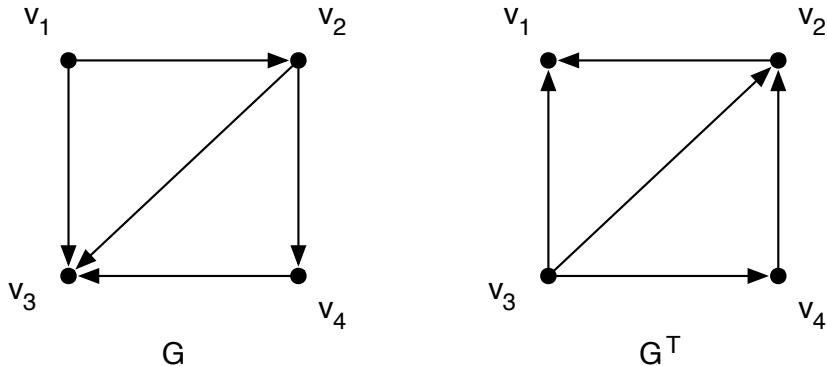
Figura 4.12 – Exemplo de grafos dirigidos forte e fracamente conexos



Fonte: o autor.

Definição 4.17 (Grafo transposto). *Dado um grafo dirigido $G = (V, E)$, seu grafo transposto é definido por $G^T = (V, E^T)$, onde $E^T = \{(u, v) : (v, u) \in E\}$. Em outras palavras, para encontrar um grafo transposto de um determinado grafo dirigido, basta que se inverta o sentido de todas as suas arestas. A Figura 4.13 mostra um grafo dirigido G e seu grafo transposto G^T .*

Figura 4.13 – Exemplo de um grafo dirigido e seu grafo transposto



Fonte: o autor.

Utilizando a busca em profundidade, o algoritmo a seguir, adaptado de [Aho, Hopcroft e Ullman \(1983\)](#), encontra as componentes fortemente conexas de grafos dirigidos:

Entrada: um grafo dirigido G .

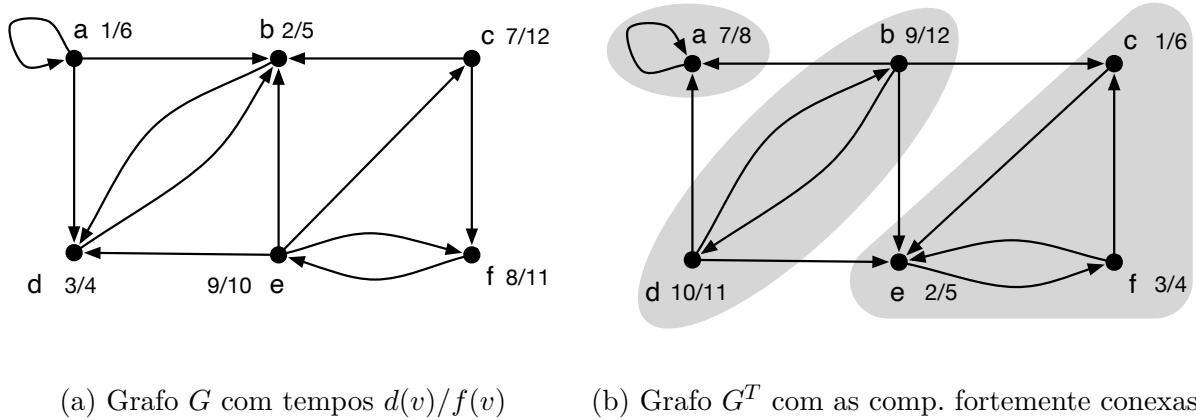
Passos:

1. Execute a busca em profundidade no grafo $BuscaProfTodos(G)$, calculando o tempo de fechamento $f(v)$ para cada um dos seus vértices;
2. Construa o grafo transposto G^T ;
3. Execute a busca em profundidade no grafo transposto $BuscaProfTodos(G^T)$, porém eu seu laço principal chame $VisitaVertice(v)$ considerando a ordem decrescente de tempos de encerramento $t_e(v)$ calculados no passo 1.
4. Os vértices de cada uma das árvores de busca geradas no passo anterior formam cada uma das componentes fortemente conexas do grafo original G .

Saída: floresta de árvores de busca, em que cada árvore é uma componente fortemente conexa.

A [Figura 4.14a](#) mostra o resultado de uma busca em profundidade em um grafo dirigido G com os tempos de abertura e fechamento mostrados próximos aos vértices. A [Figura 4.14b](#) mostra o grafo transposto G^T juntamente com as três componentes fortemente conexas agrupadas pelas regiões cinza.

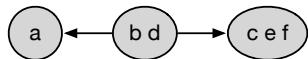
Figura 4.14 – Obtenção de componentes fortemente conexas



Fonte: o autor.

Um resultado adicional que podemos obter é o grafo acíclico dirigido de componentes fortemente conexas mostrado na Figura 4.15. Este grafo é obtido pela contração de todas as arestas dentro de cada componente fortemente conexa de G de forma que acaba restando apenas um único vértice para cada componente fortemente conexa.

Figura 4.15 – grafo acíclico dirigido de componentes fortemente conexas



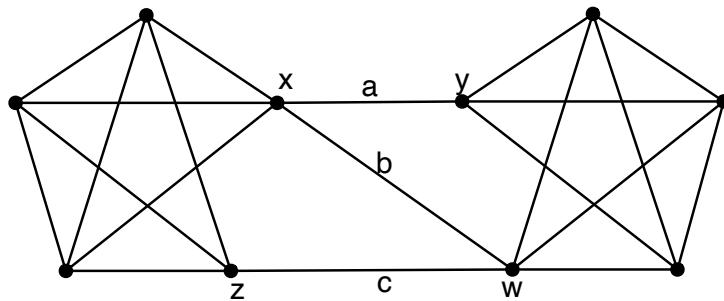
Fonte: o autor.

4.7 Exercícios

1. (PosComp – 2002) O menor número de arestas em um grafo conexo com n vértices é:

- a) 1
- b) $n/2$
- c) $n - 1$
- d) n
- e) n^2

2. Dado o grafo abaixo, encontre dois cortes de vértices com dois vértices cada (Obs.: rotule o grafo, caso seja necessário).



Fonte: o autor.

3. Para cada um dos itens abaixo, desenhe um grafo que atenda às especificações, ou então explique porque tal grafo não existe.

- a) Um grafo G com 6 vértices tal que $\kappa(G) = 2$ e $\gamma(G) = 2$;
- b) Um grafo conexo com 11 vértices, 10 arestas e sem vértice de corte;
- c) Um grafo 3-conexo com exatamente uma ponte;
- d) Um grafo 2-conexo com 8 vértices e exatamente duas pontes;

4. Determine $\kappa(K_{r,s})$ e $\gamma(K_{r,s})$, sendo $K_{r,s}$ um grafo bipartido completo.

5. Sendo $\delta(G)$ o grau do vértice de menor grau de um grafo G , mostre um exemplo de grafo G , quando possível, tal que:

- a) $\kappa(G) = 2$, e $\gamma(G) = 3$ e $\delta(G) = 4$;
- b) $\kappa(G) = 3$, e $\gamma(G) = 2$ e $\delta(G) = 4$;
- c) $\kappa(G) = 2$, e $\gamma(G) = 2$ e $\delta(G) = 4$.

6. No grafo de Petersen, encontre:

- a) um corte de arestas com 3 arestas;
- b) um corte de arestas com 4 arestas;
- c) um corte de arestas com 5 arestas;
- d) um corte de arestas com 6 arestas.

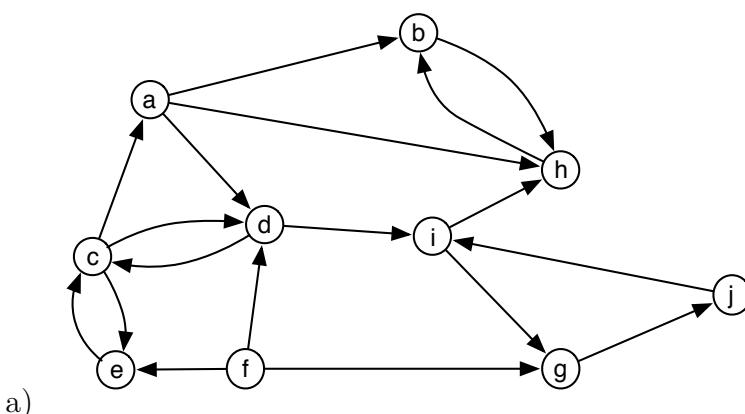
7. (PosComp – 2008) Denomina-se complemento de um grafo $G(V, E)$ o grafo H que tem o conjunto de vértices igual ao de G e tal que, para todo par de vértices distintos v, w em V , temos que a aresta (v, w) é aresta de G se e somente se (v, w) não é aresta de H .

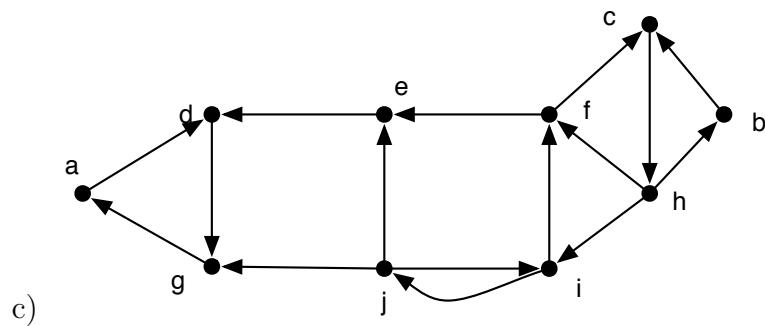
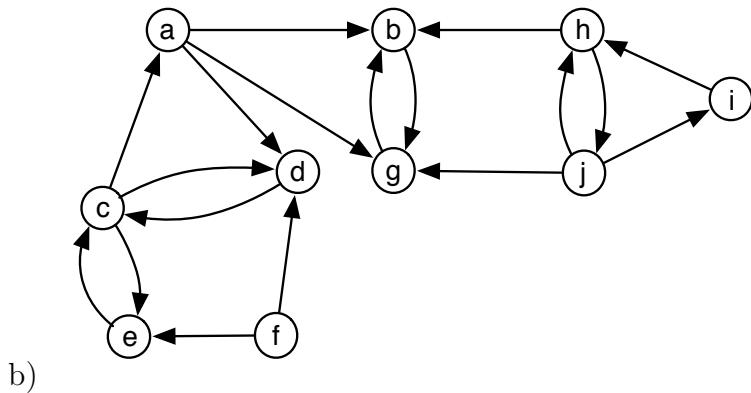
A esse respeito, assinale a afirmativa CORRETA.

- a) G e H são grafos isomorfos.
 - b) Se o grafo G é conexo, então H é conexo.
 - c) Se o grafo G não é conexo, então H é conexo.
 - d) Se o grafo G não é conexo, então H não é conexo.
 - e) Os grafos G e H têm o mesmo número de componentes conexas.
8. Dada a seguinte matriz de adjacência, encontre as componentes fortemente conexas do respectivo dígrafo, utilizando o algoritmo baseado em busca em profundidade.

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

9. Dados os dígrafos (a), (b) e (c) abaixo, encontre suas componentes fortemente conexas, mostrando passo a passo o desenvolvimento da solução do problema. Sugestão: utilizar o algoritmo baseado na geração de árvores de busca em profundidade e anotar os tempos de abertura e fechamento próximos aos vértices.





10. Se uma nova aresta for inserida em um dígrafo, como podem mudar as suas componentes fortemente conexas?
11. Desenhe um grafo conexo com 8 vértices com a menor quantidade possível de arestas.
12. Desenhe um grafo conexo com 7 vértices de tal forma que a remoção de qualquer uma de suas arestas resulta em um grafo não conexo.
13. Desenhe um grafo conexo com 5 vértices que permaneça conexo após a remoção de duas arestas quaisquer.

5 Grafos Eulerianos e Hamiltonianos

Neste capítulo vamos abordar dois tipos importantes de grafos, os Grafos Eulerianos e os Grafos Hamiltonianos. Tais grafos receberam seus nomes em homenagem a Leonard Euler e William Rowan Hamilton, respectivamente. Veremos importantes teoremas para descobrir um grafo é euleriano, bem como dois algoritmos para encontrar ciclos eulerianos, além de um estudo de caso relacionado ao jogo de dominó. Na sequência, este capítulo traz o Teorema de Ore, que define uma condição suficiente para que um grafo seja hamiltoniano e dois estudos de caso envolvendo este tipo de grafo.

5.1 Problema do Explorador e Problema do Turista

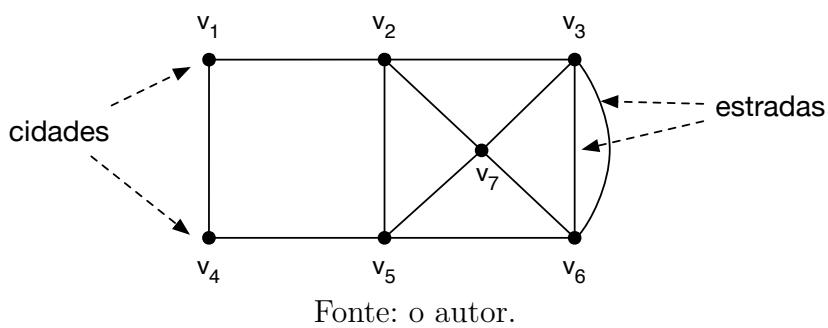
Esta seção aborda questões referentes ao caminhamento em grafos, considerando restrições em relação a arestas e vértices. Iniciamos apresentando definições intuitivas dos dois principais problemas básicos envolvendo grafos e ciclos eulerianos e hamiltonianos.

Definição 5.1 (Problema do explorador). *Um explorador quer encontrar um itinerário que passe por cada estrada exatamente uma vez e retorne ao ponto de partida.*

Definição 5.2 (Problema do turista). *Um turista quer encontrar um itinerário que visite cada cidade exatamente uma vez e retorne ao ponto de partida.*

Para ilustrar estes dois problemas, considere a [Figura 5.1](#), na qual o grafo representa um mapa, com os vértices sendo cidades, e as arestas sendo estradas ligando duas cidades. Partindo do vértice v_1 , uma solução para o problema do explorador poderia ser a seguinte sequencia: $v_1, v_2, v_3, v_6, v_3, v_7, v_6, v_5, v_7, v_2, v_5, v_4, v_1$. Já para o problema do turista, uma possível solução seria: $v_1, v_2, v_3, v_6, v_7, v_5, v_4, v_1$. O problema do explorador nos leva ao estudo dos Grafos Eulerianos, enquanto o problema do turista remete aos Grafos Hamiltonianos.

Figura 5.1 – Problemas do explorador e do turista



Fonte: o autor.

5.2 Grafos e Ciclos Eulerianos

Como o próprio título desta seção indica, agora vamos tratar de questões que têm ligação com o Problema das Pontes de Königsberg, o qual foi estudado por Leonard Euler no século XVIII ([seção 1.1](#)). Naquela ocasião, Euler tentava resolver (e provar) se era ou não possível resolver o problema de passar por todas as 7 pontes de Königsberg somente uma vez, retornando ao ponto de partida. Em seu famoso artigo ([EULER, 1741](#)), Euler demonstrou que a análise e solução deste tipo de problema dependia de uma propriedade que hoje chamamos de grau de vértices de um grafo. Em relação aos graus dos vértices, Euler enunciou os seguintes teoremas:

Teorema 5.1 (Lema do aperto de mão – *Handshaking Lemma*). *Para qualquer grafo G , a soma dos graus de todos os seus vértices é igual ao dobro do número de arestas. Simbolicamente, seja G um grafo de ordem n e tamanho m , então:*

$$\sum_{i=1}^n \text{grau}(v_i) = 2m$$

Prova. Ao somar os graus dos vértices de um grafo G , contamos duas vezes cada aresta de G , uma vez para cada um dos vértices incidentes à aresta.

Podemos dizer que os vértices são ímpares ou pares se seus graus são ímpares ou pares. A partir disso, o Teorema 5.1 traz a seguinte consequência, enunciada no Teorema 5.2 a seguir:

Teorema 5.2. *Todo grafo G tem um número par de vértices de grau ímpar.*

Prova. Seja G um grafo. Se G não contiver vértices ímpares então o resultado é trivial. Suponha que G tem k vértices de grau ímpar. Denotamos estes vértices por v_1, v_2, \dots, v_k . Se G também contém j vértices de grau par, nós os denotamos por u_1, u_2, \dots, u_j . Pelo teorema 5.1, temos:

$$[\text{grau}(v_1) + \text{grau}(v_2) + \dots + \text{grau}(v_k)] + [\text{grau}(u_1) + \text{grau}(u_2) + \dots + \text{grau}(u_j)] = 2m$$

em que m é o número de arestas de G . Como cada um dos números $\text{grau}(u_1), \text{grau}(u_2), \dots, \text{grau}(u_j)$ é par, a soma $[\text{grau}(u_1) + \text{grau}(u_2) + \dots + \text{grau}(u_j)]$ também é par. Portanto, temos que:

$$[\text{grau}(v_1) + \text{grau}(v_2) + \dots + \text{grau}(v_k)] = 2m - [\text{grau}(u_1) + \text{grau}(u_2) + \dots + \text{grau}(u_j)]$$

é par.

Por outro lado, cada um dos números $\text{grau}(v_1), \text{grau}(v_2), \dots, \text{grau}(v_k)$ é ímpar. Portanto, k necessariamente é par, ou seja, G tem um número par de vértices de grau ímpar.

Assim como nos grafos não dirigidos, podemos observar um fato interessante a respeito dos graus de entrada e de saída dos vértices, enunciado no Teorema 5.3:

Teorema 5.3. *Seja $G = (V, E)$ um grafo dirigido. Então:*

$$|E| = \sum_{i=1}^n \text{grau}_e(v_i) = \sum_{i=1}^n \text{grau}_s(v_i) = m$$

Prova. A primeira soma conta o número de arestas de entrada em todos os vértices e a segunda soma conta o número de arestas de saída em todos os vértices. Conclui-se que ambas as somas são iguais ao número de arestas do grafo.

Estes teoremas nos dão condições de, finalmente, definirmos o que são grafos e ciclos eulerianos.

Definição 5.3 (Ciclo e Grafo Euleriano). *Um grafo conexo é **euleriano** se contiver uma trilha¹ fechada que inclua todas as suas arestas. Tal trilha é chamada de **Ciclo Euleriano**.*

Dado um determinado grafo, como podemos descobrir se ele é euleriano? Esta era a questão central do problema das Pontes de Königsberg (embora naquele tempo ainda não existisse o conceito de grafo e muito menos de grafo euleriano). Euler analisou o problema da seguinte maneira: encontrar uma rota passando apenas uma vez em cada uma das pontes só é possível quando, ao atravessar uma ponte para chegar a uma parte da cidade, você deve ser capaz de deixar esta parte da cidade atravessando uma outra ponte. Tal condição leva à questão de graus de vértices, induzindo à definição de uma regra que nos indique quando um determinado grafo é euleriano, a qual conduz ao seguinte teorema:

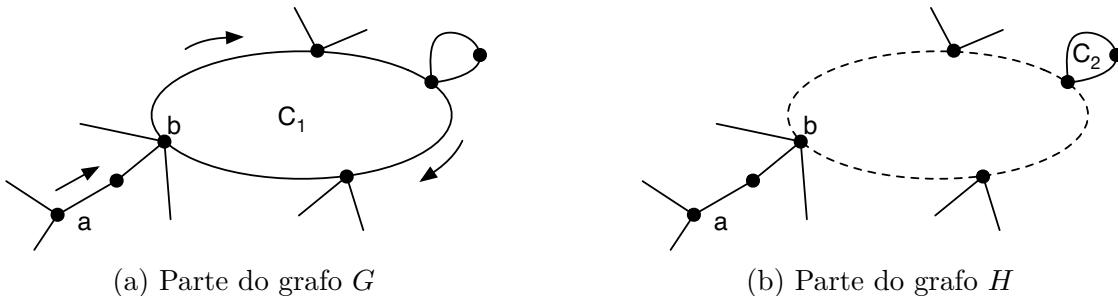
Teorema 5.4 (Ciclos Disjuntos). *Seja G um grafo no qual todos os seus vértices possuem grau par. Então G pode ser separado em ciclos disjuntos, ou seja, ciclos que não possuem arestas em comum.*

Aldous, Best e Wilson (2003) apresentam a seguinte prova para este teorema: seja $G = (V, E)$ um grafo conexo no qual todos os vértices têm grau par. Obtemos o primeiro ciclo disjunto partindo de um vértice arbitrário a e atravessando arestas de forma arbitrária, mas sem repetir arestas. Como todos os vértices têm grau par, sempre podemos chegar em um vértice por uma aresta e depois sair deste vértice por uma outra aresta. Como existe um número finito de vértices, eventualmente retornaremos a um vértice b já visitado anteriormente, formando o ciclo C_1 mostrado na Figura 5.2a.

Se removermos de G as arestas do ciclo C_1 , obtemos o grafo H , possivelmente não-conexo, mas com todos os vértices de grau par. Se H ainda tiver alguma aresta,

¹ Uma trilha é um percurso no qual todas as arestas, mas não necessariamente todos os vértices, são diferentes.

Figura 5.2 – Prova do teorema de ciclos disjuntos



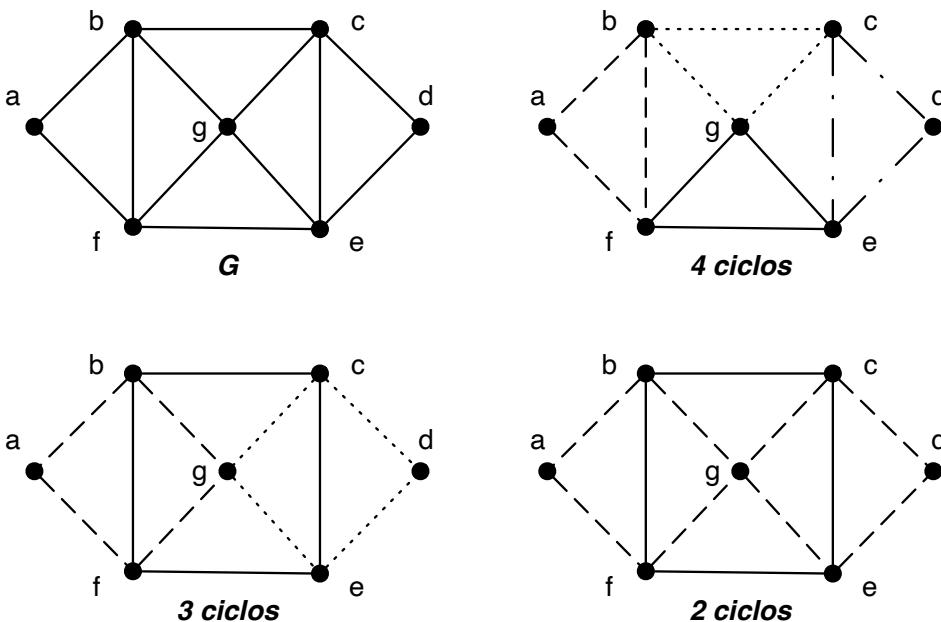
Fonte: o autor.

podemos repetir o procedimento anterior para encontrar o ciclo C_2 no grafo H , mostrado na [Figura 5.2b](#), que não tem arestas em comum com o ciclo C_1 .

A remoção das arestas de C_2 leva à obtenção de mais um novo grafo com todos os vértices de grau par, no qual podemos obter um novo ciclo C_3 . Continuamos este processo até que não haja mais arestas disponíveis e tenhamos obtido os ciclos C_1, C_2, \dots, C_k que, juntos, incluem todas as arestas de G , sem haver dois ciclos com arestas em comum.

Exemplo 5.1. Todos os vértices do grafo da [Figura 5.3](#) possuem grau par. Podemos mostrar graficamente que ele pode ser decomposto em 4, 3 e 2 ciclos disjuntos de arestas. Para facilitar a visualização, cada ciclo está representado por um tipo diferente de linha.

Figura 5.3 – Ciclos disjuntos de arestas



Fonte: o autor.

A partir dos conceitos apresentados, podemos enunciar o teorema de Euler que estabelece as condições para que um grafo seja euleriano.

Teorema 5.5 (Grafo Euleriano). *Um grafo conexo é Euleriano se e somente se todos os seus vértices tiverem grau par².*

Combinando os teoremas anteriores, podemos obter o seguinte teorema, que é a base para a implementação de um algoritmo para encontrar ciclos eulerianos em grafos.

Teorema 5.6 (Ciclos Eulerianos). *Um grafo euleriano pode ser decomposto em ciclos, de forma que nenhum par de ciclos tenha alguma aresta em comum.*

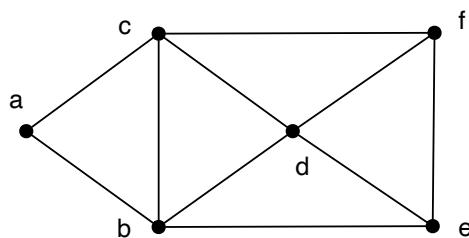
Antes de seguirmos para os algoritmos de determinação de ciclos eulerianos, vejamos o conceito de grafos semi-eulerianos.

Teorema 5.7 (Grafo Semi-euleriano). *Um grafo conexo é semi-euleriano se e somente se exatamente dois de seus vértices tiverem grau ímpar.*

Definição 5.4 (Caminho Semi-euleriano). *Um caminho semi-euleriano é uma trilha aberta que inclui todas as arestas do grafo.*

Exemplo 5.2. O grafo da [Figura 5.4](#) é semi-euleriano: é conexo e todos os seus vértices possuem grau par, exceto pelos vértices e e F , que possuem grau 3. A trilha $e, b, a, c, b, d, e, c, f, d, c, f$ é uma trilha euleriana (ou caminho euleriano) neste grafo.

Figura 5.4 – Grafo semi-euleriano



Fonte: o autor.

5.2.1 Algoritmo de Fleury

Em 1883, o matemático francês Pierre-Henry Fleury publicou um artigo no qual descrevia um procedimento extremamente intuitivo, embora ineficiente, para encontrar ciclos eulerianos e semi-eulerianos ([FLEURY, 1883](#)). O procedimento consiste em encontrar um percurso atravessando arestas indistintamente e removendo as arestas já atravessadas.

² A prova deste teorema pode ser vista em [Ore e Wilson \(1990\)](#).

A única restrição é que só se deve atravessar uma aresta do tipo “ponte” (veja a seção 4.2) somente quando não houver outra aresta disponível para ser atravessada. [Rabuske \(1992\)](#) apresenta o algoritmo da seguinte forma:

Escolha um vértice inicial arbitrário s e atravesse as arestas arbitrariamente de acordo com as seguintes regras:

- **Regra 1:** remova a aresta que foi percorrida, e se algum vértice ficar isolado, remova-o também;
- **Regra 2:** em cada passo, atravesse uma aresta do tipo ponte somente se não houver outra aresta disponível (excluindo-se o caso de resultar em um vértice isolado).

Embora seja de fácil entendimento, este algoritmo é ineficiente. Leva-se tempo $O(|V| + |E|)$ para determinar se uma aresta é ponte. Como este procedimento precisa ser feito no mínimo $|E|$ vezes, o algoritmo de Fleury acaba tendo complexidade $O(|E|^2)$. A seguir, veremos o algoritmo de Hierholzer que resolve o problema em tempo linear.

5.2.2 Algoritmo de Hierholzer

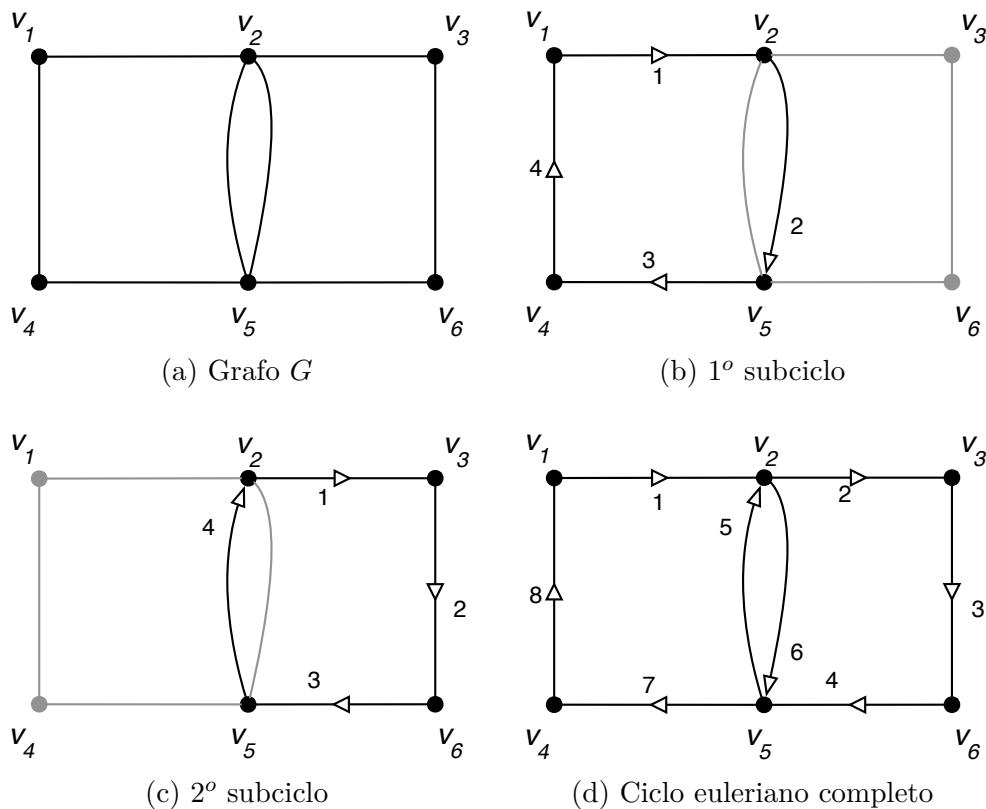
O algoritmo, a seguir, é inspirado no artigo de [Hierholzer e Wiener \(1873\)](#). Dado um grafo G euleriano, sabemos que é possível decompô-lo em ciclos disjuntos, os quais aqui chamaremos de *subciclos eulerianos*. O procedimento escolhe um vértice inicial arbitrário e vai percorrendo arestas até que retorne ao vértice inicial formando um subciclo euleriano. Como o grafo é euleriano, este procedimento obrigatoriamente fará com que se retorne ao vértice inicial. Havendo arestas não visitadas incidentes a este vértice inicial, determina-se um novo subciclo euleriano partindo-se de uma dessas arestas. Ao retornar ao vértice inicial, os dois subciclos são unidos formando um subciclo euleriano resultante. O procedimento é repetido vértice a vértice até que haja somente um único ciclo, que então configura um ciclo euleriano no grafo G .

Em resumo, dado um grafo euleriano G , o procedimento pode ser descrito da seguinte maneira:

1. Escolha um vértice inicial arbitrário u . Encontre um subciclo euleriano SC iniciando em u ;
2. Se algum vértice v em SC tiver arestas que não aparecem em SC , encontre um outro subciclo euleriano SC' que use duas dessas arestas;
3. Combine SC e SC' para formar um subciclo único SC^* ;
4. Repita os passos 2 e 3 até que todas as arestas de G tenham sido utilizadas;
5. Tendo utilizado todas as arestas do grafo, o “subciclo” euleriano SC^* é um ciclo euleriano em G .

Exemplo 5.3. A Figura 5.5 ilustra o procedimento de formação de ciclo euleriano a partir de subciclos eulerianos disjuntos. O grafo da Figura 5.5a é euleriano, pois é conexo e todos os seus vértices têm grau par. Iniciando arbitrariamente do vértice v_1 , podemos formar o primeiro subciclo euleriano destacado na Figura 5.5b. Os números próximos às arestas mostram a ordem de percorrido. Após completar este primeiro subciclo, restam arestas não utilizadas nos vértices v_2 e v_5 . Tomando-se, por exemplo, o vértice v_2 para iniciar um novo subciclo, temos o resultado mostrado na Figura 5.5c. Por fim, unindo-se os dois subciclos no vértice v_2 , encontra-se o “subciclo” mostrado na Figura 5.5d. Como este subciclo utiliza todas as arestas do grafo G , trata-se de um ciclo euleriano em G .

Figura 5.5 – Formação de ciclos disjuntos



Fonte: o autor.

Dado um grafo G euleriano, Kocay e Kreher (2004) apresentam o pseudo código do algoritmo 5.1 que implementa o procedimento de Hierholzer para determinação de ciclos eulerianos. Os subciclos eulerianos são armazenados em listas duplamente encadeadas de arestas. Quando o algoritmo começa a construir um subciclo a partir do vértice u , ele precisa conhecer uma aresta incidente a u pertencente ao ciclo euleriano, se existir algum. Ela é armazenada no atributo $EulerEdge[u]$, que permite que o algoritmo insira um novo subciclo euleriano no ciclo existente, na posição correta na lista duplamente encadeada de arestas.

Os elementos do pseudocódigo são os seguintes:

- V_{visit} : é uma coleção de vértices já visitados, ou seja, vértices nos quais já se iniciou a construção de subciclos eulerianos, caso seja possível;
- s : vértice inicial arbitrário;
- nV_{visit} : quantidade atual de vértices na coleção V_{visit} ;
- k : contador para manter controle da quantidade de vértices visitados (e encerrados);
- I_v : lista de arestas incidentes ao vértice v ;
- e_0, e_1, e_2, e_3 : referências temporárias para arestas;
- $prevEdge\langle e \rangle$: aresta anterior à aresta e na lista duplamente encadeada de arestas;
- $nextEdge\langle e \rangle$: aresta posterior à aresta e na lista duplamente encadeada de arestas;
- $EulerEdge\langle v \rangle$: “aresta de Euler” do vértice v , contém referência para uma aresta incidente a v , pertencente a um subciclo euleriano já construído.

O [algoritmo 5.1](#) é bastante eficiente. Para cada vértice v , todas as suas arestas incidentes são consideradas, e cada uma delas é ligada ao ciclo euleriano. Isto requer uma quantidade de passos igual ao grau de v ($grau(v)$). Vários subciclos em V podem ser conectados ao ciclo euleriano sendo construído. Existem, no máximo, $grau(v)/2$ subciclos em v .

Com isso, a complexidade do algoritmo é dada por:

$$\sum_{v \in G} grau(v) = 2|E| = O(E)$$

Portanto, o algoritmo de Hierholzer tem complexidade linear na quantidade de arestas do grafo.

Algoritmo: CicloEuleriano_Hierholzer(G)

```

// inicialização
 $V_{visit}[1] \leftarrow s;$ 
 $nV_{visit} \leftarrow 1;$ 
 $k \leftarrow 1;$ 

enquanto  $k \leq nV_{visit}$  faça
     $u \leftarrow V_{visit}[k];$ 

    enquanto  $I_u \neq null$  faça
         $e_0 \leftarrow I_u[1]$ ; // recebe a primeira aresta incidente a  $u$ 
         $v \leftarrow o$  outro vértice da aresta  $e_o$ ;
        Remova a aresta  $e_o$  do grafo;
         $e_1 \leftarrow e_0;$ 

        enquanto  $v \neq u$  faça
            se  $v \notin V_{visit}$  então
                 $nV_{visit} \leftarrow nV_{visit} + 1;$ 
                 $V_{visit}[nV_{visit}] \leftarrow v;$ 

                 $e_2 \leftarrow I_v[1]$ ; // recebe a primeira aresta incidente a  $v$ 
                 $nextEdge\langle e_1 \rangle \leftarrow e_2;$ 
                 $prevEdge\langle e_2 \rangle \leftarrow e_1;$ 

                se  $EulerEdge[v] = null$  então
                     $EulerEdge[v] \leftarrow e_2;$ 

                     $e_1 \leftarrow e_2;$ 
                     $v \leftarrow o$  outro vértice da aresta  $e_1$ ;
                    Remova a aresta  $e_1$  do grafo;

            // um subciclo em  $u$  acabou de ser completado
             $prevEdge\langle e_0 \rangle \leftarrow e_1;$ 
             $nextEdge\langle e_1 \rangle \leftarrow e_0;$ 

            se  $EulerEdge[u] = null$  então
                 $EulerEdge[u] \leftarrow e_0;$ 
            senão
                // insere um novo subciclo no subciclo preexistente em  $u$ 
                 $e_1 \leftarrow EulerEdge[u];$ 
                 $e_2 \leftarrow prevEdge\langle e_1 \rangle;$ 
                 $e_3 \leftarrow prevEdge\langle e_0 \rangle;$ 
                 $nextEdge\langle e_2 \rangle \leftarrow e_0;$ 
                 $nextEdge\langle e_3 \rangle \leftarrow e_1;$ 

         $k \leftarrow k + 1;$ 

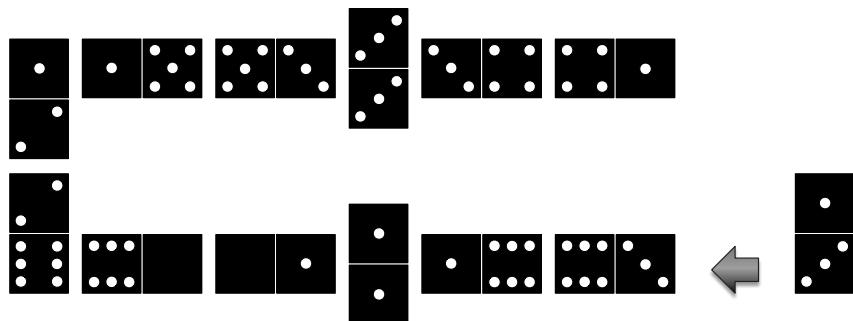
```

Algoritmo 5.1: CicloEuleriano_Hierholzer(G)

5.2.3 Estudo de caso: dominó

O dominó é um jogo de mesa formado por 28 peças (ou “pedras”, como dizem alguns de seus jogadores). As pedras são retangulares, divididas ao meio. Cada metade tem uma marcação indicando um valor de 0 a 6. Uma regra básica do jogo é que as pedras só podem ser “conectadas” por extremidades de mesmo valor. A [Figura 5.6](#) mostra algumas pedras de dominó, como tipicamente estariam posicionadas sobre a mesa de jogo. Aqui há um fato curioso: se um jogador colocar a pedra “0 e 3” na mesa, ligada ao resto do jogo pela extremidade 3, o jogo fica “trancado”. Isto é, fica impossível continuar o jogo porque as duas extremidades tem valor 1, e não há mais nenhuma peça com valor 1 fora da mesa (nas mãos de algum jogador). Além disso, neste caso fecha-se um ciclo de pedras na mesa, conectando-se as duas extremidades de valor 1.

Figura 5.6 – Exemplo de jogo de dominó



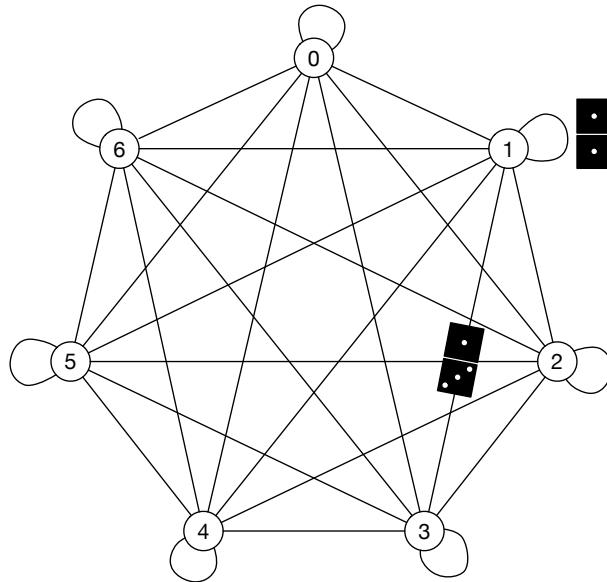
Fonte: o autor.

O jogo de dominó nos traz um problema interessante: *é possível construir um ciclo no jogo de dominó utilizando todas as 28 pedras?* Obviamente, poderíamos obter um conjunto de pedras de dominó e, fisicamente, tentar encontrar uma configuração que provasse a possibilidade de construir tal ciclo, caso isso seja possível. Note que, se não for possível, teríamos que provar essa impossibilidade, o que pode ser difícil usando o método de tentativa e erro com pedras físicas.

Uma alternativa para resolver nosso problema poderia ser por meio da modelagem do problema como um grafo. A partir daí, poderíamos demonstrar a existência (ou ausência) de determinada propriedade, o que levaria à resposta do problema. Ao modelar o problema, precisamos determinar quais seriam nossos conjuntos de vértices e arestas, para então construir e analisar o grafo resultante. Em geral, temos diferentes opções de modelagem, algumas facilitando a obtenção de respostas, outras, dificultando. É comum pensarmos nos vértices como objetos físicos, por exemplo, as pedras do jogo. Neste caso, as arestas poderiam representar a ligação de duas pedras na mesa. Note que essa modelagem não nos ajuda muito, pois ela é, praticamente, equivalente ao processo de tentativa e erro com pedras físicas.

Uma outra possibilidade de modelagem poderia ser considerar os vértices como valores nas extremidades das pedras e considerar as arestas ligando dois valores se eles estiverem presentes na mesma pedra. Ou seja, as pedras são representadas por arestas, e não por vértices. Assim, temos que $G = (V, E)$, $V = \{0, 1, 2, 3, 4, 5, 6\}$ e $E = \{(u, v) | (u, v) \text{ é um par não ordenado de valores existentes em uma pedra do jogo}\}$. A [Figura 5.7](#) mostra a representação gráfica deste grafo. Cada um dos vértices é adjacente aos demais por arestas que representam as pedras do jogo com extremidades diferentes, e a si próprio por laços representando as pedras com dois valores iguais. Como exemplos, temos, respectivamente, as arestas $(1, 3)$ e $(1, 1)$.

Figura 5.7 – Grafo do jogo de dominó

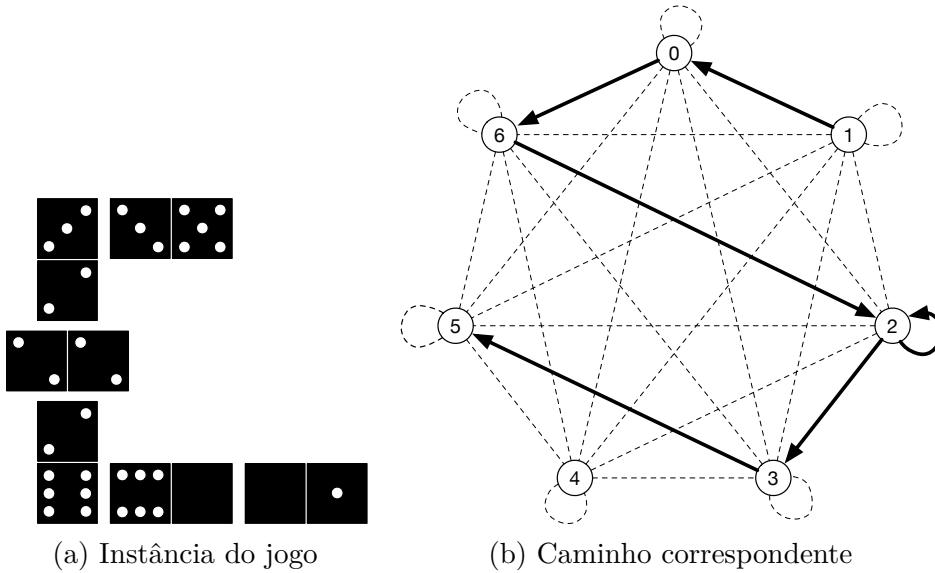


Fonte: o autor.

Para verificar a possibilidade de formar um ciclo no jogo com as 28 pedras, devemos considerar que as pedras devem ser colocadas na mesa conectando suas extremidades de valor igual. Ao analisarmos uma instância do jogo, como a mostrada na [Figura 5.8a](#), percebemos que o jogo é uma sequência de pedras na qual o valor final de uma pedra é igual ao valor inicial da próxima pedra na mesa. Isso corresponde a uma sequência de vértices e arestas no grafo, em que o vértice final de uma aresta na sequência é igual ao vértice inicial da próxima aresta na sequência. Este é o próprio conceito de caminho em um grafo. Portanto, uma instância do jogo pode ser representada por um caminho no grafo conforme mostrado na [Figura 5.8b](#).

Sendo assim, se pudermos encontrar um caminho fechado (para fechar o ciclo no jogo), que passe por todas as arestas do grafo somente uma vez (não existem pedras repetidas, logo não se pode passar mais de uma vez em uma aresta), então conseguiremos provar que é possível construir um ciclo fechado no jogo utilizando todas as 28 pedras. Em

Figura 5.8 – Modelagem de instância do jogo de dominó



Fonte: o autor.

outras palavras, basta verificarmos a propriedade do grafo ser euleriano. Isso é facilmente constatado na [Figura 5.7](#), pois o grafo é conexo e todos os seus vértices têm grau par (grau 8), demonstrando que é possível construir o ciclo no jogo. Caso se queira obter explicitamente uma instância do jogo fechado com todas as 28 pedras, basta executar o [algoritmo 5.1](#) para obter um ciclo euleriano no grafo da [Figura 5.7](#).

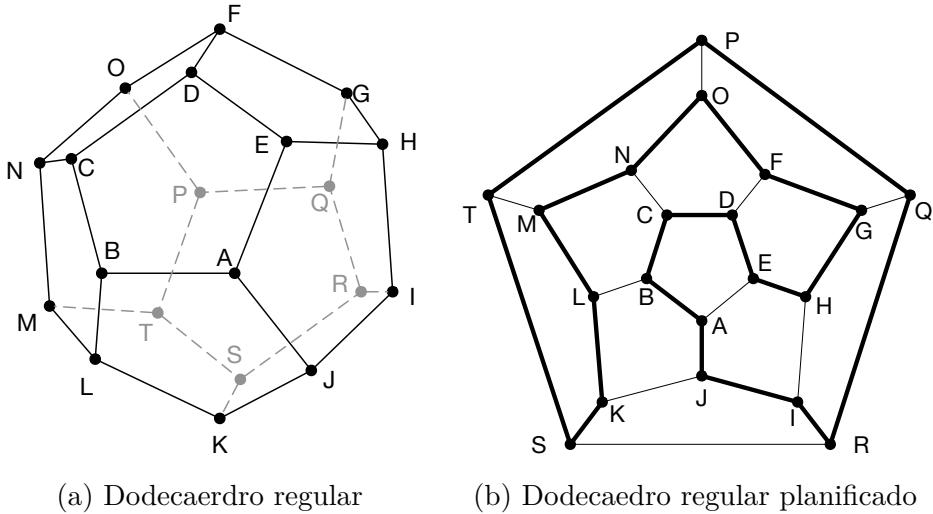
5.3 Grafos e Ciclos Hamiltonianos

Os Grafos e Ciclos Hamiltonianos receberam este nome em homenagem ao matemático e astrônomo irlandês William Rowan Hamilton. Em 1857 ele inventou um jogo, chamado de “Jogo Icosiano” que consistia em encontrar o que hoje chamamos de ciclo hamiltoniano em um dodecaedro regular. Tratava-se de um dodecaedro de madeira que tinha buracos em seus vértices. O jogador deveria inserir pinos ligados com barbantes para encontrar um caminho fechado que passasse apenas uma vez em cada um dos vértices.

Definição 5.5 (Ciclo e Grafo Hamiltoniano). *Um grafo conexo é **hamiltoniano** se contiver um ciclo que inclua cada um de sus vértices apenas uma vez. Tal ciclo é chamada de **Ciclo Hamiltoniano**.*

A [Figura 5.9a](#) mostra um dodecaedro regular com seus vértices rotulados de A a T , enquanto a [Figura 5.9b](#) traz uma possível solução para o jogo icosiano, mostrada numa planificação do mesmo dodecaedro regular. Neste caso, a solução seria a seguinte sequência de vértices: $A, B, C, D, E, H, G, F, O, N, M, L, K, S, T, P, Q, R, I, J, A$. Tal sequencia é um ciclo hamiltoniano no grafo.

Figura 5.9 – O jogo icosiano



Fonte: o autor.

O problema de determinar se um grafo é hamiltoniano pode parecer, à primeira vista, similar ao problema de decidir se um determinado grau é euleriano. A princípio esperamos que haja alguma propriedade que nos dê de forma direta esta resposta tal qual àquelas que levaram à formulação dos teoremas para ciclos eulerianos. Entretanto, no caso de grafos hamiltonianos não há uma resposta fácil e direta.

Uma primeira abordagem para determinar se um grafo é hamiltoniano consiste em tentar classificá-lo em alguma categoria especial de grafo. Por exemplo, sabemos que o grafo completo K_5 é hamiltoniano, pois, havendo arestas entre todos os pares de vértices, sabemos que ao chegar em determinado vértice, sempre podemos ir diretamente a um outro vértice ainda não visitado, até completar o ciclo hamiltoniano.

Além disso, ao adicionarmos uma aresta a um grafo hamiltoniano, ele continua sendo hamiltoniano, pois o ciclo hamiltoniano continua existindo após a adição da aresta. Intuitivamente, isso nos traz a noção de que grafos mais densos (isto é, com mais arestas) provavelmente tem mais chance de serem hamiltonianos. Em 1960, o matemático norueguês Oysten Ore provou o seguinte teorema, relacionando graus de vértices à propriedade de um grafo ser hamiltoniano ([ORE, 1960](#)).

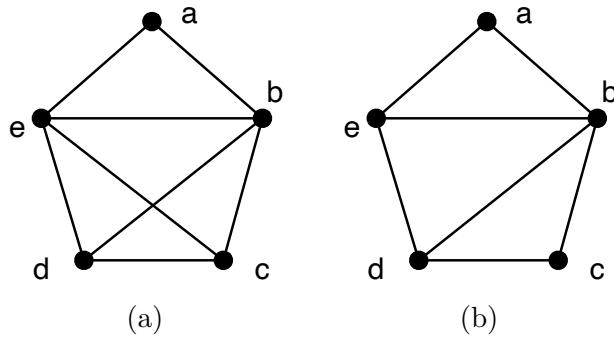
Teorema 5.8 (Teorema de Ore). *Seja G um grafo simples conexo com n vértices, em que $n \geq 3$, e $\text{grau}(u) + \text{grau}(v) \geq n$, para cada par de vértices não adjacentes u e v . Então G é hamiltoniano.*

Note que o teorema mostra uma condição suficiente, mas não necessária para que um grafo seja hamiltoniano. Isso quer dizer que, ao analisarmos um grafo à luz do teorema de Ore, pode ser que o referido grafo não atenda aos requisitos do teorema, e mesmo assim

seja hamiltoniano.

Exemplo 5.4. O grafo da [Figura 5.10a](#), tem $n = 5$ vértices e 2 pares de vértices não adjacentes. Aplicando o teorema de Ore, temos $\text{grau}(a) + \text{grau}(c) = 5 \geq n$ e $\text{grau}(a) + \text{grau}(d) = 5 \geq n$. Portanto, o grafo é hamiltoniano. No caso da [Figura 5.10b](#), o grafo não passa pelo teorema de Ore porque $\text{grau}(a) + \text{grau}(c) = 4 < n$. Por outro lado, mesmo sem cumprir os requisitos do teorema, o grafo é hamiltoniano porque podemos obter um ciclo hamiltoniano: a, b, c, d, e, a .

Figura 5.10 – Aplicação do teorema de Ore



Fonte: o autor.

Note que, havendo a necessidade de utilizar o teorema de Ore em um multigrafo, basta reduzi-lo a um grafo simples removendo todos os seus laços e arestas paralelas. Se o grafo simples for hamiltoniano, o multigrafo original também o será, pois a inclusão de novas arestas não altera esta propriedade do grafo.

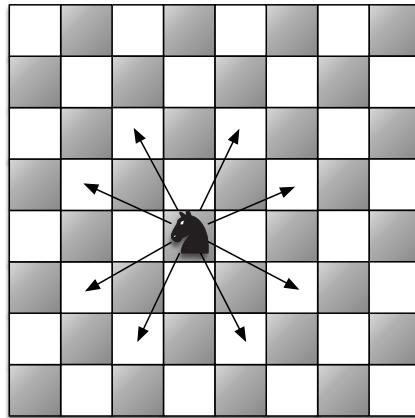
O problema de encontrar ciclos hamiltonianos pode ser simples para algumas classes especiais de grafos (grafos ciclo e grafos completos, por exemplo). Porém, no caso geral, trata-se de um problema extremamente difícil, cuja complexidade é não polinomial.

5.3.1 Estudo de caso: xadrez

O tabuleiro de xadrez é composto por uma matriz de 8×8 quadrados (ou “casas”) alternadas entre brancos e pretos. O formato de cada peça indica o tipo de movimentação permitida pela peça no tabuleiro. Por exemplo, a peça “torre” se movimenta de forma vertical e horizontal (em linhas e colunas). Já o “bispo” se movimenta em diagonal. O cavalo combina os dois tipos de movimento andando uma casa na horizontal ou vertical e mais uma na diagonal. A [Figura 5.11](#) mostra os possíveis movimentos do cavalo dentro do tabuleiro.

O “Problema do Percurso do Cavalo no Xadrez” tem sido estudado há centenas de anos, e consiste na seguinte questão: utilizando uma sequência de movimentos do cavalo, é

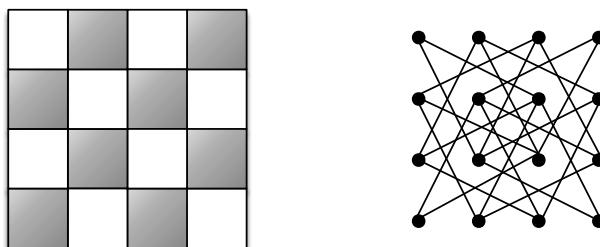
Figura 5.11 – Movimentos do cavalo no tabuleiro de xadrez



Fonte: o autor.

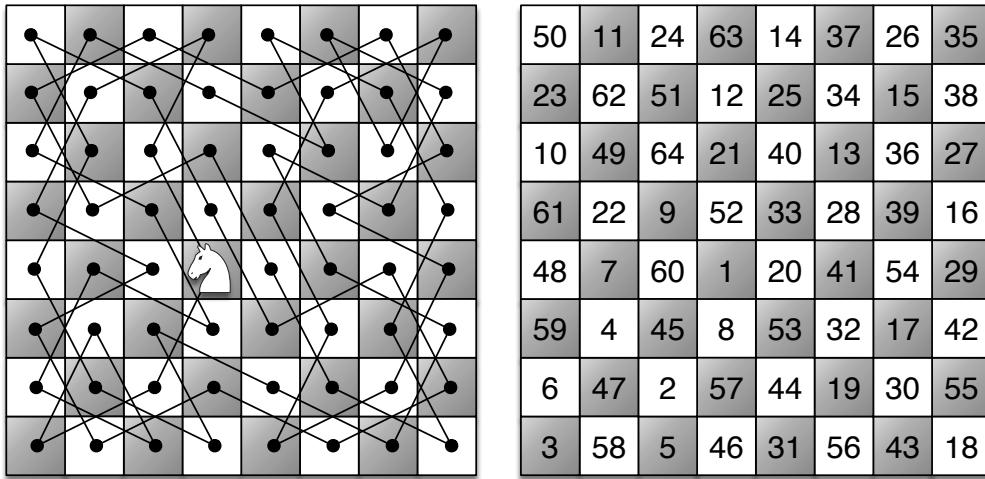
possível o cavalo visitar todas as casas do tabuleiro apenas uma vez, e retornando à casa na qual ele começou?

Para responder esta questão, podemos representando o tabuleiro por um grafo no qual cada vértice corresponde a uma casa do tabuleiro. Dois vértices são conectados por uma aresta caso seja possível o cavalo se movimentar diretamente entre as duas casas correspondentes. A [Figura 5.12](#) mostra um tabuleiro de 4×4 casas com o respectivo grafo resultante. É fácil deduzir que encontrar a solução do problema do percurso do cavalo no jogo de xadrez corresponde a encontrar um ciclo hamiltoniano no grafo.

Figura 5.12 – Modelagem do tabuleiro 4×4 Fonte: adaptado de [Chartrand \(1977\)](#).

No caso do tabuleiro de 8×8 casas, esta modelagem nos leva a um grafo com 64 vértices e 168 arestas que, de fato, contém vários ciclos hamiltonianos. A [Figura 5.13a](#) mostra um desses ciclos hamiltonianos. Esta solução é bastante peculiar, pois se escrevermos a ordem sequencial dos movimentos obtemos o diagrama da [Figura 5.13b](#), que é um quadrado mágico. Neste quadrado, a soma dos valores de todas as casas de qualquer linha ou coluna sempre resulta no mesmo valor, que é 260!

Figura 5.13 – Solução do problema do percurso do cavalo



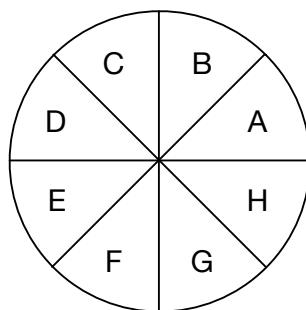
(a) Ciclo hamiltoniano

(b) Números sequenciais no ciclo

Fonte: adaptado de [Aldous, Best e Wilson \(2003\)](#).

5.3.2 Estudo de caso: *Gray codes*

Na engenharia, às vezes é necessário representar posição angular em uma haste girando continuamente por meio de um comutador. Um arranjo de escovas no comutador permite gerar uma palavra binária única para cada setor do círculo em seus 360° . Por exemplo, se precisarmos de 8 posições, temos setores de 45° cada, e necessitamos de palavras de 3 bits para representar cada um dos 8 setores. A [Figura 5.14](#) mostra os 8 setores de 45° no círculo, com as respectivas denominações de A a H.

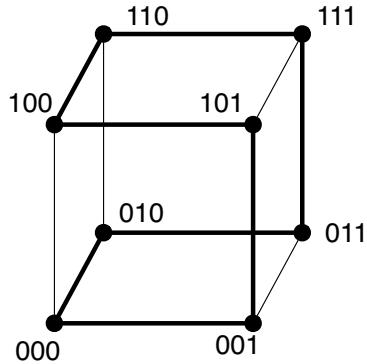
Figura 5.14 – Círculo dividido em 8 setores de 45° 

Fonte: o autor.

Para minimizar ambiguidades e erros leitura nas escovas, é conveniente que a seqüência de palavras binárias mude apenas 1 dígito de um dado setor para o setor vizinho. Portanto, a conversão convencional de base numérica decimal para binária não funciona. A seqüência de palavras binárias com essa propriedade é chamada de *Gray code*. Os *Gray codes* podem ser obtidos por meio de ciclos hamiltonianos em grafos cubo. Para *Gray*

codes de 3 dígitos, encontramos um ciclo hamiltoniano no grafo cubo Q_3 , mostrado na Figura 5.15, que gera o *Gray code* da tabela correspondente.

Figura 5.15 – Grafo Q_3 e Gray codes de 3 bits

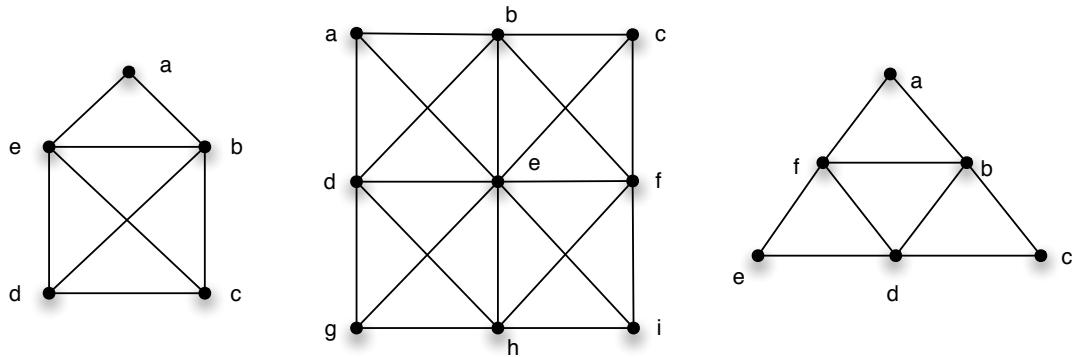


Setor	Código
A	000
B	001
C	101
D	100
E	110
F	111
G	011
H	010

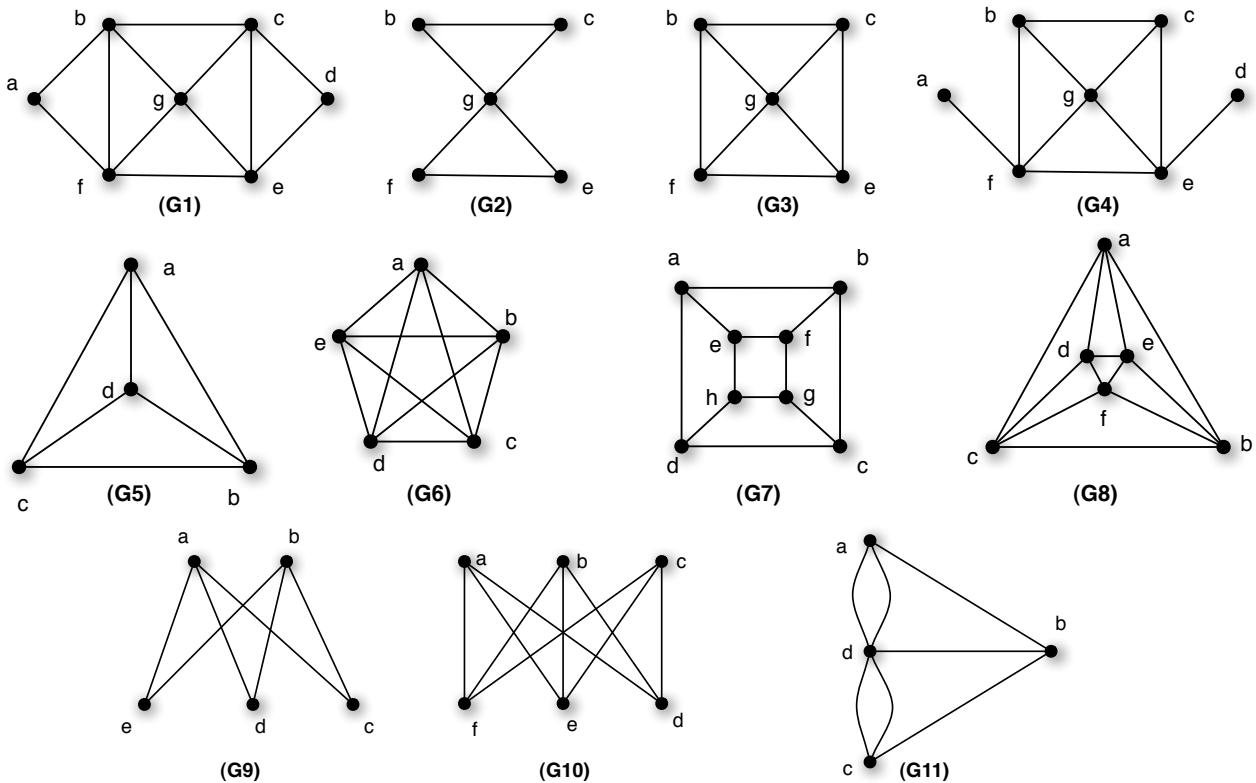
Fonte: o autor.

5.4 Exercícios

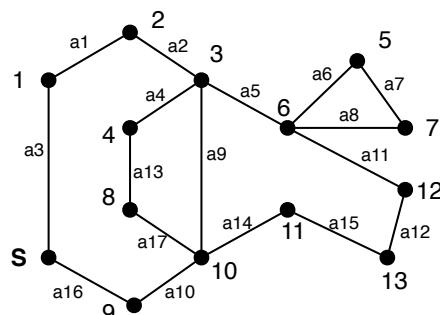
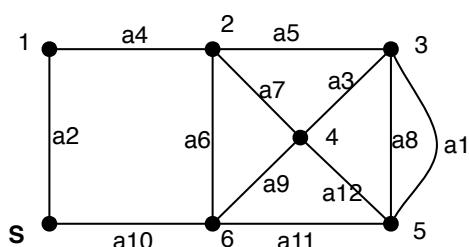
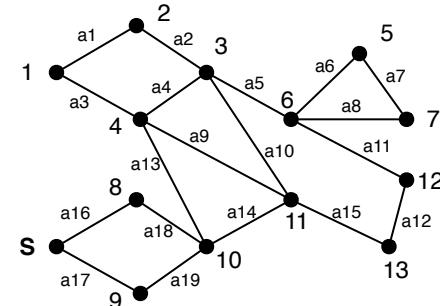
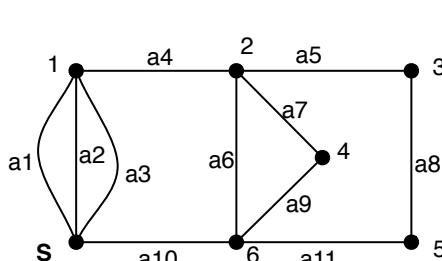
1. Verifique se os grafos abaixo são eulerianos e/ou hamiltonianos. Justifique sua resposta encontrando ciclos eulerianos e/ou hamiltonianos nos grafos.



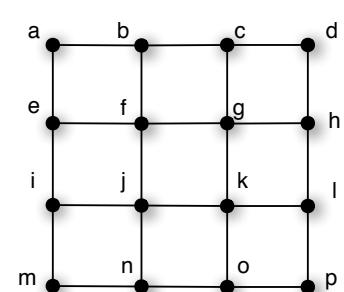
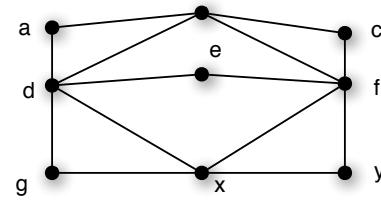
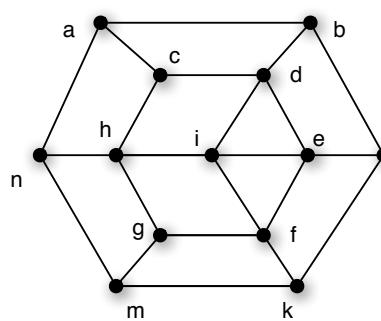
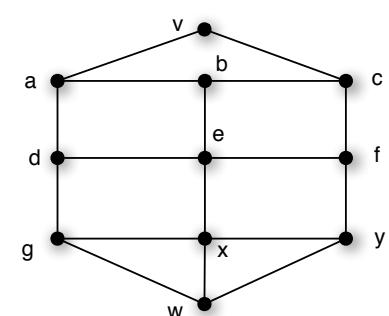
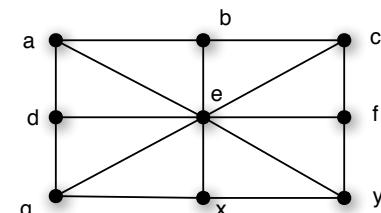
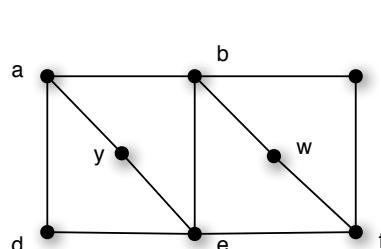
2. Para cada um dos grafos a seguir, verifique se é euleriano e/ou hamiltoniano e escreva um ciclo euleriano e um ciclo hamiltoniano quando possível. Verifique se cada um dos grafos satisfaz o Teorema de Ore.



3. Para cada um dos grafos a seguir, construa um ciclo euleriano. Inicie a construção do ciclo a partir do vértice s . Os demais vértices estão rotulados por números inteiros e as arestas por rótulos do tipo a_1 , a_2 etc.

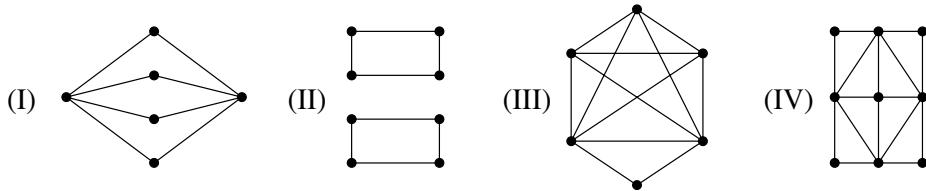


4. Para cada um dos grafos a seguir, tente determinar um ciclo hamiltoniano, ou então prove que o grafo não é hamiltoniano.



5. Prove que não é possível encontrar uma "volta do cavalo" num tabuleiro de xadrez de 3×6 casas (passar por todas as casas somente uma vez e retornar a casa inicial).

6. Para quais valores de n , r e s os grafos a seguir são eulerianos?
- o grafo completo K_n ;
 - o grafo completo bipartido $K_{r,s}$;
 - o grafo-cubo n -dimensional Q_n .
7. Para quais valores de n , r e s os grafos a seguir são hamiltonianos?
- o grafo completo K_n ;
 - o grafo completo bipartido $K_{r,s}$;
 - o grafo-cubo n -dimensional Q_n .
8. Desenhe dois grafos com 10 vértices e 13 arestas cada. O primeiro deve ser euleriano, mas não-hamiltoniano, e o segundo deve ser hamiltoniano, mas não-euleriano.
9. (PosComp – 2003) Quais dos quatro grafos abaixo são eulerianos?



- Somente I e II
 - Somente I
 - Somente II
 - Somente I, II e IV
 - Nenhum deles é Euleriano
10. (PosComp – 2005) Dadas as seguintes afirmações:

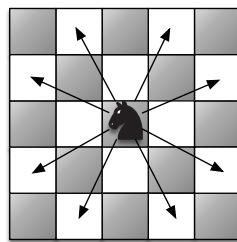
- I. Qualquer grafo conexo com n vértices deve ter pelo menos $n - 1$ arestas.
- II. O grafo bipartido completo $K_{m,n}$ é euleriano desde que m e n sejam ímpares.
- III. Em um grafo, o número de vértices de grau ímpar é sempre par.

São verdadeiras:

- Somente a afirmação I.
- Somente as afirmações I e III.
- Somente as afirmações II e III.
- Somente as afirmações I e II.
- Todas as afirmações.

11. (PosComp – 2005) Seja $T_{n,m}$ um tabuleiro de xadrez $n \times m$. Denominamos um *circuito equestre* em $T_{n,m}$ a um percurso de um cavalo, se movendo como num jogo de xadrez, que passa por cada uma das células de $T_{n,m}$ exatamente uma vez, e que começa e termina numa mesma célula (arbitrária). O número de circuitos equestrados em $T_{5,5}$ é:

Figura 5.16 – Exemplo de movimentos válidos de um cavalo.



- a) 0 b) 1 c) 5 d) 25 e) 5!

12. (PosComp – 2006) Seja $G = (V, E)$ um grafo simples conexo não-euleriano. Queremos construir um grafo H que seja euleriano e que contenha G como subgrafo. Considere os seguintes possíveis processos de construção:

 - I. Acrescenta-se um novo vértice, ligando-o a cada vértice de G por uma aresta.
 - II. Acrescenta-se um novo vértice, ligando-o a cada vértice de grau ímpar de G por uma aresta.
 - III. Cria-se uma cópia G' do grafo G e acrescenta-se uma aresta ligando cada par de vértices correspondentes.
 - IV. Escolhe-se um vértice arbitrário de G e acrescentam-se arestas ligando este vértice a todo vértice de grau ímpar de G .
 - V. Duplicam-se todas as arestas de G .
 - VI. Acrescentam-se arestas a G até se formar o grafo completo com $|V|$ vértices.

Quais dos processos acima sempre constroem corretamente o grafo H?

- a) Somente II e IV
 - b) Somente II, IV e V
 - c) Somente III, V e VI
 - d) Somente II, IV, V e VI
 - e) Somente I, III, IV e V

13. (PosComp – 2007) Seja $G = (V, E)$ um grafo simples e finito, em que $|V| = n$ e $|E| = m$. Nesse caso, analise as seguintes alternativas.

- I. Se G é hamiltoniano, então G é 2-conexo em vértices.
- II. Se G é completo, então G é hamiltoniano.
- III. Se G é 4-regular e conexo, então G é euleriano.
- IV. Se G é bipartido com partições A e B , então G é hamiltoniano se, e somente se, $|A| = |B|$.
- V. Se G é euleriano, então G é 2-conexo.

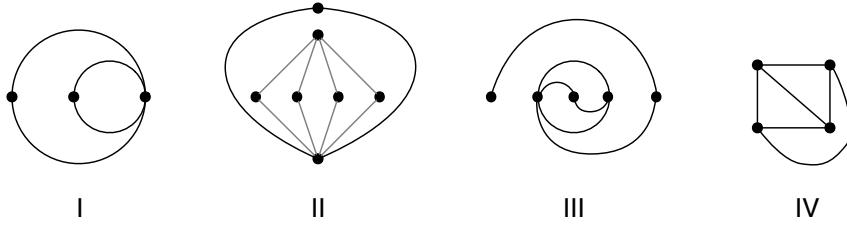
A análise permite concluir que são FALSOS:

- a) apenas os itens I e II.
- b) apenas os itens I e V.
- c) apenas os itens II e III.
- d) apenas os itens III e IV.
- e) apenas os itens IV e V.

14. (PosComp – 2008) Em um grafo $G(V, E)$, o grau de um vértice v é o número de vértices adjacentes a v . A esse respeito, assinale a afirmativa **CORRETA**.

- a) Num grafo, o número de vértices com grau ímpar é sempre par.
- b) Num grafo, o número de vértices com grau par é sempre ímpar.
- c) Num grafo, sempre existe algum vértice com grau par.
- d) Num grafo, sempre existe algum vértice com grau ímpar.
- e) Num grafo, o número de vértices com grau ímpar é sempre igual ao número de vértices com grau par.

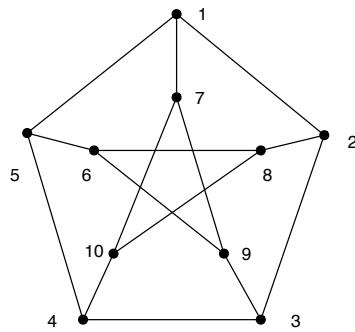
15. (PosComp – 2013) Considere as figuras representadas a seguir.



Assinale a alternativa correta.

- a) Somente os grafos I e II admitem caminho euleriano.
- b) Somente os grafos I e IV admitem caminho euleriano.
- c) Somente os grafos III e IV admitem caminho euleriano.
- d) Somente os grafos I, II e III admitem caminho euleriano.
- e) Somente os grafos II, III e IV admitem caminho euleriano.

16. (PosComp – 2011) Sejam 10 cidades conectadas por rodovias, conforme o grafo a seguir.



Um vendedor sai de uma das cidades com o intuito de visitar cada uma das outras cidades uma única vez e retornar ao seu ponto de partida. Com base no grafo e nessa informação, considere as afirmativas a seguir.

- I. O vendedor cumprirá seu propósito com êxito se sair de uma cidade par.
- II. O vendedor cumprirá seu propósito com êxito se sair de uma cidade ímpar.
- III. O vendedor não cumprirá seu propósito com êxito se sair de uma cidade par.
- IV. O vendedor não cumprirá seu propósito com êxito se sair de uma cidade ímpar.

Assinale a alternativa correta.

- a) Somente as afirmativas I e II são corretas.
- b) Somente as afirmativas I e IV são corretas.
- c) Somente as afirmativas III e IV são corretas.
- d) Somente as afirmativas I, II e III são corretas.
- e) Somente as afirmativas II, III e IV são corretas.

6 Árvores

Este capítulo aborda algumas questões relativas a árvores, considerando que uma árvore é um tipo particular de grafo. Iniciamos com a caracterização de árvores dentro da Teoria dos Grafos, em seguida apresentamos o conceito de árvores geradoras e um exemplo de aplicação na engenharia estrutural para solucionar o problema de rigidez de treliças planas. Por fim, este capítulo traz o conceito de grafo valorado, juntamente com o problema das árvores geradoras de custo mínimo e dois algoritmos para resolvê-lo.

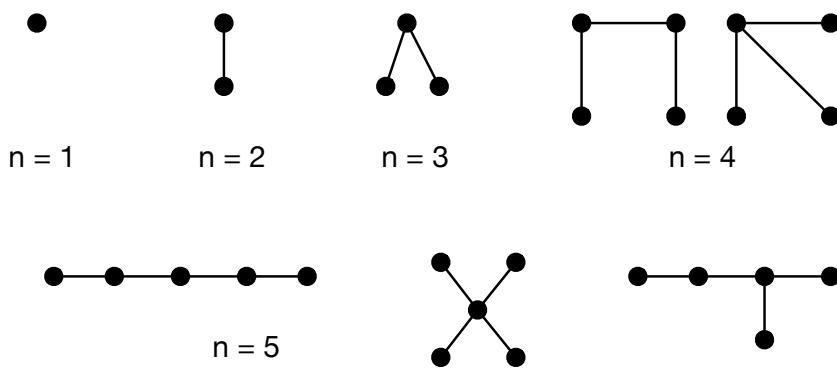
6.1 Caracterização de árvores

Árvores são importantes para o entendimento estrutural de grafos e para o design e análise de redes. Embora a disciplina de Estrutura de Dados trate de árvores como estruturas para armazenamento e recuperação eficiente de informação, entre outras aplicações, no presente capítulo as árvores são abordadas sob a ótica da Teoria dos Grafos.

Definição 6.1 (Árvore). *Uma árvore é um grafo sem ciclos.*

Exemplo 6.1. A *Figura 6.1* mostra todas as árvores não rotuladas com até 5 vértices.

Figura 6.1 – Árvores não rotuladas com até 5 vértices



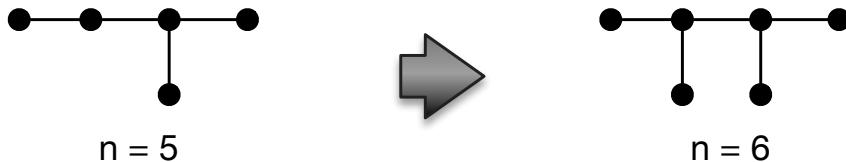
Fonte: o autor.

Árvores possuem algumas propriedades matemáticas interessantes, que auxiliam na sua caracterização como um tipo especial de Grafo. Mostraremos essas propriedades por meio das seguintes proposições:

Proposição 1: qualquer árvore com n vértices tem $n - 1$ arestas.

Qualquer árvore não rotulada com n vértices pode ser obtida a partir de uma árvore com $n - 1$ vértices adicionando-se uma aresta conectando o novo vértice a um vértice preexistente. Por exemplo, a [Figura 6.2](#) mostra o exemplo da obtenção de uma árvore não rotulada de 6 vértices a partir de uma com 5 vértices.

Figura 6.2 – Construção de nova árvore não rotulada



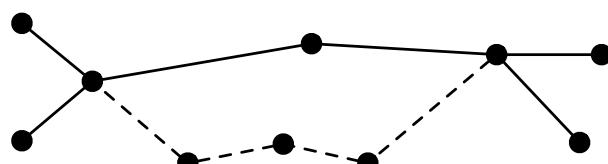
Fonte: o autor.

Iniciando por uma árvore com apenas um vértice, podemos construir qualquer árvore não rotulada adicionando sucessivamente uma nova aresta e um novo vértice. Em cada estágio, o número de vértices excede o número de arestas em 1 unidade, provando a veracidade da proposição 1.

Proposição 2: qualquer par de vértices em uma árvore está conectado por exatamente um caminho.

No processo de criação de árvores de n vértices a partir de árvores com $n - 1$ vértices, nenhum ciclo é criado porque a aresta adicionada conecta um vértice preexistente com o vértice novo. Em cada estágio o grafo permanece conexo, portanto, existe pelo menos um caminho conectando quaisquer dois vértices. Porém, eles não podem ser conectados por mais de um caminho porque qualquer par de caminhos configuraria um ciclo, conforme ilustrado pelas linhas tracejadas na [Figura 6.3](#).

Figura 6.3 – Formação de um ciclo

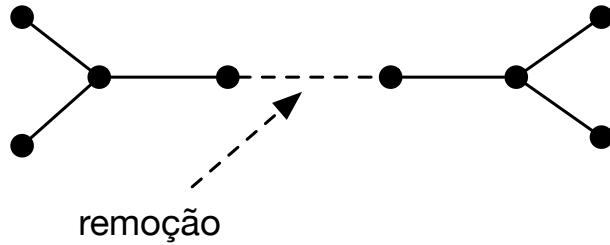


Fonte: o autor.

Proposição 3: a remoção de qualquer aresta de uma árvore desconecta a árvore.

A explicação para esta proposição é que quaisquer dois vértices **adjacentes** são conectados por exatamente um caminho — a aresta que os conecta — então a remoção desta aresta faz com que não haja mais caminho entre o par de vértices, conforme mostrado na [Figura 6.4](#).

Figura 6.4 – Remoção de aresta de uma árvore

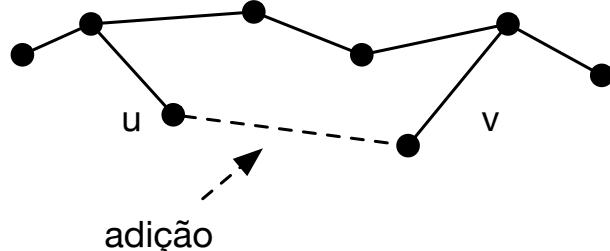


Fonte: o autor.

Proposição 4: a adição de uma aresta uma árvore cria um ciclo.

Considerando que dois vértices u e v de uma árvore já estão conectados por um caminho, a adição de uma nova aresta (u, v) produz um ciclo, o ciclo composto pelo caminho e pela aresta (u, v) , conforme ilustrado na [Figura 6.5](#).

Figura 6.5 – Adição de uma nova aresta em uma árvore



Fonte: o autor.

Estas proposições podem ser usadas como definições de árvores. O teorema [6.1](#) reúne as proposições em seis definições diferentes, sendo todas elas equivalentes. Qualquer uma das afirmações do teorema pode ser usada como definição de árvore e as outras cinco podem então ser deduzidas a partir dela.

Teorema 6.1 (Definições equivalentes de árvores). *Seja T um grafo com n vértices. Então as seguintes afirmações são equivalentes:*

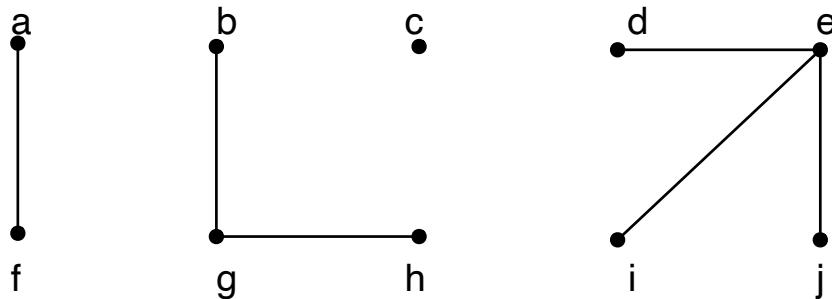
1. T é uma árvore¹.
2. T não tem ciclos e tem $n - 1$ arestas.
3. T é conexo e tem $n - 1$ arestas.
4. T é conexo e a remoção de qualquer uma de suas arestas o desconecta².
5. Quaisquer dois vértices de T estão conectados por exatamente um caminho.
6. T não contém ciclos. E para qualquer nova aresta e , o grafo $(T + e)$ tem exatamente um ciclo³.

Complementando as definições anteriores, temos o conceito de **floresta**:

Definição 6.2 (Floresta). *Floresta é um grafo no qual qualquer par de vértices está conectado **no máximo** por um caminho. Uma floresta pode ser composta de uniões disjuntas de árvores.*

Exemplo 6.2. A [Figura 6.6](#) mostra um exemplo de floresta: o grafo $G = (V, E)$, sendo $V = \{a, b, c, d, e, f, g, h, i, j\}$ e $E = \{(a, f), (b, g), (h, g), (d, e), (j, e), (e, i)\}$

Figura 6.6 – Exemplo de Floresta



Fonte: o autor.

Ao considerarmos grafos dirigidos, a definição de árvore é significativamente diferente, como veremos a seguir.

Definição 6.3 (Árvore dirigida). *Uma árvore dirigida T é um digrafo acíclico onde o grau de entrada de cada vértice é 1, exceto o do vértice raiz, que tem grau de entrada zero. A raiz de uma árvore dirigida T é um vértice r tal que qualquer outro vértice de T pode ser alcançado a partir de r .*

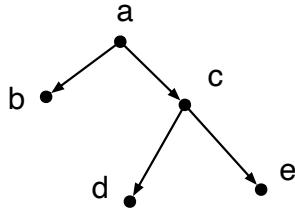
¹ Ou seja, T é um grafo conexo sem ciclos.

² Em outras palavras, todas as arestas de T são pontes.

³ A adição da aresta e cria um ciclo em T .

Exemplo 6.3. A *Figura 6.7* mostra uma árvore dirigida, com o vértice a sendo sua raiz.

Figura 6.7 – Árvore dirigida



Fonte: o autor.

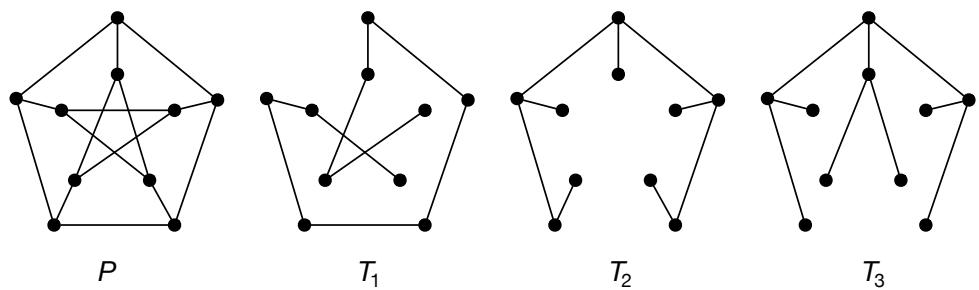
6.2 Árvores geradoras

As árvores geradoras, do inglês *Spanning Trees*⁴ são amplamente utilizadas na engenharia e na tecnologia da informação para otimização e redução de custos de redes. Um exemplo deste conceito em tecnologia é o protocolo de redes *Spanning Tree Protocol*, que impede a formação de loops quando switches ou pontes são interligadas por vários caminhos diferentes, construindo redes lógicas livres de loops em redes Ethernet (STEVENS, 1993).

Definição 6.4 (Árvore geradora). *Seja G um grafo conexo. Uma árvore geradora T em G é um subgrafo de G que inclui todos os seus vértices e também é uma árvore.*

Exemplo 6.4. A *Figura 6.8* mostra o grafo de Petersen P , com três de suas árvores geradoras: T_1 , T_2 e T_3 . Se o grafo for rotulado, ele tem 2000 árvores geradoras.

Figura 6.8 – Árvores geradoras do grafo de Petersen



Fonte: o autor.

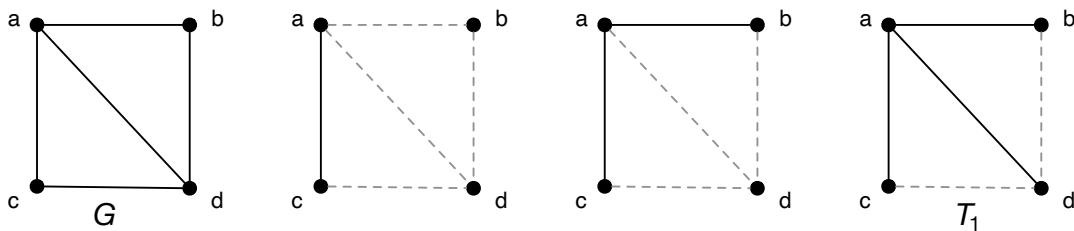
Dado um grafo conexo G , intuitivamente podemos utilizar dois métodos diferentes para encontrar uma árvore geradora T em G , o método por construção e o método por

⁴ Não há um consenso sobre a tradução deste termo para o português. Alguns autores traduzem como “árvore de abrangência” ou “árvore de extensão”. Nós preferimos usar o termo “árvore geradora”.

remoção. O conhecimento desses métodos é útil para a compreensão dos algoritmos para o problema do conector mínimo, que veremos a seguir.

No método construtivo, selecionamos arestas do grafo, uma de cada vez, tomando o cuidado para não formar ciclos. Repetimos o procedimento até que todos os vértices sejam incluídos. Por exemplo, no grafo G da Figura 6.9, selecionamos as arestas (a, c) , (a, c) e (a, c) , obtendo a árvore geradora T_1 .

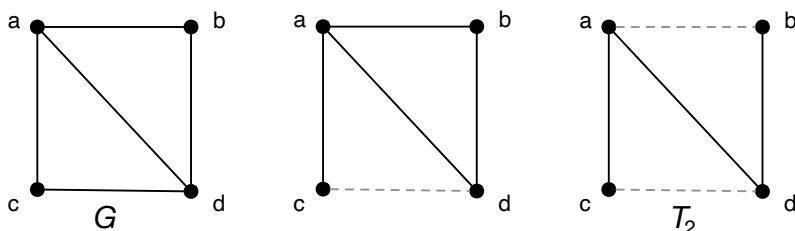
Figura 6.9 – Árvores geradora pelo método construtivo



Fonte: o autor.

No método por remoção, escolhemos um ciclo e removemos qualquer uma de suas arestas. Repetimos este procedimento até que não restem mais ciclos. Por exemplo, no grafo G da Figura 6.10, removemos a aresta (c, d) , destruindo o ciclo acd , em seguida removemos a aresta (a, b) destruindo o ciclo abd e obtendo a árvore geradora T_2 .

Figura 6.10 – Árvores geradora por redução



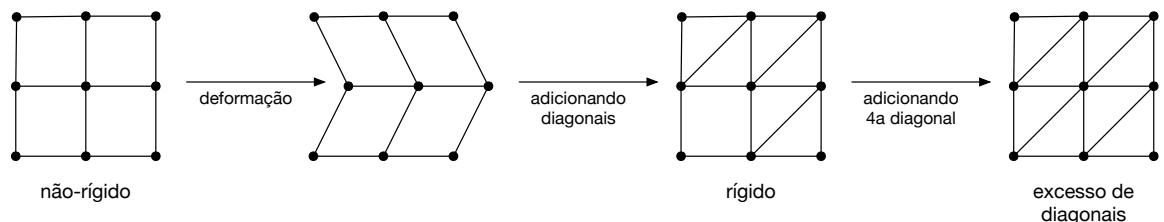
Fonte: o autor.

6.2.1 Estudo de caso: rigidez de treliças planas

Treliças planas são estruturas muito utilizadas na construção de telhados, galpões, pontes e torres de comunicação. Compostas por barras metálicas ou de madeira que funcionam por tração ou compressão, elas tem como característica principal a falta de rigidez para rotação nas conexões entre as barras. Isso faz com que a estrutura possa ser facilmente deformada no plano, o que não é desejável. Para adicionar rigidez à estrutura, utilizam-se barras diagonais. A Figura 6.11 mostra um exemplo de treliça não rígida com

2 linhas e duas colunas. Uma força lateral facilmente a deformaria. Ao adicionarmos 3 diagonais, tornamos a estrutura rígida. Note que não seria necessário adicionar uma quarta diagonal, visto que estariam “gastando” material desnecessariamente em uma estrutura já rígida.

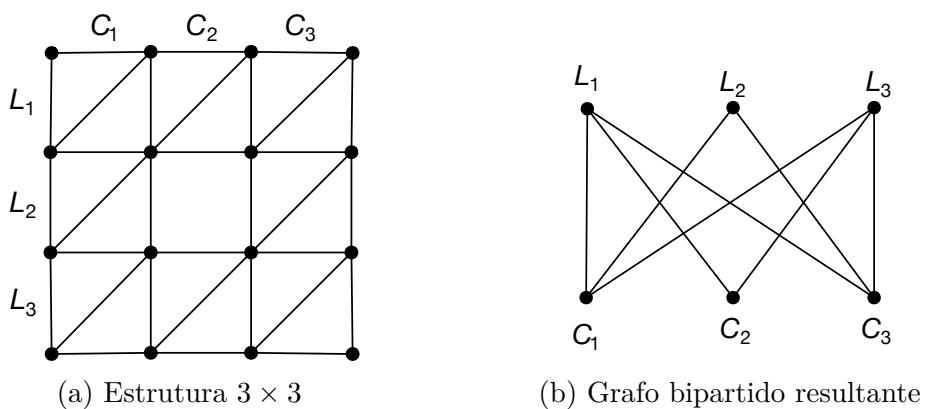
Figura 6.11 – Adicionando rigidez a uma treliça plana



Fonte: o autor.

Podemos utilizar grafos para analisar a questão de rigidez, bem como falta ou excesso de diagonais em treliças planas. Para modelar estes grafos, vamos considerar as linhas e colunas de retângulos formados nos espaços entre as barras horizontais e verticais. Enumerando as linhas de retângulo sequencialmente de cima pra baixo, temos l_1, l_2, \dots . Analogamente, enumeramos as colunas c_1, c_2, \dots da esquerda para a direita. O retângulo na linha i e coluna j é $r(i, j)$. Na modelagem do problema, criamos um grafo bipartido $G = (V_1 \cup V_2, E)$, em que $V_1 = \{l_1, l_2, \dots\}$, $V_2 = \{c_1, c_2, \dots\}$ e $E = \{(i, j) |$ existe uma barra diagonal no retângulo $r(i, j)\}$. A Figura 6.12a mostra uma estrutura de treliça com 3 linhas e 3 colunas. Na Figura 6.12b, podemos ver o grafo bipartido resultante da modelagem do problema. Note que os vértices l_2 e c_2 não são adjacentes porque não existe barra diagonal no retângulo $r(2, 2)$ da estrutura.

Figura 6.12 – Modelagem de treliças



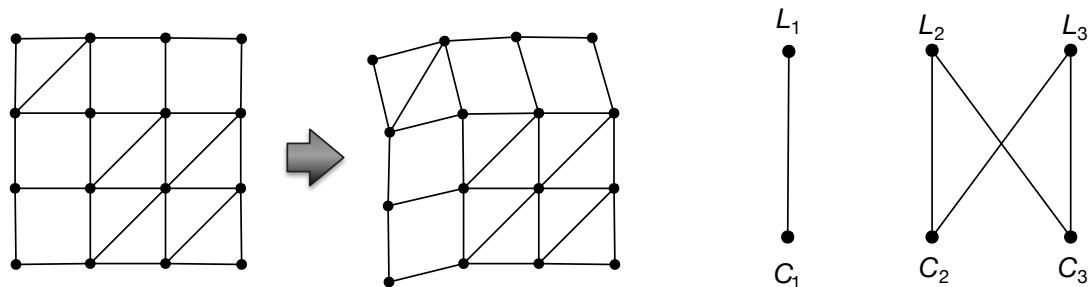
Fonte: o autor.

Para a estrutura de treliça ser rígida, as seguintes condições precisam ser atendidas: (i) a linha l_i deve permanecer paralela à linha l_j , para todas as linhas l_i e l_j ; (ii) a coluna

c_i deve permanecer paralela à coluna c_j , para todas as colunas c_i e c_j ; e (iii) a linha l_i deve permanecer perpendicular à coluna c_j , para todas as linhas l_i e colunas c_j .

Treliças rígidas sempre resultam em grafos bipartidos conexos, como o da Figura 6.12. Por outro lado, se o grafo for não-conexo, significa que a treliça modelada não tem rigidez, podendo ser facilmente deformada como a estrutura mostrada na Figura 6.13.

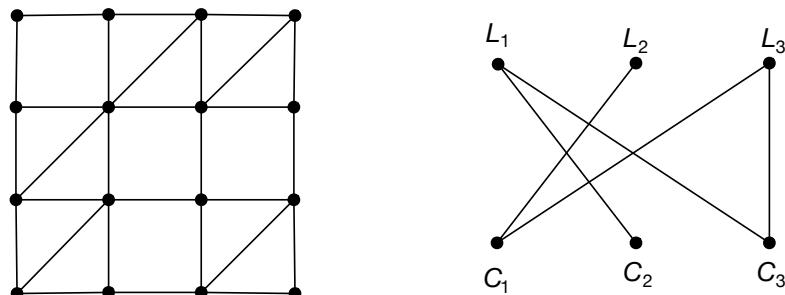
Figura 6.13 – Treliça sem rigidez



Fonte: o autor.

Além disso, se o grafo resultante da modelagem da estrutura for uma árvore geradora, a estrutura está numa configuração rígida utilizando o mínimo de barras diagonais possível para manter a rigidez, conforme ilustrado pelo exemplo da Figura 6.14.

Figura 6.14 – Treliça rígida com o mínimo de barras diagonais



Fonte: o autor.

6.3 Grafos valorados

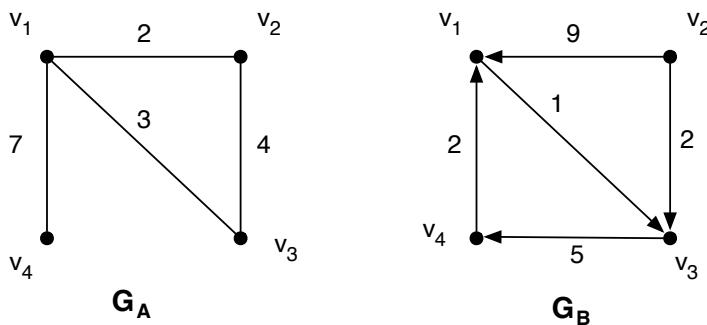
A seguir, veremos o problema das árvores geradoras de custo mínimo. Mas antes precisamos conhecer um novo tipo de grafo, chamado Grafo Valorado.

Definição 6.5 (Grafo valorado). *Um grafo valorado é um grafo (dirigido ou não dirigido) no qual cada uma de suas arestas está associada a um valor numérico chamado peso ou custo da aresta.*

O custo de uma aresta representa algum dado significativo do problema modelado pelo grafo. Este dado pode ser distância, custo monetário, tempo, ou qualquer outro parâmetro relevante que pode ser expresso como um valor numérico. Dependendo do problema, podemos, inclusive, ter custos negativos. Dependendo do contexto, o custo de uma aresta (v_i, v_j) pode ser denotado por $c_{i,j}$ (de “custo”), $d_{i,j}$ (de “distância”) ou $w_{i,j}$ (da palavra “weight”). Um custo de valor infinito entre v_i e v_j representa ausência de aresta ou de caminho entre estes dois vértices.

A Figura 6.15 ilustra dois exemplos de grafos valorados. No grafo G_A , por exemplo, o custo da aresta (v_1, v_2) é igual a 2, ou $c_{1,2} = d_{1,2} = w_{1,2} = 2$.

Figura 6.15 – Exemplos de grafos valorados



Fonte: o autor.

Definição 6.6 (Matriz de custos). *Dado um grafo valorado $G = (V, E)$ com n vértices, sua matriz de custos é uma matriz W de tamanho $n \times n$, com seus elementos definidos da seguinte forma:*

$$w_{i,j} = \begin{cases} 0 & \text{se } v_i = v_j, \\ \infty & \text{se } (v_i, v_j) \notin E, \\ \text{custo} & \text{se } (v_i, v_j) \in E. \end{cases}$$

Exemplo 6.5. Para os grafos G_a e G_b , da Figura 6.15, teríamos, respectivamente as seguintes matrizes de custo W_a e W_b :

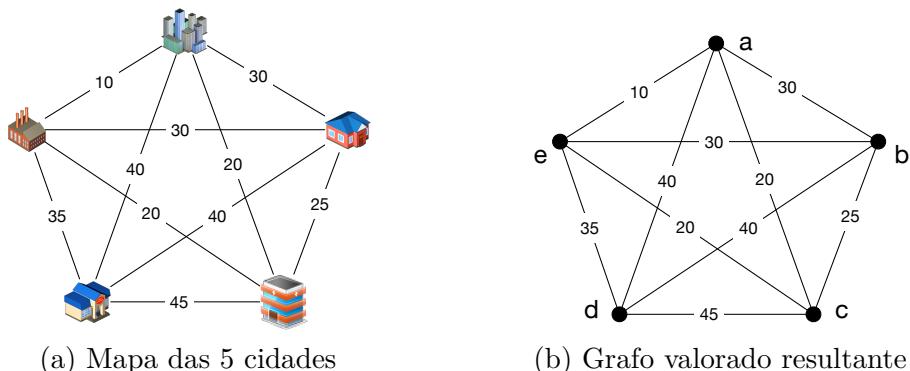
$$W_a = \begin{matrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & \begin{bmatrix} 0 & 2 & 3 & 7 \\ 2 & 0 & 4 & \infty \\ 3 & 4 & 0 & \infty \\ 7 & \infty & \infty & 0 \end{bmatrix} \\ v_2 & \\ v_3 & \\ v_4 & \end{matrix} \quad W_b = \begin{matrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & \begin{bmatrix} 0 & \infty & e_3 & \infty \\ 9 & 0 & 2 & \infty \\ \infty & \infty & 0 & 5 \\ 2 & \infty & \infty & 0 \end{bmatrix} \\ v_2 & \\ v_3 & \\ v_4 & \end{matrix}$$

6.4 Problema de árvore geradora de custo mínimo

No desenho de circuitos eletrônicos, uma série de componentes é interconectada por fios localizados em uma placa de apoio. Por diversas razões (técnicas, econômicas etc.), é desejável que se use a menor quantidade possível de fio no circuito. Um problema semelhante ocorre, por exemplo em redes de computadores e projetos de redes de fornecimento de água tratada, eletricidade ou estradas.

Para ilustrar o problema, a Figura 6.16a mostra 5 localidades mutuamente interligadas por estradas não pavimentadas. Os valores associados às linhas correspondem à distância de cada trecho de estrada. Queremos pavimentar estradas usando a quantidade mínima de asfalto de forma que todas as localidades estejam interligadas indiretamente por estradas asfaltadas. Portanto, precisamos definir quais estradas serão pavimentadas e quais não serão.

Figura 6.16 – Problema de pavimentação de estradas



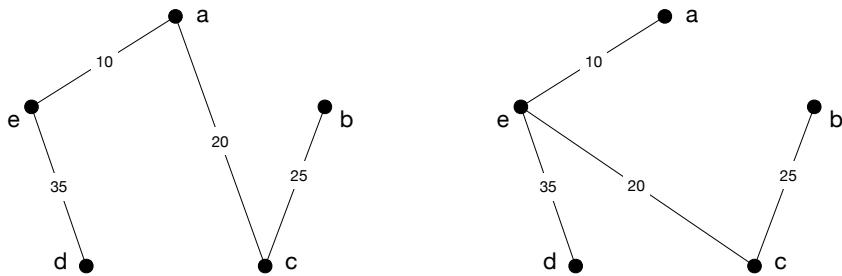
Fonte: o autor.

Para buscar a solução do problema, vamos modelar o mapa como um grafo valorado, mostrado na Figura 6.16b tendo as cidades como vértices e os trechos de estradas como arestas valoradas. O custo de cada aresta corresponde à extensão de cada trecho de estrada.

Sabendo que a pavimentação das estradas deve ser feita utilizando a quantidade mínima de asfalto e que todas as cidades devem estar “conectadas” indiretamente por estradas pavimentadas, a escolha de quais estradas pavimentar leva a um desenho de

árvore geradora do grafo. De todas as árvores geradoras, queremos encontrar aquela cuja soma dos custos de suas arestas seja o menor possível, correspondendo ao projeto mais “barato” para a pavimentação da malha rodoviária. A Figura 6.17 mostra duas soluções, ambas com custo total no valor de 90.

Figura 6.17 – Duas soluções do problema de pavimentação de estradas



Fonte: o autor.

Este problema é chamado de Problema da Árvore Geradora de Custo Mínimo ou Problema do Mínimo Conector. Existem dois algoritmos para encontrar árvores geradoras de custo mínimo em grafos valorados: o algoritmo de Prim e o algoritmo de Kruskal. Antes de conhecer os algoritmos, vamos definir o conceito de árvore geradora de custo mínimo.

Definição 6.7 (Árvore geradora de custo mínimo). *Seja T uma árvore geradora com custo total mínimo em um grafo conexo valorado G . Então, T é uma árvore geradora de custo mínimo ou um conector mínimo de G .*

6.4.1 Algoritmo de Prim

Este algoritmo foi inicialmente desenvolvido pelo matemático tcheco Vojtěch Jarník em 1930 ([JARNÍK, 1931](#)), e depois redescoberto e publicado por Robert Prim em 1957 ([PRIM, 1957](#)). Trata-se de um algoritmo guloso e sua ideia básica é iniciar a formação da árvore geradora de custo mínimo a partir de um vértice raiz arbitrário. Em cada iteração do algoritmo, a árvore cresce por meio da adição de um novo vértice. Dentre todos os vértices candidatos a se conectarem na árvore, é escolhido aquele que adiciona o menor custo possível naquele momento.

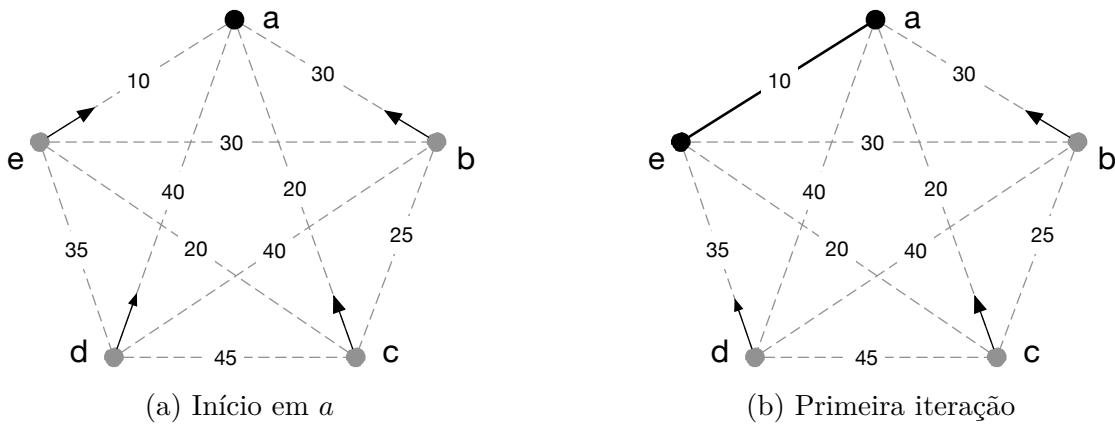
Informalmente, o algoritmo pode ser descrito pelos seguintes passos:

1. Inicie a árvore a partir de um vértice arbitrário;
2. Faça a árvore crescer com uma aresta. Dentre as arestas que conectam a árvore a vértices que ainda não fazem parte da árvore, escolha a aresta de menor custo;
3. Repita o passo anterior até que todos os vértices sejam conectados à árvore.

O pseudocódigo do algoritmo [algoritmo 6.1](#) é descrito em [Cormen et al. \(2001\)](#), e apresenta uma forma interessante de implementar o processo de agregação de novos vértices à árvore. Ao invés da árvore manter controle de suas opções de crescimento “de dentro para fora”, a lógica é invertida e cada vértice ainda não conectado à árvore mantém o custo mínio atual para se conectar e uma referência para o vértice da árvore no qual ele se conectararia com este custo mínimo. Estes dados são mantidos no algoritmo como predecessor (ρ) e “chave” de cada vértice. O algoritmo também usa uma fila de prioridades para escolher em cada iteração o vértice com o menor custo para se conectar à árvore naquele momento.

Por exemplo, para o grafo da [Figura 6.16b](#), se escolhermos arbitrariamente o vértice **a** para iniciar a construção da árvore, temos os seguintes valores de predecessor e chave: $\rho(b) = a$ e $chave(b) = 30$; $\rho(c) = a$ e $chave(c) = 20$; $\rho(d) = a$ e $chave(d) = 35$; $\rho(e) = a$ e $chave(e) = 10$. A [Figura 6.18a](#) mostra as referências para predecessores como setas pequenas. Na próxima iteração do algoritmo, o vértice **e** é escolhido para se conectar à árvore por ter o menor custo de “conexão”. Neste momento, os demais vértices que ainda não fazem parte da árvore verificam se existe opção mais barata para se conectar à árvore considerando o vértice recém adicionado (vértice **e**). Note que na [Figura 6.18b](#), o vértice **d** tem seus atributos atualizados para $\rho(d) = e$ e $chave(d) = 35$.

Figura 6.18 – Desenvolvimento do algoritmo de Prim



Fonte: o autor.

O pseudocódigo do algoritmo de Prim tem os seguintes elementos:

- G : um grafo valorado
- W : matriz de custos do grafo
- r : vértice raiz da árvore, escolhido arbitrariamente;
- $\rho(v)$: predecessor do vértice v na árvore. Representa também referência para o vértice v se conectar à árvore com custo mínio;

- $chave(v)$: custo mínimo atual para o vértice v se conectar à árvore; item Q : é uma fila de prioridades (mínima) de vértices, tendo como chave o custo mínimo atual para o vértice se conectar à árvore. A operação $\text{REMOVE-MINIMO}(Q)$ retira da fila e retorna o vértice com menor valor atual de chave
- $adj(u)$: é uma iteração com todos os vértices adjacentes ao vértice u (veja seção 2.2);
- $w_{u,v}$: custo da aresta (u, v) .

Algoritmo: PRIM(G, W, r)

```

// Inicialização dos atributos dos vértices
para cada  $v \in V$  faça
    |    $\rho(v) \leftarrow \text{nil};$ 
    |    $chave(v) \leftarrow \infty;$ 

// Inicialização do vértice raiz
 $chave(r) \leftarrow 0;$ 
// Inicialização da fila de prioridades
 $Q \leftarrow V;$ 

// Laço principal
enquanto  $Q \neq \emptyset$  faça
    |    $u \leftarrow \text{REMOVE\_MINIMO}(Q);$ 
    |   para cada  $v \in Adj(u)$  faça
        |       |   se  $(v \in Q)$  e  $(w_{u,v} < chave(v))$  então
        |           |       |    $\rho(v) \leftarrow u;$ 
        |           |       |    $chave(v) \leftarrow w_{u,v};$ 
```

Algoritmo 6.1: Algoritmo de Prim

O algoritmo de Prim encontra uma solução ótima, isto é, ele encontra uma árvore geradora de custo mínimo. Seu desempenho, todavia, depende fundamentalmente da implementação da fila de prioridades. Caso a fila seja implementada com *binary heaps* a complexidade do algoritmo de Prim é $O(E \log V)$.

6.4.2 Algoritmo de Kruskal

Anteriormente vimos o método construtivo para formação de árvores geradoras. Kruskal (1956) se inspirou neste método para desenvolver um algoritmo para determinação de árvores geradoras de custo mínimo. O algoritmo de Kruskal encontra uma **floresta** geradora de custo mínimo em grafos valorados. Se o grafo for conexo, então o algoritmo encontra uma **árvore** geradora de custo mínimo.

Basicamente, trata-se de um algoritmo guloso que adiciona uma nova aresta de custo mínimo em cada passo, tomando o cuidado de não formar ciclos durante este processo. Em resumo, o algoritmo pode ser descrito pelos seguintes passos:

1. Crie uma floresta F na qual cada vértice do grafo individualmente é uma árvore;
2. Crie um conjunto S com todas as arestas do grafo;
3. Enquanto S não estiver vazio e F ainda puder crescer:
 - a) remova de S uma aresta com custo mínimo;
 - b) se a aresta removida de S conectar duas árvores diferentes, então adiciona a aresta à floresta F , unindo as duas árvores para formar uma árvore única.

O aspecto mais complicado do algoritmo é evitar a formação de ciclos, que pode ocorrer pela inclusão em F de uma aresta que conecta dois vértices que já fazem parte de uma mesma árvore. Uma maneira extremamente elegante, simples e eficiente para resolver esta questão é apresentada por [Cormen et al. \(2001\)](#), que sugerem o uso de estruturas de dados de conjuntos disjuntos de vértices para manter o controle da formação da floresta e evitar a formação de ciclos (veja [seção 4.5](#) sobre estrutura de dados para conjuntos disjuntos de vértices).

Algoritmo: KRUSKAL(G, W)

```

// Inicialização da árvore
 $T \leftarrow \emptyset$ ;
// Inicialização dos conjuntos
para cada  $v \in V$  faça
  | CriaConjunto( $v$ );
// Ordenação das arestas
Ordene as arestas de  $E$  em ordem crescente de custo;
// Laço principal
para cada  $(u, v) \in E$ , em ordem crescente de custo faça
  | se BuscaConjunto( $u$ ) = BuscaConjunto( $v$ ) então
    |   | Uniao( $u, v$ );
    |   |  $T \leftarrow T \cup (u, v)$ ;
  |

```

Algoritmo 6.2: Algoritmo de Kruskal

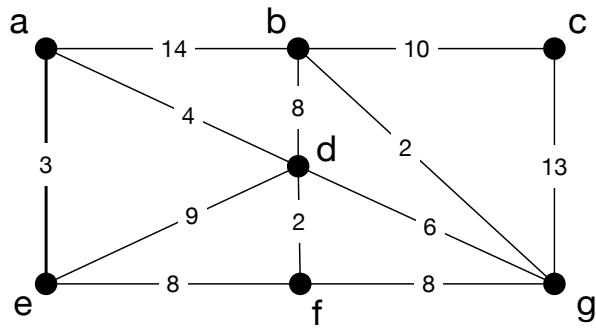
O pseudocódigo mostrado no [algoritmo 6.2](#) implementa o método de Kruskal utilizando a estrutura de conjuntos disjuntos. Os principais elementos do pseudocódigo são os seguintes:

- G e W : representam um grafo valorado, sendo W o conjunto de informações sobre os custos das arestas (em uma matriz de custos, por exemplo);
- T : é a árvore geradora de custo mínimo (pode ser armazenada em uma estrutura de dados de grafo não dirigido);
- $CriaConjunto(v)$, $BuscaConjunto(v)$ e $Uniao(u, v)$: são operações sobre a estrutura de dados de conjuntos disjuntos, conforme descrito na [seção 4.5](#).

O tempo de execução do algoritmo para um grafo $G = (V, E)$ depende do algoritmo utilizado para a ordenação das arestas e da implementação da estrutura de dados de conjuntos disjuntos. A inicialização da árvore leva tempo $O(1)$. Assumindo que estamos utilizando a implementação apresentada na [seção 4.5](#), a inicialização dos conjuntos disjuntos leva tempo $O(V)$ para ser realizada. A ordenação de arestas pode ser feita em tempo $O(E \log E)$. O laço principal executa $O(E)$ operações *BuscaConjunto* e *Uniao*, resultando em um tempo $O(E + \alpha(V))$, em que $\alpha(V)$ é uma função de decresce assintoticamente devido ao mecanismo de achatamento das árvores que representam os conjuntos disjuntos de vértices, fazendo com que $\alpha(V) = O(\log V) = O(\log E)$. Portanto podemos considerar que o algoritmo é $O(E \log V)$.

Como exemplo de execução do algoritmo de Kruskal, vamos utilizá-lo para obter uma árvore geradora de custo mínimo do grafo da [Figura 6.19](#).

Figura 6.19 – Grafo valorado para o algoritmo de Kruskal



Fonte: o autor.

Ordenando as arestas por ordem crescente de custo, podemos obter a seguinte sequência, mostrada no [Quadro 3](#).

Quadro 3 – Sequência de arestas ordenadas

v_i	d	b	a	a	g	e	f	b	e	b	c	a
v_j	f	g	e	d	d	f	g	d	d	c	g	b
custo	2	2	3	4	6	8	8	8	9	10	13	14

Fonte: o autor.

A [Figura 6.20](#) mostra passo a passo a formação da árvore geradora de custo mínimo juntamente com a representação da floresta por meio dos conjuntos disjuntos de vértices.

No [Quadro 4](#), podemos acompanhar passo a passo a evolução da floresta formada pelos conjuntos disjuntos de vértices, seguindo a sequência de arestas ordenadas, mostrada no [Quadro 3](#). Note que os dados apresentados no [Quadro 4](#) estão relacionados com as representações gráficas da [Figura 6.20](#). A primeira linha do quadro mostra o estado dos conjuntos logo após sua inicialização. Neste momento, temos uma floresta formada por 7 vértices individuais, conforme mostrado na [Figura 6.20a](#). As arestas ainda não incluídas na árvore geradora de custo mínimo são mostradas em linhas cinza tracejadas.

Logo em seguida, na primeira iteração do laço principal, o algoritmo tenta incluir a aresta (d, f) na árvore geradora de custo mínimo. Como os vértices d e f não fazem parte da mesma árvore, a inclusão da aresta não vai criar um ciclo. Portanto, a aresta é incluída na árvore, e os conjuntos que contém os vértices d e f são unidos por meio da operação $uniao(d, f)$.

Quadro 4 – Formação dos conjuntos disjuntos

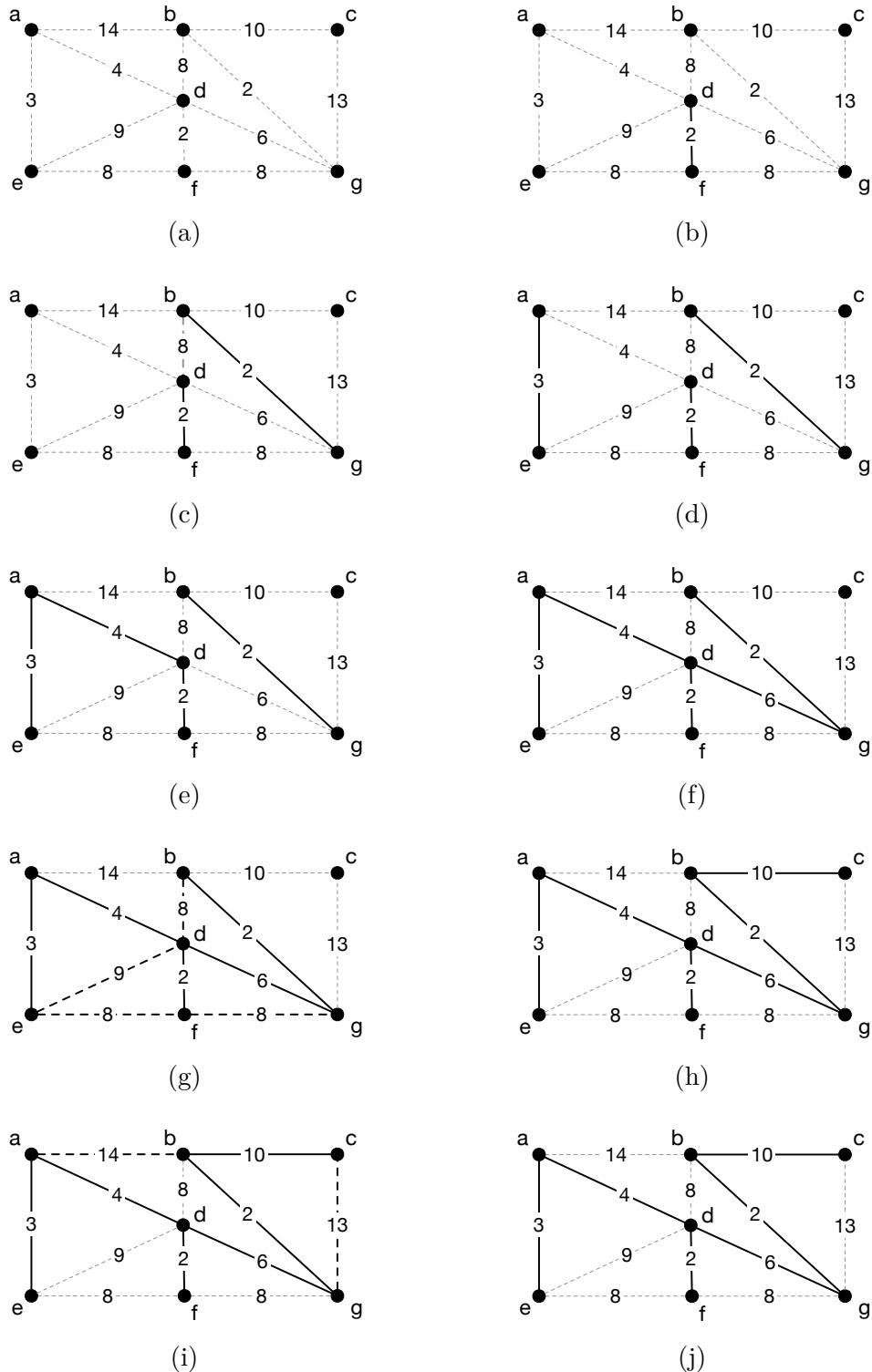
aresta	operação	Coleção de conjuntos disjuntos						
<i>inicio</i>	—	{a}	{b}	{c}	{d}	{e}	{f}	{g}
(d, f)	$uniao(d, f)$	{a}	{b}	{c}	{d, f}	{e}		{g}
(b, g)	$uniao(b, g)$	{a}	{b, g}	{c}	{d, f}	{e}		
(a, e)	$uniao(a, e)$		{b, g}	{c}	{d, f}		{e, a}	
(a, d)	$uniao(a, d)$		{b, g}	{c}	{d, f, e, a}			
(g, d)	$uniao(g, d)$			{c}	{d, f, e, a, b, g}			
(e, f)	—			{c}	{d, f, e, a, b, g}			
(f, g)	—			{c}	{d, f, e, a, b, g}			
(b, d)	—			{c}	{d, f, e, a, b, g}			
(e, d)	—			{c}	{d, f, e, a, b, g}			
(b, c)	$uniao(b, c)$				{d, f, e, a, b, g, c}			
(c, g)	—				{d, f, e, a, b, g, c}			
(a, b)	—				{d, f, e, a, b, g, c}			

Fonte: o autor.

O algoritmo continua a adicionar sucessivamente arestas à árvore até chegar na aresta (e, f) . Neste ponto os vértices e e f já fazem parte da mesma árvore na floresta. Por conseguinte, a inclusão desta aresta formaria um ciclo e ela não é incluída na árvore. O mesmo acontece com as arestas (f, g) , (b, d) , e (e, d) . A [Figura 6.20g](#) mostra estas quatro arestas em linhas pretas tracejadas: claramente podemos observar que a inclusão de qualquer uma delas na floresta formaria um ciclo.

Na iteração subsequente, a aresta (b, c) é adicionada à árvore geradora de custo mínimo, e depois as arestas (a, b) e (c, g) são ignoradas por formarem ciclos (mostradas em linhas pretas pontilhadas na [Figura 6.20i](#)). Finalmente, o algoritmo encerra encontrando a árvore geradora de custo mínimo mostrada na [Figura 6.20j](#), com custo total igual a 27.

Figura 6.20 – Exemplo de execução do algoritmo de Kruskal

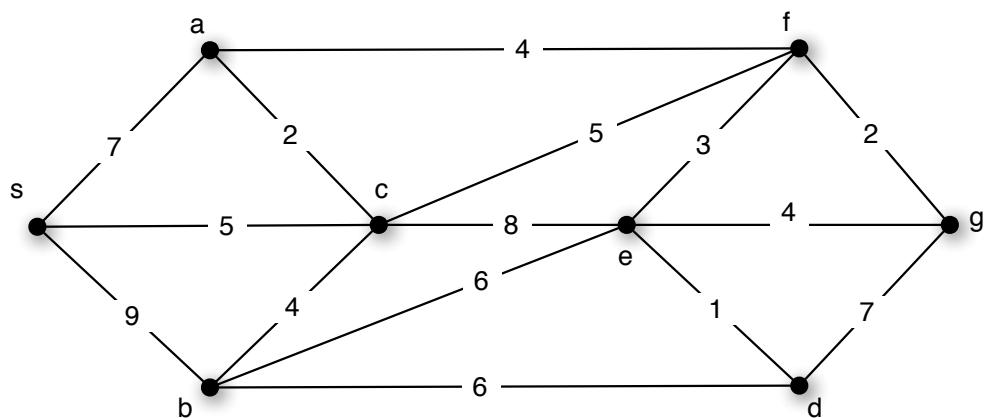
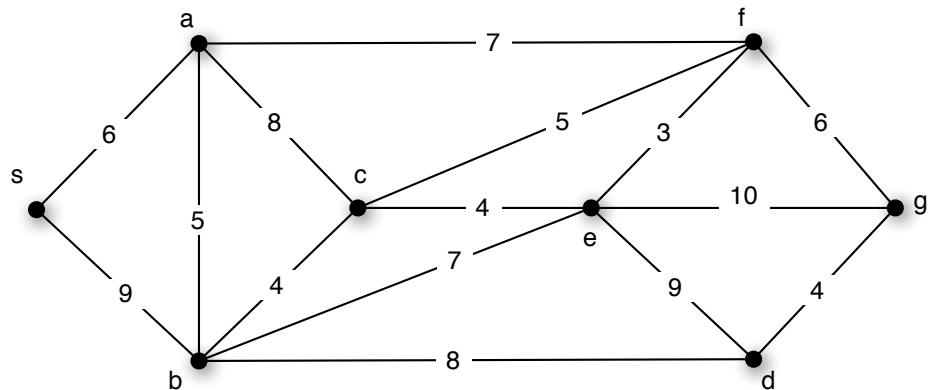
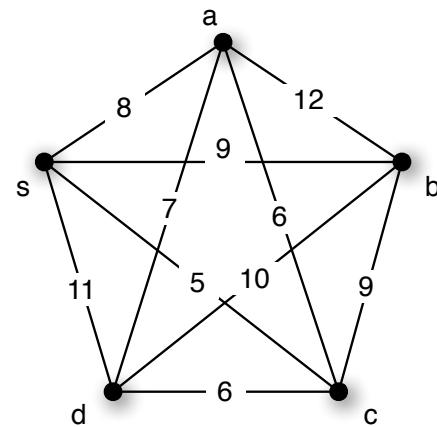
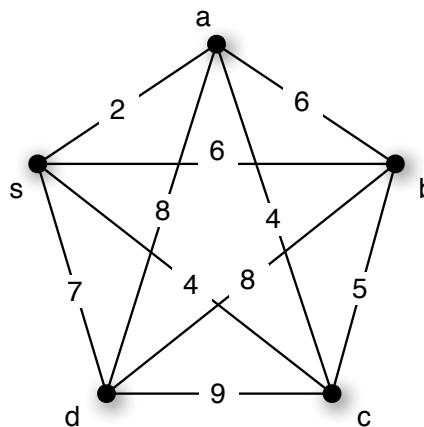


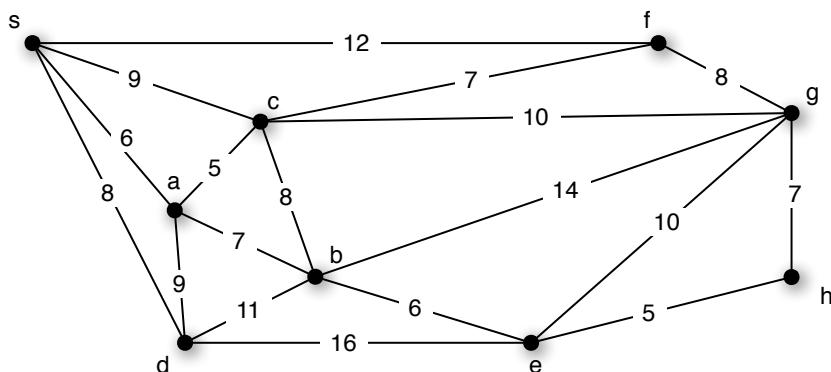
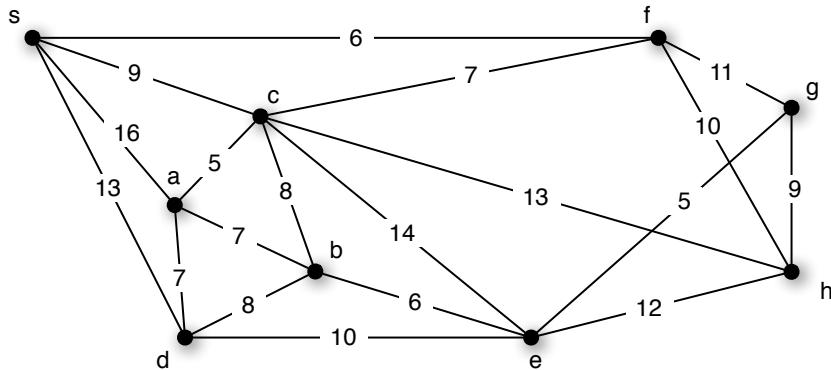
Fonte: o autor.

6.5 Exercícios

1. Sabendo que uma árvore é um tipo particular de grafo, escreva pelo menos cinco definições diferentes para árvores.
2. Desenhe um exemplo de uma árvore com sete vértices e...
 - a) exatamente dois vértices de grau 1;
 - b) exatamente quatro vértices de grau 1;
 - c) exatamente seis vértices de grau 1.
3. Utilize o “lema do aperto de mão” (*handshaking lemma*) para provar que qualquer árvore com n vértices, em que $n \geq 2$, tem pelo menos dois vértices de grau 1.
4. Uma **floresta** é um grafo (não necessariamente conexo), em que cada uma de suas componentes conexas é uma árvore.
 - a) Seja G uma floresta com n vértices e k componentes conexas. Quantas arestas existem em G ?
 - b) Construa (e desenhe) uma floresta com 12 vértices e 9 arestas.
 - c) É verdade que qualquer floresta com k componentes tem pelo menos $2k$ vértices de grau 1?
5. Para cada um dos itens abaixo, tente desenhar um grafo que atenda a descrição dada, ou então explique por que tal grafo não existe.
 - a) Um grafo simples com 6 vértices, 2 componentes conexas e 11 arestas.
 - b) Um grafo simples com 7 vértices, 3 componentes conexas e 10 arestas.
 - c) Um grafo simples com 8 vértices, 2 componentes conexas, 9 arestas e exatamente 3 ciclos.
 - d) Um grafo simples com 8 vértices, 2 componentes conexas, 10 arestas e exatamente 3 ciclos.
 - e) Um grafo simples com 9 vértices, 2 componentes conexas, 10 arestas e exatamente 2 ciclos.
 - f) Um grafo simples conexo com 9 vértices, 12 arestas e exatamente 2 ciclos.
 - g) Um grafo simples conexo com 9 vértices, 12 arestas e exatamente 3 ciclos.
 - h) Uma floresta com 10 vértices, 2 componentes conexas e 9 arestas.
 - i) Uma floresta com 10 vértices, 3 componentes conexas e 9 arestas.
 - j) Um grafo simples conexo com 11 vértices, 14 arestas e 5 ciclos disjuntos (ciclos que não compartilham arestas).

6. Prove que a seguinte afirmação é verdadeira ou falsa: “Qualquer grafo simples conexo com n vértices e n arestas deve conter exatamente 1 ciclo”.
7. Dados os grafos abaixo, encontre suas respectivas árvores geradoras de custo mínimo utilizando o algoritmo de Prim. Mostre passo a passo a solução de acordo com os passos de execução do algoritmo, mostrando os valores de $chave(v)$ e $\rho(v)$ para todos os vértices. Utilize o vértice s como vértice inicial e resolva empates dando preferência para os vértices em ordem alfabética. Desenhe as árvores resultantes e calcule seus custos.





8. Encontre as respectivas árvores geradoras de custo mínimo dos grafos da questão anterior utilizando o algoritmo de Kruskal. Escreva a lista de arestas em ordem crescente de custo. Escreva a lista de conjuntos de vértices em cada passo do algoritmo, indicando a ocorrência de operações $UNION(u, v)$. Desenhe as árvores resultantes e calcule seus custos.
9. Adapte os algoritmos de Prim e Kruskal para encontrar árvores geradoras de custo **máximo** de grafos valorados.
10. Implemente o algoritmo de Prim.
11. Implemente o algoritmo de Kruskal.
12. (PosComp – 2002) Considere um grafo G satisfazendo as seguintes propriedades:
 (i) G é conexo e (ii) se removermos qualquer aresta de G , o grafo obtido é desconexo. Então é correto afirmar que o grafo G é:
- Um circuito (ciclo)
 - Não bipartido
 - Uma árvore
 - Hamiltoniano
 - Euleriano

13. (PosComp – 2008) Um grafo $G(V, E)$ é uma árvore se G é conexo e acíclico. Assinale a definição que **NÃO** pode ser usada para definir árvores.
- G é conexo e o número de arestas é mínimo.
 - G é conexo e o número de vértices excede o número de arestas por uma unidade.
 - G é acíclico e o número de vértices excede o número de arestas por uma unidade.
 - G é acíclico e, para todo par de vértices v, w , que não são adjacentes em G , a adição da aresta (v, w) produz um grafo contendo exatamente um ciclo.
 - G é acíclico e o número de arestas é mínimo.
14. (PosComp – 2008) Seja $G(V, E)$ um grafo tal que $|V| = n$ e $|E| = m$. Analise as seguintes sentenças:
- Se G é acíclico com no máximo $n - 1$ arestas, então G é uma árvore.
 - Se G é um ciclo, então G tem n árvores geradoras distintas.
 - Se G é conexo com no máximo $n - 1$ arestas, então G é uma árvore.
 - Se G é conexo e tem um ciclo, então para toda árvore geradora T de G , $E(G) - E(T) = \emptyset$.
- A análise permite concluir que (assinale a alternativa correta):
- apenas os itens I e III são verdadeiros.
 - apenas os Itens II e III são verdadeiros.
 - apenas o item I é falso.
 - todos os itens são verdadeiros.
 - apenas os itens II e IV são verdadeiros.
15. (PosComp – 2012) Sejam $G = (V, E)$ um grafo conexo não orientado com pesos distintos nas arestas e $e \in E$ uma aresta fixa, em que $|V| = n$ é o número de vértices e $|E| = m$ é o número de arestas de G , com $n \leq m$. Com relação à geração da árvore de custo mínimo de G , AGM_G , assinale a alternativa correta.
- Quando e tem o peso da aresta com o $(n - 1)$ -ésimo menor peso de G então e garantidamente estará numa AGM_G .
 - Quando e tem o peso da aresta com o maior peso em G então e garantidamente não estará numa AGM_G .
 - Quando e tem o peso maior ou igual ao da aresta com o n -ésimo menor peso em G então e pode estar numa AGM_G .
 - Quando e tem o peso distinto do peso de qualquer outra aresta em G então pode existir mais de uma AGM_G .
 - Quando e está num ciclo em G e tem o peso da aresta de maior peso neste ciclo então e garantidamente não estará numa AGM_G .

16. Deseja-se supervisionar as redes de comunicação de dados de um conjunto de empresas. Cada empresa tem sua própria rede, que é independente das redes das outras empresas e é constituída de ramos de fibra óptica. Cada ramo conecta duas filiais distintas (ponto-a-ponto) da empresa. Há, no máximo, um ramo de fibra interligando diretamente um mesmo par de filiais. A comunicação entre duas filiais pode ser feita diretamente por um ramo de fibra que as interliga, se este existir, ou, indiretamente, por meio de uma sequência de ramos e filiais. A rede de cada empresa permite a comunicação entre todas as suas filiais. A tabela 5 apresenta algumas informações acerca das redes dessas empresas.

Quadro 5 – Empresas

empresa	n.º de filiais	n.º de ramos de fibra entre filiais
E1	9	18
E2	10	45
E3	14	13
E4	8	24

Com relação à situação apresentada acima, é correto deduzir que (responda V ou F):

- a) no caso da empresa E1, a falha de um ramo da rede certamente fará que, ao menos, uma filial não possa mais comunicar-se diretamente com todas as outras filiais da empresa;
- b) na rede da empresa E2, a introdução de um novo ramo de rede certamente violará a informação de que há somente um par de fibras entre duas filiais;
- c) no caso da empresa E3, a falha de um único ramo de rede certamente fará que, ao menos, uma filial não possa mais comunicar-se direta ou indiretamente com todas as outras filiais da empresa;
- d) na rede da empresa E4, todas as filiais da empresa comunicam-se entre si diretamente.

7 Caminho mínimo

Um tópico bastante importante da Teoria dos Grafos trata de problemas que envolvem algum tipo de caminhamento na estrutura de um grafo. Nós já abordamos alguns conceitos e aplicações de caminhamento no [Capítulo 3](#). Agora vamos explorar em mais detalhes problemas envolvendo caminhamento em grafos, tais como caminhos mínimos em grafos valorados, bem como os famosos problemas do carteiro chinês e do caixeiro viajante.

7.1 Problema do caminho mínimo

O problema do caminho mínimo em grafos valorados é um dos problemas mais conhecidos e com enorme gama de aplicações. O problema consiste em encontrar um caminho com custo total mínimo entre um vértice inicial e um vértice final. Este tipo de resultado pode ser obtido tanto em grafos dirigidos quanto em grafos não dirigidos, e os algoritmos para isso podem ser aplicados em ambos os tipos de grafos. Antes de ver os algoritmos, precisamos de algumas definições:

Definição 7.1 (Custo de um caminho). *O custo de um caminho $p = \langle v_0, v_1, \dots, v_k \rangle$ entre os vértices v_0 e v_k , denotado por $w(p)$, é igual ao somatório dos custos de todas as arestas valoradas do caminho, ou seja: $w(p) = \sum_{i=1}^k w_{i-1,i}$.*

Definição 7.2 (Custo do caminho mínimo). *O custo do um caminho mínimo do vértice v_i para o vértice v_j é definido por:*

$$\delta_{i,j} = \begin{cases} \min\{w(p) : v_i \rightsquigarrow v_j\} & \text{se } \exists \text{ caminho de } v_i \text{ para } v_j, \\ \infty & \text{caso contrário.} \end{cases}$$

Definição 7.3 (Caminho mínimo). *O caminho mínimo entre dois vértices v_i e v_j é definido como qualquer caminho p com custo igual a $\delta_{i,j}$.*

7.1.1 Algoritmo de Dijkstra

O problema do caminho mínimo é um dos mais conhecidos da Teoria dos Grafos, com inúmeras aplicações práticas, como, por exemplo, encontrar a menor rota entre duas localizações em um mapa viário. A seguir, temos uma definição formal do problema do caminho mínimo.

Definição 7.4 (Problema do caminho mínimo). *Sejam v_i e v_j dois vértices de um grafo valorado conexo. Encontre $\delta_{i,j}$.*

O algoritmo Dijkstra (1959), apresenta uma solução para o problema do caminho mínimo, encontrando todos os caminhos mínimos de um determinado vértice inicial arbitrário para todos os demais vértices do grafo. Com isso, o algoritmo produz uma árvore de caminhos mínimos, tendo o vértice inicial como sua raiz. Assumimos que todas as arestas possuem custos não negativos, pois não há garantia de que o algoritmo encontre uma solução correta no caso de arestas negativas. Ou seja, $w_{i,j} \geq 0$ para cada aresta $(v_i, v_j) \in E$. Note também que, se todas as arestas do grafo tiverem o mesmo custo, o problema se reduz à uma busca em largura (veja a [seção 3.2](#)). Basicamente, o algoritmo funciona da seguinte maneira:

- inicia-se a partir de um vértice de sua escolha. A partir disso, o algoritmo analisa o grafo para encontrar os caminhos mínimos deste vértice para os demais;
- o algoritmo mantém os custos atualmente conhecidos do vértice inicial até todos os demais vértices. À medida que o algoritmo vai executando, estes custos são atualizados;
- em cada iteração, o algoritmo conclui o cálculo final do custo do vértice inicial (origem) para um determinado vértice de destino. Neste momento, o vértice destino precisa de alguma forma ser sinalizado como “visitado” ou “encerrado” e deve estar devidamente posicionado na árvore de caminhos mínimos;
- o processo continua até que todos os vértices do grafo possíveis de serem visitados¹ sejam incluídos na árvore de caminhos mínimos.

Para implementação do algoritmo de caminho mínimo a partir de um vértice origem, um dos pseudocódigos mais populares é apresentado por Cormen et al. (2001). A árvore de caminhos mínimos é construída passo a passo e é estruturada por meio de referências ao vértice predecessor (nó pai) de cada vértice da árvore. O [algoritmo 7.1](#) tem os seguintes elementos:

- s : representa o vértice inicial, origem dos caminhos mínimos;
- d_v : custo do caminho mínimo do vértice s (vértice inicial) até o vértice v . Este custo é inicializado com valor infinito e vai sendo atualizado à medida que o algoritmo analisa o grafo e calcula caminhos mínimos;
- ρ_v : referência ao vértice predecessor do vértice v na árvore de caminhos mínimos;
- S : é o conjunto de todos os vértices cujo caminho mínimo já foi calculado pelo algoritmo;
- Q : é uma fila de prioridades (mínima) de vértices, tendo como chave o valor de d_v . A operação REMOVE-MINIMO(Q) retira da fila e retorna o vértice com menor valor atual de d_v ;

¹ Se o grafo conexo for dirigido, é possível que haja vértices que não possam ser alcançados a partir do vértice inicial escolhido.

- $adj(u)$: é uma iteração com todos os vértices adjacentes ao vértice u (veja seção 2.2);

Algoritmo: Dijkstra(G, w, s)

```

para cada  $v \in V$  faça
   $d_v \leftarrow \infty;$ 
   $\rho_v \leftarrow nil;$ 

   $d_s \leftarrow 0;$ 
   $S \leftarrow \emptyset;$ 
   $Q \leftarrow V;$ 

enquanto  $Q \neq \emptyset$  faça
   $u \leftarrow REMOVE - MINIMO(Q);$ 
   $S \leftarrow S \cup \{u\};$ 
  para cada vértice  $v \in adj(u)$  faça
    se  $v \in Q$  e  $[d_v > (d_u + w_{u,v})]$  então
       $d_v \leftarrow (d_u + w_{u,v});$ 
       $\rho_v \leftarrow u;$ 

```

Algoritmo 7.1: Algoritmo de Dijkstra

Uma operação importante neste pseudocódigo é o relaxamento de aresta. Quando um determinado vértice u é considerado “encerrado”, todos os seus vértices adjacentes v são verificados para fazer o relaxamento de aresta. Isto significa que, ao se calcular o caminho mínimo até u , verifica-se a possibilidade de haver para o vértice v um caminho passando por u que seja mais barato do que o caminho atual para v . Neste caso o custo do caminho para v é atualizado, bem como seu predecessor na árvore de caminhos mínimos, que passa a ser o vértice u .

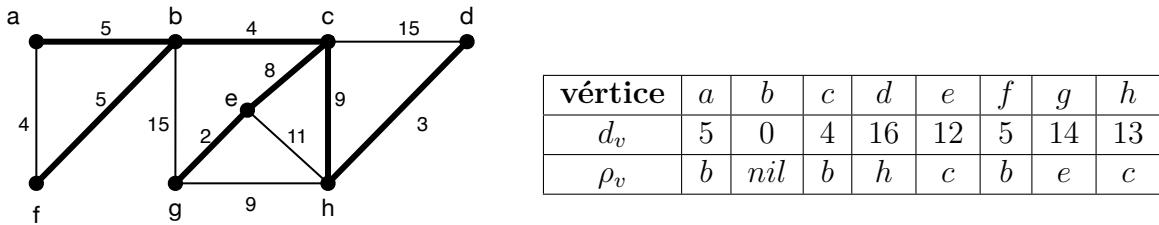
Esta operação ocorre no comando condicional dentro do laço principal do algoritmo: a atualização de custo é realizada pela expressão $d_v \leftarrow (d_u + w_{u,v})$ e a atualização de caminho é feita pela expressão $\rho_v \leftarrow u$.

Exemplo 7.1. A [Figura 7.1](#) mostra um exemplo de árvore de custo mínimo gerada com o algoritmo de Dijkstra, tendo do vértice b como vértice inicial. As arestas que compõem a árvore são aquelas mostradas com linhas mais grossas. A tabela apresenta os valores de d_v e ρ_v calculados pelo algoritmo.

7.1.2 Algoritmo de Floyd

O algoritmo de Floyd utiliza um procedimento incremental sobre a matriz de custos do grafo para encontrar os caminhos mínimos entre todos os pares de vértices. O grafo pode ser dirigido ou não dirigido e pode, inclusive ter arestas com custos negativos, porém não pode ter ciclos com custo total negativo. Este algoritmo foi publicado na sua forma atual

Figura 7.1 – Árvore de caminhos mínimos produzida pelo algoritmo de Dijkstra



Fonte: o autor.

por Robert Floyd ([FLOYD, 1962](#)), mas essencialmente é o mesmo algoritmo publicado por Bernard Roy ([ROY, 1959](#)) e por Stephen Warshall ([WARSHALL, 1962](#)). Por este motivo, o algoritmo também é conhecido como Floyd-Warshall, Roy-Warshall ou Roy-Floyd.

Seja $G = (V, E, W)$ um grafo valorado com n vértices, inicia-se o cálculo dos caminhos mínimos a partir da matriz de custos do grafo. Em seguida, são calculadas n matrizes de distância D de dimensões $n \times n$. Um elemento $d_{i,j}$ em uma matriz de distâncias contém o custo (atual) do caminho mínimo com origem em v_i e destino em v_j . O valor final dos caminhos mínimos vai estar na última das n matrizes geradas.

Para descrever o algoritmo, os índices i e j , representam respectivamente linha e coluna das matrizes, correspondendo aos vértices de origem e destino, com $i = 1, 2, \dots, n$ e $j = 1, 2, \dots, n$. O índice k representando a iteração do algoritmo, com $k = 0, 1, 2, \dots, n$.

As matrizes de distância são definidas da seguinte forma, na qual os elementos da k -ésima matriz são calculados a partir dos elementos da matriz anterior:

$$d_{i,j}^k = \begin{cases} w_{i,j} & \text{se } k = 0, \\ \min(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}) & \text{se } k \geq 1. \end{cases}$$

Note que a expressão $\min(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1})$ promove o mesmo efeito de cálculo de relaxamento de aresta que é feito no algoritmo de Dijkstra.

Sempre que o custo do caminho mínimo entre um determinado par de vértices é atualizado, também é necessário atualizar os dados que mantém os caminhos (ou rotas). Tanto nos algoritmos de busca quanto no algoritmo de Dijkstra, os caminhos formam árvores tendo o vértice inicial na raiz. Foi visto que tais árvores podem ser facilmente armazenadas mantendo-se referências a vértices predecessores em vetores de roteamento.

Como o algoritmo de Floyd gera caminhos tomando cada um dos vértices do grafo como origem, são necessários n vetores de roteamento. Uma forma conveniente de manter estes dados é por meio de Matrizes de Roteamento, denotadas por R . Um elemento $\rho_{i,j}$ de uma matriz de roteamento armazena uma referência para o predecessor do vértice v_j num caminho iniciado no vértice v_i . Em outras palavras, cada linha i da matriz de roteamento

R é um vetor de roteamento armazenando os caminhos com origem no vértice v_i .

O algoritmo de Floyd constrói k matrizes de roteamento. A matriz inicial (para $k = 0$) é dada da seguinte forma:

$$\rho_{i,j}^0 = \begin{cases} \text{nil} & \text{se } i = j \text{ ou } w_{i,j} = \infty, \\ v_i & \text{se } i \neq j \text{ e } w_{i,j} < \infty. \end{cases}$$

Para $k \geq 1$, havendo uma atualização de caminho mínimo passando pelo vértice v_k , ou seja, $v_i \rightsquigarrow v_k \rightsquigarrow v_j$, deve haver a atualização de vértice predecessor:

$$\rho_{i,j}^k = \begin{cases} \rho_{i,j}^{k-1} & \text{se } d_{i,j}^{k-1} \leq d_{i,k}^{k-1} + d_{k,j}^{k-1}, \\ \rho_{k,j}^{k-1} & \text{se } d_{i,j}^{k-1} > d_{i,k}^{k-1} + d_{k,j}^{k-1}. \end{cases}$$

O pseudocódigo do [algoritmo 7.2](#) incorpora as inicializações e atualizações das matrizes D e R para cálculo dos caminhos mínimos entre todos os pares de vértices.

Algoritmo: Foyd(G, W)

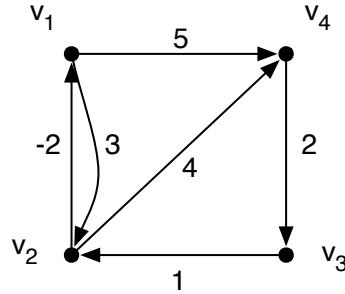
```

para cada  $i \leftarrow 1$  até  $n$  faça
  para cada  $j \leftarrow 1$  até  $n$  faça
     $d_{i,j}^0 \leftarrow w_{i,j};$ 
     $\rho_{i,j}^0 \leftarrow v_i;$ 
  para cada  $k \leftarrow 1$  até  $n$  faça
    para cada  $i \leftarrow 1$  até  $n$  faça
      para cada  $j \leftarrow 1$  até  $n$  faça
        se  $(d_{i,k}^{k-1} + d_{k,j}^{k-1}) < d_{i,j}^{k-1}$  então
           $d_{i,j}^k \leftarrow d_{i,k}^{k-1} + d_{k,j}^{k-1};$ 
           $\rho_{i,j}^k \leftarrow \rho_{k,j}^{k-1};$ 
        senão
           $d_{i,j}^k \leftarrow d_{i,j}^{k-1};$ 
           $\rho_{i,j}^k \leftarrow \rho_{i,j}^{k-1};$ 
      para cada  $j \leftarrow 1$  até  $n$  faça
    para cada  $i \leftarrow 1$  até  $n$  faça
  
```

Algoritmo 7.2: Algoritmo de Floyd

Exemplo 7.2. A Figura 7.2 mostra um exemplo de digrafo valorado com aresta negativa, e as respectivas matrizes de custo e de roteamento geradas pelo algoritmo de Floyd para cada uma das k iterações.

Figura 7.2 – Digrafo valorado com as matrizes D e R



	$v_1 \quad v_2 \quad v_3 \quad v_4$	$v_1 \quad v_2 \quad v_3 \quad v_4$
$k = 0$	$D^0 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 3 & \infty & 5 \\ v_2 & -2 & 0 & \infty & 4 \\ v_3 & \infty & 1 & 0 & \infty \\ v_4 & \infty & \infty & 2 & 0 \end{bmatrix}$	$R^0 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & \text{nil} & v_1 & \text{nil} & v_1 \\ v_2 & v_2 & \text{nil} & \text{nil} & v_2 \\ v_3 & \text{nil} & v_3 & \text{nil} & \text{nil} \\ v_4 & \text{nil} & \text{nil} & v_4 & \text{nil} \end{bmatrix}$
$k = 1$	$D^1 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 3 & \infty & 5 \\ v_2 & -2 & 0 & \infty & 3 \\ v_3 & \infty & 1 & 0 & \infty \\ v_4 & \infty & \infty & 2 & 0 \end{bmatrix}$	$R^1 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & \text{nil} & v_1 & \text{nil} & v_1 \\ v_2 & v_2 & \text{nil} & \text{nil} & v_1 \\ v_3 & \text{nil} & v_3 & \text{nil} & \text{nil} \\ v_4 & \text{nil} & \text{nil} & v_4 & \text{nil} \end{bmatrix}$
$k = 2$	$D^2 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 3 & \infty & 5 \\ v_2 & -2 & 0 & \infty & 3 \\ v_3 & -1 & 1 & 0 & 4 \\ v_4 & \infty & \infty & 2 & 0 \end{bmatrix}$	$R^2 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & \text{nil} & v_1 & \text{nil} & v_1 \\ v_2 & v_2 & \text{nil} & \text{nil} & v_1 \\ v_3 & v_2 & v_3 & \text{nil} & v_1 \\ v_4 & \text{nil} & \text{nil} & v_4 & \text{nil} \end{bmatrix}$
$k = 3$	$D^3 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 3 & \infty & 5 \\ v_2 & -2 & 0 & \infty & 3 \\ v_3 & -1 & 1 & 0 & 4 \\ v_4 & 1 & 3 & 2 & 0 \end{bmatrix}$	$R^3 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & \text{nil} & v_1 & \text{nil} & v_1 \\ v_2 & v_2 & \text{nil} & \text{nil} & v_1 \\ v_3 & v_2 & v_3 & \text{nil} & v_1 \\ v_4 & v_2 & v_3 & v_4 & \text{nil} \end{bmatrix}$
$k = 4$	$D^4 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 3 & 7 & 5 \\ v_2 & -2 & 0 & 5 & 3 \\ v_3 & -1 & 1 & 0 & 4 \\ v_4 & 1 & 3 & 2 & 0 \end{bmatrix}$	$R^4 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & \text{nil} & v_1 & v_4 & v_1 \\ v_2 & v_2 & \text{nil} & v_4 & v_1 \\ v_3 & v_2 & v_3 & \text{nil} & v_1 \\ v_4 & v_2 & v_3 & v_4 & \text{nil} \end{bmatrix}$

Fonte: o autor.

Para interpretar os resultados do algoritmo de Floyd, devemos observar as últimas matrizes de distância e roteamento geradas (neste caso, D^4 e R^4). Por exemplo, na Figura 7.2, o custo do caminho mínimo de v_2 para v_3 está localizado no elemento $d_{2,3}^4$ e seu valor é igual a 5. Já o caminho mínimo entre v_2 e v_3 deve ser obtido na segunda linha da matriz R^4 , o qual é $p : \langle v_2, v_1, v_4, v_3 \rangle$. Devemos começar pelo último vértice do caminho (v_3 neste caso) e a partir dele, encontrar seu predecessor em um caminho que começou em v_2 , ou seja, o elemento $\rho_{2,3}^4 = v_4$. Repete-se o processo sucessivamente até chegar ao vértice inicial do caminho (v_2). A seguir veremos o pseudocódigo para realizar este procedimento.

7.1.3 Imprimindo caminhos na matriz de roteamento

Os caminhos gerados pelo algoritmo de Floyd, armazenados em uma matriz de roteamento R , podem ser reconstituídos por meio do algoritmo 7.3. Note que o algoritmo é recursivo, e pode ser considerado uma generalização do algoritmo 3.5.

```
Algoritmo: ImprimeCaminhoMatriz( $R, v_i, v_j$ )
  se  $v_i = v_j$  então
    | imprime( $v_i$ );
  senão
    se  $\rho_{i,j} = nil$  então
      | imprime("não existe caminho de  $v_i$  para  $v_j$ ");
    senão
      |   ImprimeCaminhoMatriz( $R, v_i, \rho_{i,j}$ );
      |   imprime( $v_j$ );
```

Algoritmo 7.3: Procedimento para imprimir caminhos

7.2 Problema do Carteiro Chinês

O problema do carteiro foi estudado pelo matemático chinês Kwan Mei-Ko em 1962 (KWAN, 1962) e consiste em encontrar um caminho fechado com custo mínimo em um grafo conexo valorado. O termo "carteiro" é uma analogia ao serviço de entrega de correspondência, imaginando que um carteiro precisaria entregar correspondência em todas as ruas de um bairro, voltando ao seu ponto de partida e além disso caminhando a menor distância total possível.

Se o grafo valorado for euleriano, a solução para o problema é trivial. Como um ciclo euleriano percorre todas as arestas somente uma vez, este já seria a rota com custo mínimo escolhida pelo carteiro e o seu custo total é igual à soma dos custos de todas as arestas.

Por outro lado, se o grafo não for euleriano, o requisito de passarmos por todas as ruas (isto é, todas as arestas) faz com que seja necessário definir um trajeto repetindo

algumas ruas. A questão fundamental é decidir quais ruas repetir de forma que se acrescente o menor custo adicional possível.

7.2.1 Eulerização de grafos

Para resolver a questão de repetição das “ruas” na rota do carteiro, vejamos o processo de “eulerização” de grafos não-eulerianos, descrito por [Saoub \(2017\)](#). Existem duas condições para um grafo não ser euleriano: não ser conexo ou não ter todos os seus vértices de grau par. O processo de “eulerização” de grafos conexos, aborda a questão dos graus dos vértices para obter um grafo euleriano a partir de um grafo não-euleriano conexo.

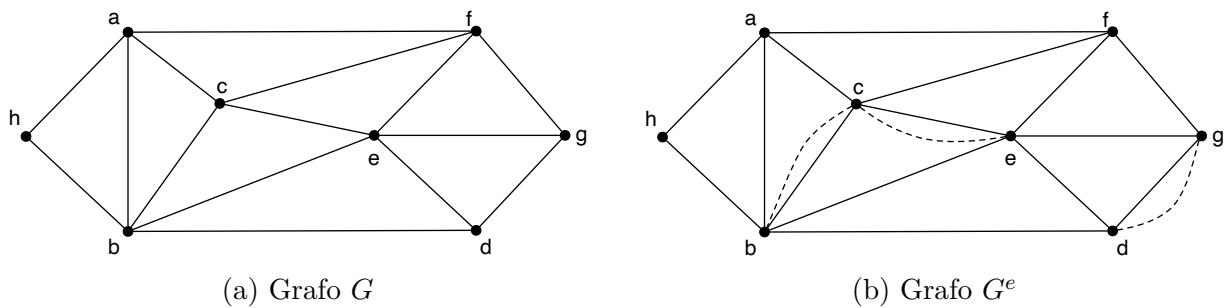
Definição 7.5 (Eulerização). *Dado um grafo conexo $G = (V, E)$, uma Eulerização de G é o grafo $G^e = (V, E^e)$ tal que:*

1. G^e é obtido pela duplicação de arestas de G ; e
2. todos os vértices de G^e possuem grau par.

Uma maneira de se obter a eulerização de um grafo é duplicando as arestas que ligam pares de vértices de grau ímpar. Caso reste algum par de vértices de grau ímpar não adjacentes, deve-se duplicar arestas ao longo de um caminho entre estes dois vértices. Tal caminho é chamado **caminho artificial** e as arestas duplicadas são **arestas artificiais**.

A [Figura 7.3](#) mostra um exemplo deste processo. O grafo G da [Figura 7.3a](#) tem 4 vértices de grau ímpar. Obtemos a eulerização de G mostrada na [Figura 7.3b](#) criando uma aresta artificial entre os vértices g e d , e um caminho artificial entre os vértices e e b , representados pelas arestas tracejadas. Note que os vértices de grau par no meio de um caminho artificial (como o vértice c , por exemplo) continuam tendo grau par após a criação do caminho.

Figura 7.3 – Exemplo de eulerização



Fonte: o autor.

7.2.2 Algoritmo para o problema do carteiro chinês

O pseudocódigo do [algoritmo 7.4](#) encontra a solução do problema do carteiro chinês em grafos valorados determinando um caminho fechado com custo mínimo que passe por todas as arestas. A ideia básica é promover a eulerização do grafo construindo caminhos artificiais com custo mínimo entre pares de vértices de grau ímpar.

Algoritmo: CarteiroChines(G, W)

```

// Inicialização
 $V^{par} \leftarrow \{\text{conjunto dos vértices de grau par}\};$ 
 $G^e \leftarrow G;$ 

// Calcula os caminhos mínimos
Execute  $FLOYD(G, W)$  para construir a matriz de distâncias  $D^n$ ;

// Remove da matriz as linhas e colunas dos vértices de grau par
 $D^{impar} \leftarrow D^n - (\text{linhas e colunas de } V^{par});$ 

// Laço principal
enquanto  $D^{impar} \neq \emptyset$  faça
    Determine em  $D^{impar}$  o par de vértices  $v_i$  e  $v_j$  com menor custo  $d_{i,j}^{impar}$ ;
    Construa um caminho artificial de  $v_i$  para  $v_j$  com custo  $d_{i,j}^{impar}$  no grafo  $G^e$ ;
    // Remove da matriz as linhas e colunas de  $v_i$  e  $v_j$ 
     $D^{impar} \leftarrow D^{impar} - (\text{linhas e colunas de } v_i \text{ e } v_j);$ 

// Encontrando o resultado final
Encontre um ciclo euleriano do grafo  $G^e$ ;
```

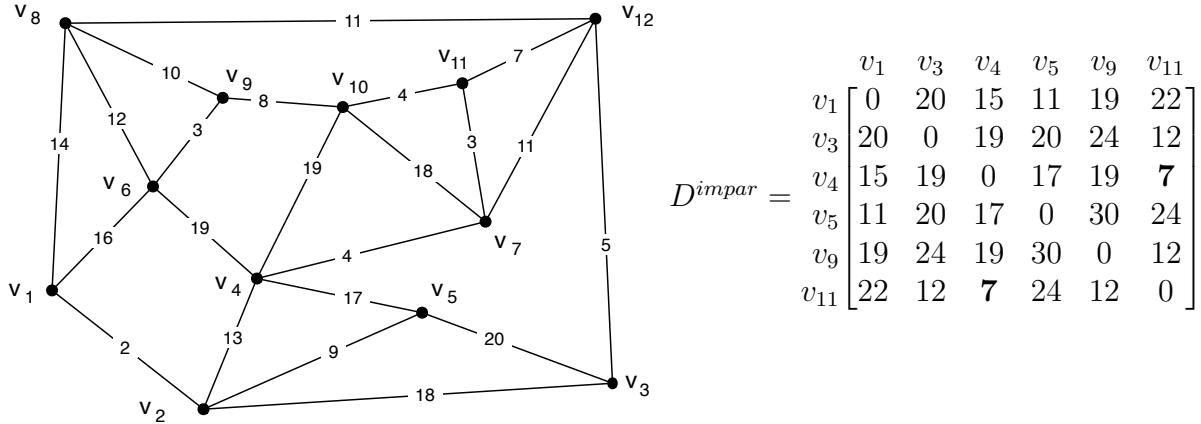
Algoritmo 7.4: Problema do carteiro chinês

Inicialmente, determinamos quais são os vértices de grau par e criamos um grafo G^e que, ao final do algoritmo, será uma eulerização de G com custo mínimo. Em seguida chamamos o algoritmo de Floyd calculamos os caminhos mínimos entre todos os pares de vértices e obtemos a matriz de distâncias (ou custos) mínimas D^n . Como só precisamos encontrar caminhos artificiais entre pares de vértices de grau ímpar, removemos as linhas e colunas de vértices de grau par da matriz D^n para construir uma nova matriz chamada D^{impar} que contém somente as linhas e colunas, e portanto, os caminhos mínimos, dos vértices de grau ímpar.

O laço principal do algoritmo encontra na matriz D^{impar} o par de vértices v_i e v_j com menor custo mínimo atual e constrói no grafo G^e o respectivo caminho artificial. As linhas e colunas correspondentes a v_i e v_j são removidas da matriz D^{impar} e processo se repete enquanto a matriz não estiver vazia. Ao final, o grafo G^e corresponde a uma eulerização do grafo G com custo mínimo e basta encontrar um ciclo euleriano em G^e para obter a solução do problema.

Exemplo 7.3. Dado o grafo $G = (V, E)$ da Figura 7.4, observamos que os vértices v_1 , v_3 , v_4 , v_5 , v_9 e v_{11} possuem grau ímpar, enquanto que os vértices v_2 , v_6 , v_7 , v_8 , v_{10} e v_{12} possuem grau par. Após calcular os caminhos mínimos entre todos os pares de vértices e eliminar da matriz D^n as linhas e colunas referentes aos vértices de grau par, obtemos a matriz $D^{ímpar}$ mostrada ao lado do grafo.

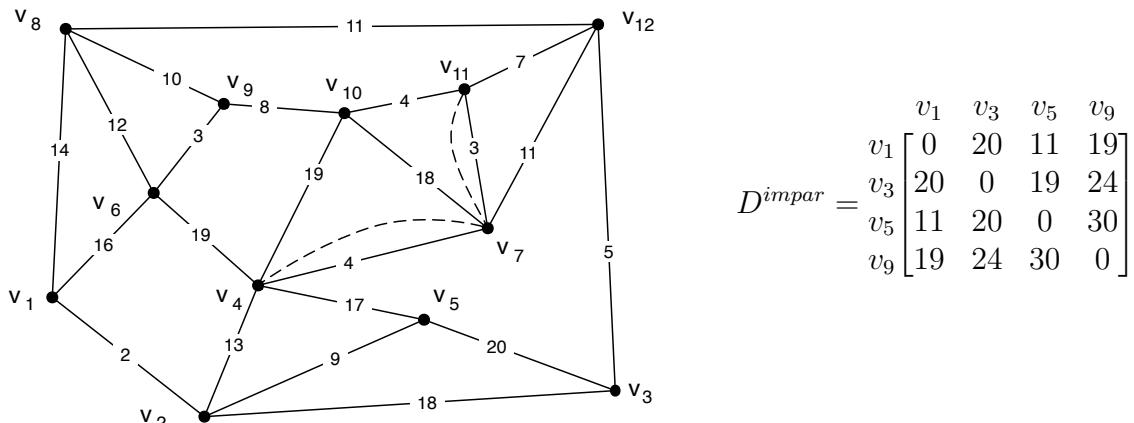
Figura 7.4 – Primeira iteração do carteiro chinês



Fonte: adaptado de Rabuske (1992).

O caminho mínimo entre os vértices v_4 e v_{11} é o menor dentre todos os caminhos mínimos da matriz $D^{ímpar}$. Portanto, é construído o caminho artificial entre estes dois vértices, mostrado em linhas tracejadas na Figura 7.5. As linhas e colunas referentes aos vértices v_4 e v_{11} são removidas da matriz $D^{ímpar}$.

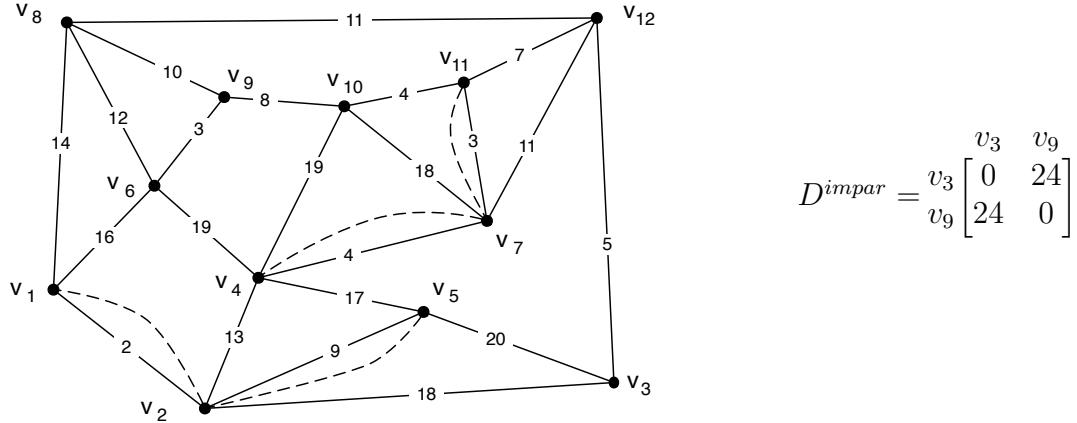
Figura 7.5 – Segunda iteração do carteiro chinês



Fonte: adaptado de Rabuske (1992).

Repetindo o mesmo procedimento, cria-se um caminho artificial com custo 11 entre os vértices v_1 e v_5 . A matriz D^{impar} resultante tem um único caminho mínimo ([Figura 7.6](#)).

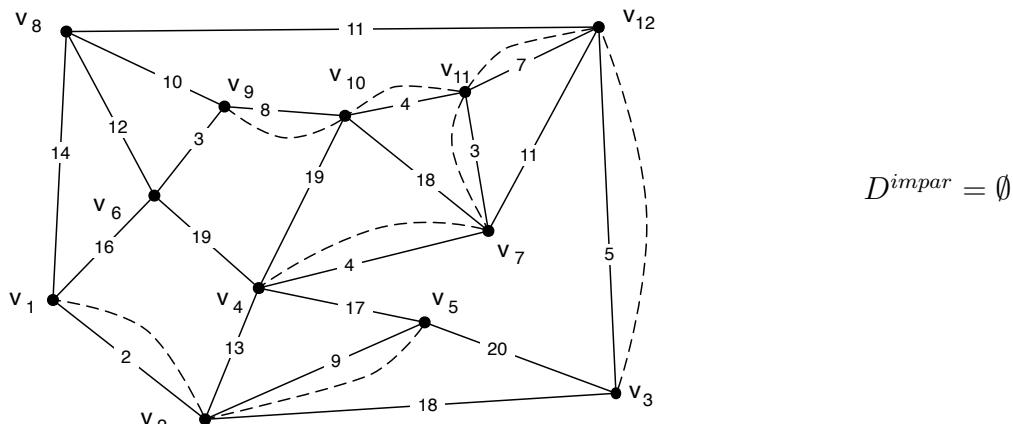
Figura 7.6 – Terceira iteração do carteiro chinês



Fonte: adaptado de [Rabuske \(1992\)](#).

Na figura [Figura 7.7](#), podemos ver o resultado final do algoritmo, cujo laço principal se encerra quando a matriz D^{impar} fica vazia. Note que, considerando a existência das arestas artificiais, o grafo resultante é uma eulerização de G . Por fim, basta encontrar um ciclo euleriano incluindo as arestas artificiais.

Figura 7.7 – Resultado final do carteiro chinês

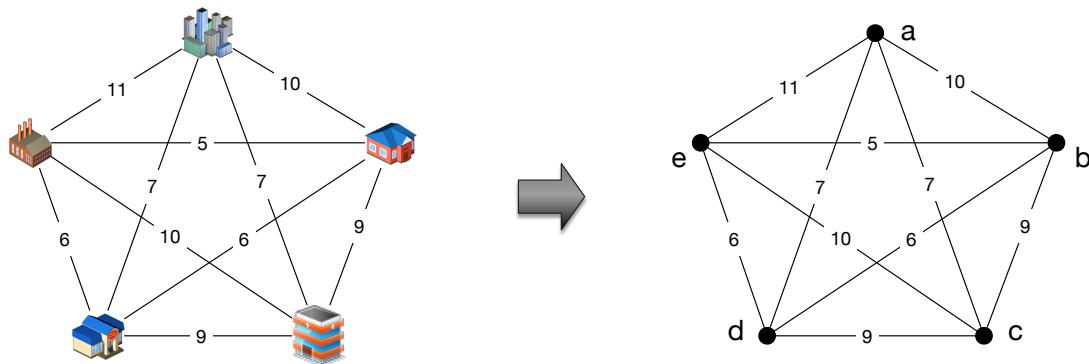


Fonte: adaptado de [Rabuske \(1992\)](#).

7.3 Problema do Caixeiro Viajante

O problema do caixeiro viajante talvez seja um dos problemas mais famosos da computação e da pesquisa operacional. Seu nome se deve ao exemplo básico no qual um “caixeiro viajante”² precisa visitar um determinado número de cidades retornando à sua cidade de origem e gastando o mínimo possível em passagens aéreas. Em sua formulação clássica, existem voos entre todos os pares de cidades e o problema é modelado por um grafo completo valorado, com os vértices representando as cidades e os custos das arestas representando o preço das passagens aéreas entre cada par de cidades. A Figura 7.8 exemplifica a modelagem de uma instância do problema com 5 cidades.

Figura 7.8 – Modelagem do problema do caixeiro viajante



Fonte: o autor.

7.3.1 Algoritmo força bruta

Em resumo, o problema consiste em encontrar um ciclo hamiltoniano de custo mínimo em um grafo completo valorado (SAOUB, 2021). Sabemos que todos os grafos completos com 3 ou mais vértices são hamiltonianos, então poderíamos imaginar um algoritmo do tipo força bruta relativamente simples para resolver o problema: bastaria enumerar todos os ciclos hamiltonianos e escolher aquele com menor custo. Esta solução torna-se inviável porque a quantidade de ciclos hamiltonianos em um grafo completo K_n é igual a $(n - 1)!/2$, levando o algoritmo a ter ordem de complexidade $O(n!)$.

O algoritmo força bruta pode ser realizado pelos seguintes passos:

Entrada: um grafo completo valorado K_n .

Passos:

1. Escolha um vértice inicial arbitrário v ;

² “Caixeiro viajante” é um termo antigo (e já um tanto fora de moda!) para denominar um representante comercial ou vendedor ambulante.

2. Encontre todos os ciclos hamiltonianos iniciando pelo vértice v . Calcule o custo total de cada ciclo;
3. Compare todos os $(n - 1)!/2$ ciclos. Escolha um com o menor custo total.

Saída: um ciclo hamiltoniano com custo mínimo.

O [Quadro 6](#) traz a quantidade de ciclos hamiltonianos em grafos completos em função da sua quantidade de vértices , bem como os tempos aproximados de processamento para encontrar todos os ciclos considerando que cada ciclo levaria 0,2 nanosegundo para ser calculado (0,0000000002 segundo, ou 0,2 ηs). Note que para encontrar um ciclo hamiltoniano em um grafo completo, basta determinar uma permutação de seus vértices.

Quadro 6 – Quantidade de ciclos hamiltonianos em grafos completos

n	ciclos	tempo
1	0	0
5	12	2,4 ηs
10	181.440	36,29 μs
15	43.589.145.600	8,71s
20	$6,08 \times 10^{16}$	140 dias
30	$8,84 \times 10^{20}$	28 trilhões de anos

Fonte: o autor.

7.3.2 Heurística do vizinho mais próximo

Apesar de obter uma solução ótima, algoritmo de força bruta sofre uma explosão combinatória devido à quantidade $O(n!)$ de permutações geradas, tornando-se inviável já com uma quantidade relativamente pequena de vértices. Dada esta dificuldade , podemos recorrer a heurísticas para nos ajudar na escolha entre as várias alternativas de ciclos hamiltonianos buscando soluções não exatas, porém suficientemente boas (isto é, relativamente próximas da solução exata). De fato, [Sipser \(1996\)](#) prova que o problema do caixeiro viajante é \mathcal{NP} -difícil e por isso os algoritmos que rodam rápido não garantem solução ótima e vice-versa ([LAWLER et al., 1985](#)).

Novamente, uma solução relativamente simples é a heurística do vizinho mais próximo ([GROSS; YELLEN; ZHANG, 2013](#)). A ideia básica é iniciar o ciclo hamiltoniano em um vértice qualquer e, a cada passo, percorrer uma aresta escolhendo um novo vértice adjacente com custo mínimo.

O algoritmo com heurística do vizinho mais próximo é descrito pelos seguintes passos:

Entrada: um grafo completo valorado K_n .

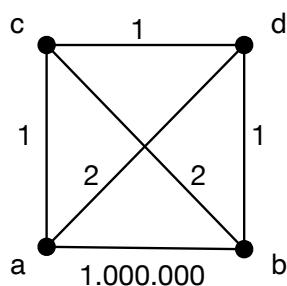
Passos:

1. Escolha um vértice inicial arbitrário v_i . Marque este vértice como visitado;
2. Faça $u \leftarrow v_i$;
3. Dentre todas as arestas incidentes ao vértice u , escolha a aresta (u, v) com menor custo.
Se duas opções diferentes tiverem o mesmo custo escolha uma delas aleatoriamente;
4. Marque a aresta (u, v) como percorrida e vá para o vértice v . Marque o vértice v como visitado;
5. Faça $u \leftarrow v$;
6. Repita os passos (3), (4) e (5), considerando somente arestas para vértices não visitados;
7. Feche o ciclo adicionando a aresta do último vértice visitado v até o vértice inicial v_i ;

Saída: um ciclo hamiltoniano.

A heurística de vizinho mais próximo encontra rapidamente um resultado e é fácil de implementar. Em geral, o algoritmo pode funcionar muito bem, porém não há garantia de que encontre uma solução ótima. Em alguns casos, como explicam Gross, Yellen e Anderson (2018), o algoritmo pode resultar em ciclos hamiltonianos particularmente ruins (isto é, com custo elevado) como pode ser visto no grafo da Figura 7.9. Se iniciarmos pelo vértice a , esta heurística leva ao ciclo hamiltoniano $\langle a, c, d, b, a \rangle$, com custo 1.000.003, enquanto que há um ciclo hamiltoniano neste grafo com custo 6.

Figura 7.9 – Exemplo de resultado ruim com heurística do vizinho mais próximo



Fonte: o autor.

7.3.3 Heurística da desigualdade do triângulo

Em alguns casos particulares é possível utilizar métodos alternativos que eventualmente garantem a qualidade da solução encontrada. O teorema 7.1 enunciado por [Sahni e Gonzalez \(1976\)](#) trata da garantia de desempenho de algoritmos para o problema do caixeiro viajante.

Teorema 7.1. *Se existe um algoritmo aproximado em tempo polinomial cuja solução para cada instância do problema geral do caixeiro viajante nunca é pior do que uma constante r vezes a solução ótima, então $P = NP$.*

Para grafos valorados que satisfazem a **desigualdade do triângulo**, temos o chamado “Problema Métrico do Caixeiro Viajante”, que pode ser resolvido em tempo polinomial com garantia de qualidade da solução obtida. O termo desigualdade do triângulo refere-se ao fato conhecido na geometria que atesta que nenhum lado de um triângulo é maior que a soma dos outros dois lados e é definida da seguinte forma:

Definição 7.6 (Desigualdade do triângulo). *Seja $G = (V, E)$ um grafo simples valorado com vértices v_1, v_2, \dots, v_n , de tal forma que a aresta (v_i, v_j) tem custo c_{ij} . Então G satisfaça a desigualdade do triângulo se $c_{ij} \leq c_{ik} + c_{kj}$ para todos i, j e k .*

O algoritmo para o problema métrico do caixeiro viajante utiliza diversos conceitos vistos nos capítulos anteriores tais como ciclos eulerianos e árvores geradoras de custo mínimo. Para instâncias de grafos que satisfaçam a desigualdade do triângulo, é garantido que o algoritmo a seguir encontra um ciclo hamiltoniano com custo, no pior caso, nunca superior ao dobro do custo ótimo.

Entrada: um grafo completo valorado K_n que satisfaça a desigualdade do triângulo.

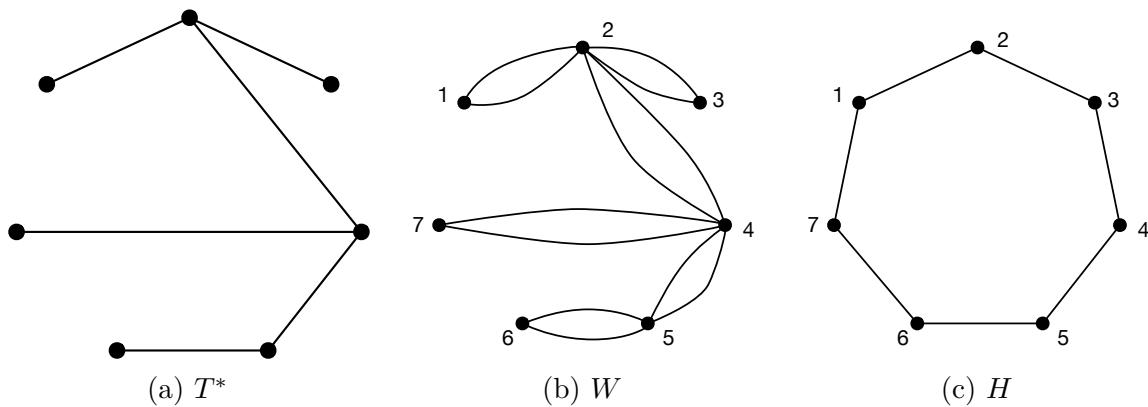
Passos:

1. Encontre uma árvore geradora de custo mínimo T^* do grafo K_n ;
2. Crie um grafo euleriano G^e duplicando todas as arestas de T^* ;
3. Encontre um ciclo euleriano W em G^e ;
4. Construa um ciclo hamiltoniano H em G a partir do ciclo W da seguinte maneira:
sigue a sequência de arestas e vértices de W até que a próxima aresta da sequência esteja conectada a um vértice previamente visitado. Neste ponto, pule para o próximo vértice não visitado utilizando um atalho (ou seja, uma aresta que não faz parte de W). Retome o percorrido de W , pegando atalhos sempre que necessário, até que todos os vértices tenham sido visitados. Complete o ciclo hamiltoniano retornando ao vértice inicial pela aresta que o conecta até o último vértice visitado.

Saída: um ciclo hamiltoniano.

Exemplo 7.4. Suponha que a árvore da Figura 7.10a é uma árvore geradora de custo mínimo T^* de um grafo K_7 valorado (na figura, o grafo original e os custos das arestas foram omitidos). A Figura 7.10b mostra o ciclo euleriano W formado pela duplicação das arestas do grafo, com os números representando a ordem de visitação dos vértices no ciclo euleriano. A Figura 7.10c mostra o ciclo hamiltoniano H resultante quando o ciclo euleriano é modificado com os “atalhos” formados por arestas não pertencentes à árvore geradora de custo mínimo. A desigualdade do triângulo garante que esses atalhos são realmente atalhos.

Figura 7.10 – Exemplo do algoritmo métrico do caixeiro viajante



Fonte: o autor.

7.3.4 Aplicações do problema do caixearo viajante

O problema do caixearo viajante naturalmente surge em muitas aplicações em transportes e logística. Contudo, devido à simplicidade do modelo, ele pode ser aplicado em inúmeras situações práticas em outras áreas. A seguinte lista mostra alguns exemplos de aplicação:

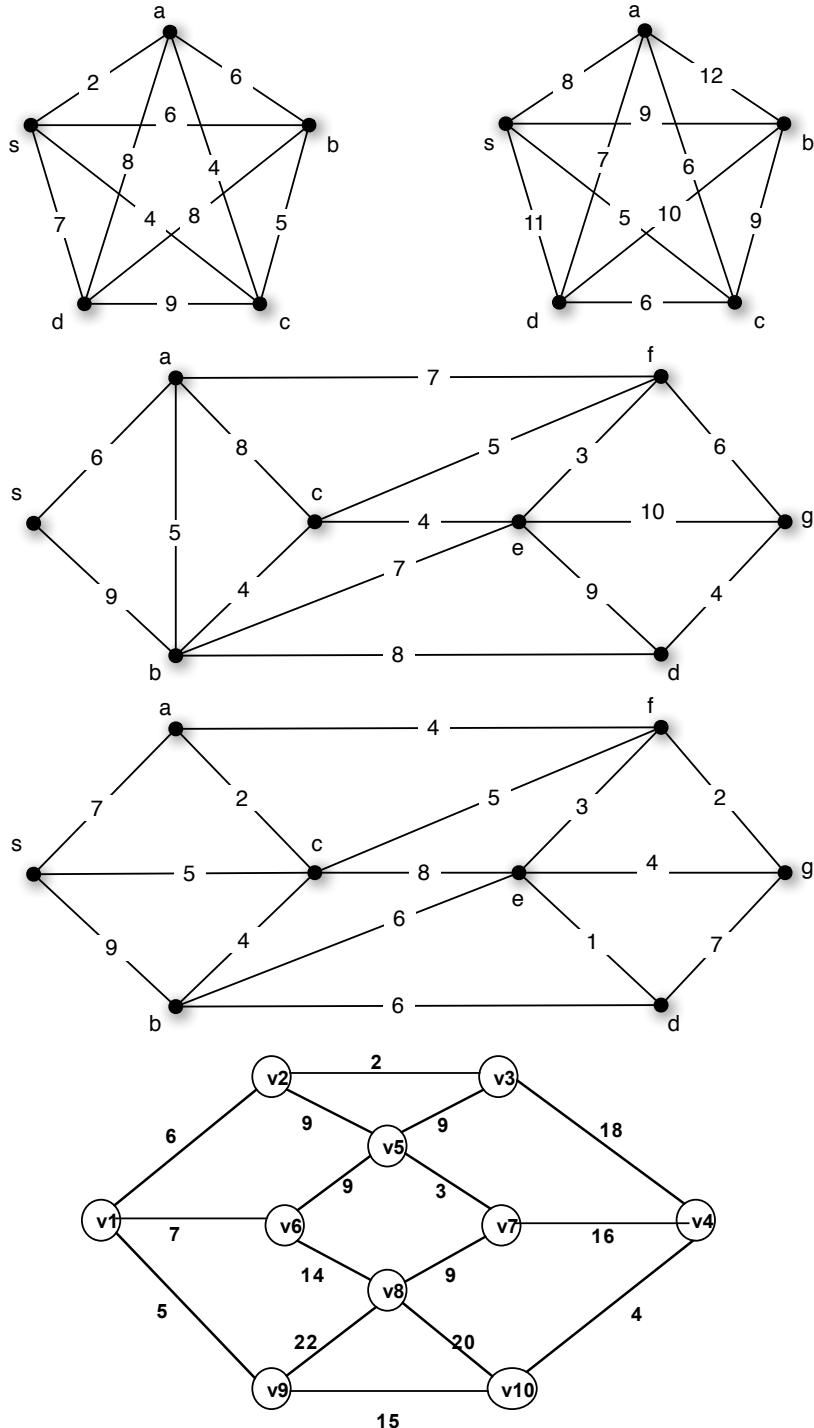
- **Programação de tarefas em série:** suponha que existem n tarefas a serem executadas em série (por exemplo, por uma única máquina). O tempo necessário para preparar a máquina para executar a tarefa j logo após a execução da tarefa i é representado por c_{ij} . Para encontrar uma sequência de todas as tarefas minimizando o tempo total, modelamos um grafo dirigido com n vértices correspondendo às tarefas e as arestas valoradas correspondendo ao tempo de preparação entre as tarefas. A sequência ótima de tarefas corresponde a um ciclo hamiltoniano de custo mínimo.
- **Perfuração de placas:** uma aplicação clássica é a programação de máquinas para perfuração de uma série de furos em placas de circuito impresso ou de qualquer outro objeto. O grafo é modelado com os furos sendo os vértices e as arestas valoradas

representando o tempo de deslocamento da broca entre dois furos. A tecnologia de furação pode variar de uma indústria para outra, mas sempre que o tempo de deslocamento da broca for significativo em relação ao tempo total de furação da placa, o problema do caixeiro viajante pode ter um papel importante nos ganhos de produtividade e redução de custos.

- **Programação de satélites:** o programa da Agência Aeroespacial Norte Americana (NASA) chamado *Starlight Interferometer Program* teve como objetivo prover uma tecnologia para detecção de bioassinaturas no espaço (como água em estado líquido, por exemplo) utilizando imagens geradas por dois satélites operando em conjunto. [Bailey, McLain e Beard \(2001\)](#) publicaram um estudo no qual aplicam o problema do caixeiro viajante para minimizar o uso de combustível para manobrar os satélites nas operações de mira. Os objetos celestes a serem observados foram modelados como vértices e os custos das arestas valoradas representam a quantidade de combustível necessária para reposicionar a mira dos dois satélites de um objeto celeste para outro.
- **Programação de rota de entrega:** esta é uma das aplicações mais conhecidas, na qual os vértices são os pontos de entrega e as arestas representam o tempo (ou distância) entre dois pontos. Um exemplo com importância histórica é a programação de rota de ônibus escolares, que motivou [Flood e Savage \(1948\)](#), pioneiros da pesquisa operacional, a cunharem o termo “Problema do Caixeiro Viajante”.

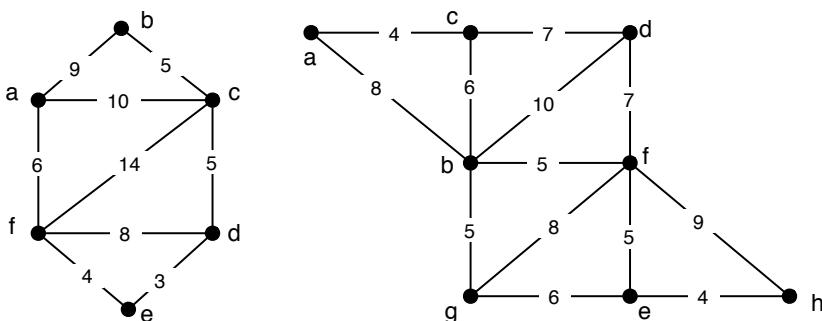
7.4 Exercícios

1. Dados os grafos abaixo, utilize o algoritmo de Dijkstra para calcular os caminhos mínimos entre todos os pares de vértices. O algoritmo deverá ser usado n vezes, uma vez para cada vértice como origem.



2. Implemente o algoritmo de Floyd em uma linguagem de programação de sua preferência.

3. Nos grafos do exercício 1, encontre os caminhos mínimos entre todos os pares de vértices utilizando o algoritmo de Floyd. Construa também as matrizes de roteamento.
4. Para os cinco grafos do exercício 1, resolva o Problema do Carteiro Chinês, mostrando as principais etapas para chegar à solução, conforme mostrado nos slides de aula.
5. Para cada um dos grafos abaixo, resolva o Problema do Carteiro Chinês, mostrando as principais etapas para chegar à solução, conforme mostrado nos slides de aula.



6. (PosComp – 2007) Considere o problema do caixeiro viajante, definido como se segue. Seja S um conjunto de n cidades (com $n \geq 0$) e $d_{ij} > 0$ a distância entre as cidades i e j , $i, j \in S$, $i \neq j$. Define-se um percurso fechado como sendo um percurso que parte de uma cidade $i \in S$, passa exatamente uma vez por cada cidade de $S - i$, e retorna à cidade de origem. A distância de um percurso fechado é definida como sendo a soma das distâncias entre cidades consecutivas no percurso. Deseja-se encontrar um percurso fechado de distância mínima. Suponha um algoritmo guloso que, partindo da cidade 1, move-se para a cidade mais próxima ainda não visitada e que repita esse processo até passar por todas as cidades, retornando à cidade 1. Considere as seguintes afirmativas.

- I. Todo percurso fechado obtido com esse algoritmo tem distância mínima.
- II. O problema do caixeiro viajante pode ser resolvido com um algoritmo de complexidade linear no número de cidades.
- III. Dado que todo percurso fechado corresponde a uma permutação das cidades, existe um algoritmo de complexidade exponencial no número de cidades para o problema do caixeiro viajante.

Em relação a essas afirmativas, pode-se afirmar que:

- a) I é falsa e III é correta.
- b) I, II e III são corretas.
- c) apenas I e II são corretas.
- d) apenas I e III são falsas.
- e) I, II e III são falsas.

7. (PosComp – 2014) Assinale a alternativa que apresenta, corretamente, o algoritmo utilizado para determinar o caminho mínimo entre todos os pares de vértices de um grafo.
- a) Bellman-Ford.
 - b) Floyd-Warshall.
 - c) Dijkstra.
 - d) Kruskal.
 - e) Prim.

Lista de ilustrações

Figura 1.1 – Gravura da cidade de Königsberg, no século XVIII	7
Figura 1.2 – Diagrama representando o problema das pontes de Königsberg	8
Figura 1.3 – Exemplo de grafo não dirigido	9
Figura 1.4 – Grafo não dirigido - incidência e adjacência	10
Figura 1.5 – Exemplo de grafo dirigido (ou digrafo)	10
Figura 1.6 – Adjacência de vértices em digrafos.	11
Figura 1.7 – Exemplo de modelagem de sistema de trânsito	11
Figura 1.8 – Grafo subjacente a um grafo dirigido	12
Figura 1.9 – Grafo transposto a um grafo dirigido	12
Figura 1.10–Laços e arestas paralelas	13
Figura 1.11–Grafos denso e esparsos	14
Figura 1.12–Grafos rotulado e não-rotulado	15
Figura 1.13–Isomorfismo $G \simeq H$	16
Figura 1.14–Dois subgrafos de um grafo G	17
Figura 1.15–Grafo de Petersen	18
Figura 1.16–Grafos nulos	19
Figura 1.17–Grafos completos	19
Figura 1.18–Grafos ciclo	20
Figura 1.19–Grafos bipartidos	20
Figura 1.20–Grafos bipartidos completos	21
Figura 1.21–Grafo cubo Q_3	21
Figura 1.22–Percorso em um grafo	22
Figura 1.23–Percorso dirigido em um grafo	23
Figura 1.24–Caminho, trilha, ciclo e circuito em um grafo	24
Figura 1.25–Exemplo menor caminho	24
Figura 1.26–Exemplo de modelagem de labirinto	25
Figura 1.27–Instância do problema dos 4 cubos	26
Figura 1.28–Instância do problema dos 4 cubos com solução trivial	26
Figura 1.29–Instância do problema dos 4 cubos sem solução	27
Figura 1.30–Modelagem de um grafo para cada cubo	27
Figura 1.31–União dos grafos dos 4 cubos	28
Figura 1.32–Solução do problema dos 4 cubos	28
Figura 1.33–Exemplo sistema de ruas de mão única	30
Figura 1.34–Exercício labirintos	30
Figura 1.35–Exercício sobre isomorfismo	31
Figura 1.36–Exercício isomorfismo	31

Figura 1.37–Exercício isomorfismo	32
Figura 1.38–Exercício isomorfismo	32
Figura 1.39–Exercício sequência de graus	32
Figura 1.40–Grafos conexos não rotulados com até 5 vértices	33
Figura 1.41–Exercício 4 cubos	35
Figura 1.42–Exercício 4 cubos sem solução	36
Figura 1.43–Exercício 4 cubos com solução única	36
Figura 1.44–Problema dos oito círculos	36
Figura 2.1 – Representação por lista de arestas de um grafo simples	44
Figura 2.2 – Listas de adjacência de um grafo simples	45
Figura 2.3 – Listas de adjacência de um grafo dirigido	45
Figura 2.4 – Mapas de adjacência de um grafo dirigido	46
Figura 2.5 – Matriz de adjacência de grafo simples	47
Figura 2.6 – Matriz de adjacência de grafo simples dirigido	48
Figura 2.7 – Matriz de adjacência de grafo simples	48
Figura 2.8 – Matriz de adjacência de grafo simples dirigido	48
Figura 3.1 – Exemplo de árvore de busca em largura	55
Figura 3.2 – Estados inicial e final do problema	60
Figura 3.3 – Grafo de estados do robô	61
Figura 3.4 – Exemplo de grafo acíclico dirigido	62
Figura 3.5 – Grafo acíclico dirigido	63
Figura 3.6 – Resultado da ordenação topológica	63
Figura 4.1 – Exemplo de conexidade	69
Figura 4.2 – Exemplo de grafo subjacente a um digrafo	70
Figura 4.3 – Exemplo de remoção de vértice	71
Figura 4.4 – Exemplo de subtração de grafos	71
Figura 4.5 – Exemplo de remoção de aresta	72
Figura 4.6 – Exemplo de corte de vértices	72
Figura 4.7 – Exemplo de vértice de corte	73
Figura 4.8 – Grafo G com $\kappa(G) = 2$ e $\gamma(G) = 3$	74
Figura 4.9 – Operação de fusão de vértices	76
Figura 4.10–Exemplo conjuntos disjuntos com árvores	78
Figura 4.11–Exemplo de formação de conjuntos disjuntos	80
Figura 4.12–Exemplo de grafos dirigidos forte e fracamente conexos	81
Figura 4.13–Exemplo de um grafo dirigido e seu grafo transposto	82
Figura 4.14–Obtenção de componentes fortemente conexas	83
Figura 4.15–grafo acíclico dirigido de componentes fortemente conexas	83
Figura 5.1 – Problemas do explorador e do turista	87
Figura 5.2 – Prova do teorema de ciclos disjuntos	90

Figura 5.3 – Ciclos disjuntos de arestas	90
Figura 5.4 – Grafo semi-euleriano	91
Figura 5.5 – Formação de ciclos disjuntos	93
Figura 5.6 – Exemplo de jogo de dominó	96
Figura 5.7 – Grafo do jogo de dominó	97
Figura 5.8 – Modelagem de instância do jogo de dominó	98
Figura 5.9 – O jogo icosiano	99
Figura 5.10–Aplicação do teorema de Ore	100
Figura 5.11–Movimentos do cavalo no tabuleiro de xadrez	101
Figura 5.12–Modelagem do tabuleiro 4×4	101
Figura 5.13–Solução do problema do percurso do cavalo	102
Figura 5.14–Círculo dividido em 8 setores de 45°	102
Figura 5.15–Grafo Q_3 e Gray codes de 3 bits	103
Figura 5.16–Exemplo de movimentos válidos de um cavalo.	107
Figura 6.1 – Árvores não rotuladas com até 5 vértices	111
Figura 6.2 – Construção de nova árvore não rotulada	112
Figura 6.3 – Formação de um ciclo	112
Figura 6.4 – Remoção de aresta de uma árvore	113
Figura 6.5 – Adição de uma nova aresta em uma árvore	113
Figura 6.6 – Exemplo de Floresta	114
Figura 6.7 – Árvore dirigida	115
Figura 6.8 – Árvores geradoras do grafo de Petersen	115
Figura 6.9 – Árvores geradora pelo método construtivo	116
Figura 6.10–Árvores geradora por redução	116
Figura 6.11–Adicionando rigidez a uma treliça plana	117
Figura 6.12–Modelagem de treliças	117
Figura 6.13–Treliça sem rigidez	118
Figura 6.14–Treliça rígida com o mínimo de barras diagonais	118
Figura 6.15–Exemplos de grafos valorados	119
Figura 6.16–Problema de pavimentação de estradas	120
Figura 6.17–Duas soluções do problema de pavimentação de estradas	121
Figura 6.18–Desenvolvimento do algoritmo de Prim	122
Figura 6.19–Grafo valorado para o algoritmo de Kruskal	125
Figura 6.20–Exemplo de execução do algoritmo de Kruskal	127
Figura 7.1 – Árvore de caminhos mínimos produzida pelo algoritmo de Dijkstra	136
Figura 7.2 – Dígrafo valorado com as matrizes D e R	138
Figura 7.3 – Exemplo de eulerização	140
Figura 7.4 – Primeira iteração do carteiro chinês	142
Figura 7.5 – Segunda iteração do carteiro chinês	142

Figura 7.6 – Terceira iteração do carteiro chinês	143
Figura 7.7 – Resultado final do carteiro chinês	143
Figura 7.8 – Modelagem do problema do caixeiro viajante	144
Figura 7.9 – Exemplo de resultado ruim com heurística do vizinho mais próximo . .	146
Figura 7.10–Exemplo do algoritmo métrico do caixeiro viajante	148

Listas de quadros

Quadro 1 – Operações mais comuns em estruturas de dados de Grafos	43
Quadro 2 – Exemplo de vetor de roteamento $R(G)$	55
Quadro 3 – Sequência de arestas ordenadas	125
Quadro 4 – Formação dos conjuntos disjuntos	126
Quadro 5 – Empresas	132
Quadro 6 – Quantidade de ciclos hamiltonianos em grafos completos	145

Referências

- AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D. *Data Structures and Algorithms*. 1st. ed. Reading, Massachusetts: Addison-Wesley, 1983. Paperback. (Computer Science and Information Processing). ISBN 0201000237. Disponível em: <<http://www.worldcat.org/isbn/0201000237>>. Citado na página 82.
- ALDOUS, J.; BEST, S.; WILSON, R. *Graphs and Applications: An Introductory Approach*. Springer London, 2003. (The Open University). ISBN 9781852332594. Disponível em: <https://books.google.com.br/books?id=1qRvTI__oWUAC>. Citado 5 vezes nas páginas 25, 26, 27, 89 e 102.
- BAILEY, C. A.; MCLAIN, T. W.; BEARD, R. W. Fuel-saving strategies for dual spacecraft interferometry missions. *The Journal of the Astronautical Sciences*, v. 49, n. 3, p. 469–488, Sep 2001. ISSN 2195-0571. Disponível em: <<https://doi.org/10.1007/BF03546233>>. Citado na página 149.
- BIGGS, N.; LLOYD, E. K.; WILSON, R. J. *Graph Theory, 1736-1936*. New York, NY, USA: Clarendon Press, 1986. ISBN 0-198-53916-9. Citado na página 7.
- CHARTRAND, G. *Introductory Graph Theory*. Dover, 1977. (Dover Books on Mathematics Series). ISBN 9780486247755. Disponível em: <<https://books.google.com.br/books?id=rYuToT7vHbMC>>. Citado na página 101.
- CORMEN, T. H. et al. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001. Paperback. ISBN 0262531968. Disponível em: <<http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{&}path=ASIN/0262531>>. Citado 8 vezes nas páginas 55, 56, 58, 63, 77, 122, 124 e 134.
- DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische mathematik*, Springer, v. 1, n. 1, p. 269–271, 1959. Citado na página 134.
- EULER, L. Solutio problematis ad geometriam situs pertinentis. *Commentarii academieae scientiarum Petropolitanae*, p. 128–140, 1741. Citado na página 88.
- FLEURY, P.-H. Deux problèmes de géométrie de situation. *Journal de mathématiques élémentaires*, C. Delagrave., v. 2, p. 257–261, 1883. Disponível em: <<https://books.google.com.br/books?id=l-03AAAAMAAJ>>. Citado na página 91.
- FLOOD, M. M.; SAVAGE, L. J. *A Game Theoretic Study of the Tactics of Area Defense*. Santa Monica, CA: RAND Corporation, 1948. Citado na página 149.
- FLOYD, R. W. Algorithm 97: Shortest path. *Communications of the ACM*, ACM Press, New York, NY, USA, v. 5, n. 6, p. 345, 1962. ISSN 0001-0782. Citado na página 136.
- GOODMAN, S.; HEDETNIEMI, S. *Introduction to the Design and Analysis of Algorithms*. [S.l.]: McGraw-Hill, 1977. (J. Ranade Workstation Series, v. 1). ISBN 9780070237537. Citado na página 75.

GOODRICH, M. T.; TAMASSIA, R.; GOLDWASSER, M. H. *Data Structures and Algorithms in Python*. 1st. ed. [S.l.]: Wiley Publishing, 2013. ISBN 1118290275. Citado 3 vezes nas páginas [44](#), [45](#) e [46](#).

GROSS, J. L.; YELLEN, J.; ANDERSON, M. *Graph Theory and Its Applications*. 3rd. ed. [S.l.]: Chapman & Hall/CRC, 2018. ISBN 1482249480. Citado na página [146](#).

GROSS, J. L.; YELLEN, J.; ZHANG, P. *Handbook of Graph Theory, Second Edition*. 2nd. ed. [S.l.]: Chapman & Hall/CRC, 2013. ISBN 1439880182. Citado na página [145](#).

HIERHOLZER, C.; WIENER, C. Ueber die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, v. 6, n. 1, p. 30–32, 1873. Disponível em: <<https://doi.org/10.1007/BF01442866>>. Citado na página [92](#).

HOPCROFT, J.; TARJAN, R. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 16, n. 6, p. 372–378, jun. 1973. ISSN 0001-0782. Disponível em: <<https://doi.org/10.1145/362248.362272>>. Citado na página [74](#).

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321455363. Citado na página [60](#).

JARNÍK, V. *Über ein Minimalproblem*. 1931. Práce moravské přírodovědecké společnosti 6, 57-63 (1931). Citado na página [121](#).

KERZNER, H. *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*. USA: John Wiley & Sons, Inc., 2005. ISBN 0471741876. Citado na página [62](#).

KOAY, W.; KREHER, D. L. *Graphs, Algorithms and Optimization*. [S.l.]: Chapman & Hall/CRC, 2004. ISBN 1584883960. Citado na página [93](#).

KRUSKAL, J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In: *Proceedings of the American Mathematical Society*, 7. [S.l.: s.n.], 1956. Citado na página [123](#).

KWAN, M. ko. Graphic programming using odd or even points. *Chinese Mathematics*, v. 1, p. 273–277, 1962. Citado na página [139](#).

LAWLER, E. L. et al. *The traveling salesman problem : a guided tour of combinatorial optimization*. Chichester: Wiley, 1985. (Wiley-Interscience series in discrete mathematics). ISBN 0471904139. Citado na página [145](#).

ORE, O. Note on hamilton circuits. *The American Mathematical Monthly*, Mathematical Association of America, v. 67, n. 1, p. 55–55, 1960. ISSN 00029890, 19300972. Disponível em: <<http://www.jstor.org/stable/2308928>>. Citado na página [99](#).

ORE, O.; WILSON, R. J. *Graphs and Their Uses*. [S.l.]: Mathematical Association of America, 1990. ISBN 0883856352. Citado na página [91](#).

- PAOLETTI, T. Leonard euler's solution to the konigsberg bridge problem. *Convergence*, Mathematical Association of America, v. 3, 2006. Disponível em: <<https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem>>. Citado na página 7.
- PETERSEN, J. Sur le théorème de tait. *L'Intermédiaire des Mathématiciens*, v. 5, p. 225–227, 1898. Citado na página 18.
- PRIM, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, v. 36, n. 6, p. 1389–1401, 1957. Citado na página 121.
- RABUSKE, M. A. *Introdução à Teoria dos Grafos*. [S.l.]: Editora da UFSC, 1992. Citado 3 vezes nas páginas 92, 142 e 143.
- ROY, B. Transitivité et connexité. *C. R. Acad. Sci. Paris*, v. 249, p. 216–218, 1959. Citado na página 136.
- SAHNI, S.; GONZALEZ, T. P-complete approximation problems. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 23, n. 3, p. 555–565, jul 1976. ISSN 0004-5411. Disponível em: <<https://doi.org/10.1145/321958.321975>>. Citado na página 147.
- SAOUB, K. R. *A Tour through Graph Theory*. [S.l.]: CRC Press, 2017. (Textbooks in Mathematics). ISBN 9781138197817. Citado na página 140.
- SAOUB, K. R. *Graph Theory: An Introduction to Proofs, Algorithms, and Applications*. [S.l.]: CRC Press, 2021. (Textbooks in Mathematics). ISBN 9780429779886. Citado na página 144.
- SBC. *POSCOMP*: exame nacional para ingresso na pós-graduação em computação. 2022. Disponível em: <https://www.sbc.org.br/index.php?option=com_content&view=article&layout=edit&id=458>. Acesso em: 24 jun 2022. Citado na página 6.
- SIPSER, M. *Introduction to the Theory of Computation*. 1st. ed. [S.l.]: International Thomson Publishing, 1996. ISBN 053494728X. Citado na página 145.
- STEVENS, W. R. *TCP/IP Illustrated (Vol. 1): The Protocols*. USA: Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN 0201633469. Citado na página 115.
- WARSHALL, S. A theorem on boolean matrices. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 9, n. 1, p. 11–12, jan 1962. ISSN 0004-5411. Disponível em: <<https://doi.org/10.1145/321105.321107>>. Citado na página 136.

Índice

- adjacência, 9, 11
 - listas de, 44
 - mapas de, 46
 - matriz de, 47
- Algoritmo
 - BFS, 55
 - ComponentesConexasConjuntos, 79
 - ConexidadeDFS, 75
 - ConexidadeFusao, 75
 - de Dijkstra, 133
 - de Fleury, 91
 - de Floyd, 135
 - de Hierholzer, 92
 - de Kruskal, 123
 - de Prim, 121
 - DFS, 57
 - do carteiro chinês, 141
 - ImprimeCaminho, 59
 - ImprimeCaminhoMatriz, 139
- aresta
 - artificial, 140
 - de corte, 73
 - subtração de, 72
- arestas
 - listas de, 42
 - paralelas, 12
- arvore, 111
 - de busca em largura, 54
 - de caminhos mínimos, 135
 - definição, 111
 - dirigida, 114
 - geradora, 115
 - geradora de custo mínimo, 120
- busca
 - em largura, 54
 - em profundidade, 57
- Caixeiro Viajante, 144
 - algoritmo força bruta, 144
 - aplicações, 148
 - heurística da desigualdade do triângulo, 147
 - heurística do vizinho mais próximo, 145
 - caminho, 23
 - artificial, 140
 - custo de, 133
 - mínimo, 24, 133
 - semi-euleriano, 91
 - trivial, 23
 - Carteiro Chinês, 139
 - ciclo, 23
 - euleriano, 89
 - hamiltoniano, 98
 - ciclos
 - disjuntos, 89
 - circuito, 23
 - componente conexa, 70
 - componente fortemente conexa, 81
 - comprimento, 22
 - conectividade
 - de arestas, 74
 - de vértices, 73
 - conexidade, 69
 - em grafo dirigido, 81
 - conjuntos disjuntos, 77
 - corte
 - de arestas, 73
 - de vértices, 72
 - custo
 - matriz de, 119

- densidade, 14
 desigualdade do triângulo, 147
 digrafo, 10
 conexo, 70
 dominó, 96
 endpts, 9
 euleriano
 ciclo, 88
 grafo, 88
 eulerização, 140
 floresta, 114
 fusão de vértices, 75
 grafo, 8
 acíclico dirigido, 62
 bipartido, 20
 bipartido completo, 21
 ciclo, 20
 completo, 19
 conexo, 70
 de estados, 60
 dirigido, 10
 euleriano, 89
 fortemente conexo, 81
 hamiltoniano, 98
 nulo, 18
 regular, 18
 semi-euleriano, 91
 simples, 13
 subjacente, 70
 subtração de, 71
 transposto, 81
 valorado, 119
 grau de vértice, 14
 gray codes, 102
 hamiltoniano
 ciclo, 98
 grafo, 98
 incidência, 9
 função de, 9
 isomorfismo, 16
 k-conexão, 73
 k-conexão-de-arestas, 74
 Kruskal
 Algoritmo de, 123
 Könisberg
 problema das pontes de, 7
 labirinto, 25
 laço, 12
 Lema do aperto de mão, 88
 listas de adjacência, 44
 listas de arestas, 42
 mapas de adjacência, 46
 matriz
 de adjacência, 47
 de custos, 119
 de roteamento, 137
 multigrafo, 13
 ordem, 13
 ordenação topológica, 61
 Ore
 Teorema de, 99
 percurso, 22
 dirigido, 23
 fechado, 22
 trivial, 23
 ponte, 73
 predecessores
 vetor de, 55
 Prim
 Algoritmo de, 121
 Problema
 das pontes de Könisberg, 7
 do Caixeiro Viajante, 144
 do caminhamento do cavalo, 100

- do Carteiro Chinês, 139
- do explorador, 87
- do turista, 87
- dos quatro cubos, 25
- roteamento
 - matriz de, 136, 137
 - vetor de, 55
- semi-euleriano
 - caminho, 91
 - grafo, 91
- Sequência de graus, 14
- sub-grafo
 - de remoção de aresta, 71
- subgrafo, 17
 - de remoção de vértice, 70
- tamanho, 13
- Teorema de Ore, 99
- treliças planas, 116
- trilha, 23
 - trivial, 23
- vértice
 - de corte, 73
 - grau de, 14
 - subtração de, 71
- vértices
 - fusão de, 75
- xadrez, 100

O Autor

Paulo César Rodacki Gomes é professor do Instituto Federal de Educação, Ciência e Tecnologia Catarinense – Campus Blumenau (IFC). Obteve seu Doutorado em Informática pela PUC-Rio em 1999, e foi pesquisador do ICAD/VisionLab (Laboratório de Pesquisa e Desenvolvimento em Visualização, TV Digital/Cinema e Jogos). Logo após a conclusão do doutorado, trabalhou no Departamento de Engenharia de Multimídia da Rede Globo de Televisão (RJ). De 2001 a 2012 foi professor do Departamento de Sistemas e Computação da FURB - Universidade de Blumenau, onde atuou nos cursos de Bacharelado em Ciência da Computação e Mestrado em Ensino de Ciências e Matemática. Em 2012 ingressou no quadro de professores do IFC, onde leciona disciplinas nas áreas de Programação de Computadores, Estruturas de Dados, Teoria dos Grafos e Teoria da Computação.

Grafos são poderosas abstrações matemáticas utilizadas para modelar e resolver uma grande variedade de problemas na Ciência da Computação, Engenharias, Ciências Naturais e Sociais entre outras.

A presente obra traz tópicos introdutórios da Teoria dos Grafos, e pode ser indicada como bibliografia básica inicial em cursos de graduação em Ciência da Computação e áreas afins, podendo ser útil também em cursos de pesquisa operacional, matemática discreta e estruturas de dados.

O conteúdo é abordado em três aspectos principais: (i) fundamentação teórica, (ii) aspectos relativos à implementação computacional de algoritmos e (iii) aplicação da teoria para a resolução de problemas. O conteúdo está dividido em sete capítulos, iniciando com conceitos fundamentais, em seguida abordando representação computacional de grafos como estruturas de dados e também algoritmos de busca em grafos; conexidade; grafos eulerianos e hamiltonianos; árvores e problemas relativos ao caminhamento em grafos tais como problema do caminho mínimo e problema do caixeiro viajante. Ao final de cada capítulo, são propostos exercícios para melhor entendimento e fixação do conteúdo.